COMP.SE.140 Project - End report

- Author: Ville Penttinen

Instructions on how to run the application and its tests may be found in README.md

Total amount of hours used for the project was about 70 hours in total, this is the time spent on the project, time spent on the original exercise is not included. Implementation took about 60 hours and rest of the time was spent in writing the README and EndReport and documenting things such as the `.gitlab-ci.yml`.

Overview of CI/CD pipeline

This section gives an overview of the CI/CD pipeline utilizing Gitlab `.gitlab-ci.yml`. The full contents of the file are also available in Appendix A.

The Gitlab CI/CD pipeline is configured to utilize Docker engine in the Gitlab runner. This provides flexibility in allowing testing of the pipeline locally and makes the environment reproducible.

The pipeline is split into three stages:

1. build
2. test
3. deploy

The stages are executed in the order above. Each stage may contain multiple jobs, but in this case each stage contains only a single job. Next we'll cover the stages in more detail.

Build-stage

In the `build`-stage the docker-images are built using a custom shell-script. The shell script takes advantage of the multi-stage Dockerfile to first build testrunner-stage. This testrunner-stage can then be used as a cache for the subsequent application images. This ensures that the dependencies are restored only once. Additionally the full solution is also built only once. The testrunner-stage can be utilized directly to execute the unit and end-to-end tests.

After build-stage has been successfully completed, the built-images are tested in the test-stage.

Test-stage

In the test-stage tests are run. The application contains various several tests of various kinds. All of these are run in the test-job using some utility shell scripts.

To run end-to-end tests new temporary service containers are started with a different docker-compose project-name to ensure the containers don't collide with the default application. Additionally the test-services are configured to use different ports.

The tests are executed using the `testrunner` stage defined in the Dockerfile. This executes the tests inside a docker container, which is connected to the same network as the temporary service containers. This allows using the hostnames defined by docker-compose to refer to the services.

The testrunner outputs results in JUnit format inside the container, these results are then copied from the testcontainer to the runner and published as job-artifacts.

If any of the tests fail, the job fails and the pipeline stops execution.

Deploy-stage

After the tests have successfully been executed, the deploy stage is started. In the deploy stage first the possibly running applications are brought down using `docker-compose down`. After that they are brought up again using the images built in the build-stage.

Limitations of the pipeline

Currently the pipeline is configured to deploy the applications on the same host as the runner.

Learnings and challenges

Implementation of the project on top of the previously built project provided better understanding on how some things may or may not work simply due to different requirements in different systems.

For example in the original project, the files were written to a docker bind mount, meaning a host folder was mounted to the container. This, however was not well supported in the Gitlab CI pipeline, so this had to be changed into using a named volume.

Gitlab CI provided further challenges in trying to save the testresults from the testrunner Docker-container. Locally using a volume worked well, however inside the Gitlab CI runner volumes were not supported in a way that was easy to get working so I had to resort to copying the testresult files from the testrunner container.

One of the biggest things I learned from this project was the value of test-driven development (TDD). As the requirements specified that TDD had to be utilized in implementing the new features on top of the original project.

First this caused some challenges in creating an appropriate setup for executing the tests against the original application. After the first tests had been created adding more tests became easier.

Test-driven development was found to provide the following benefits:

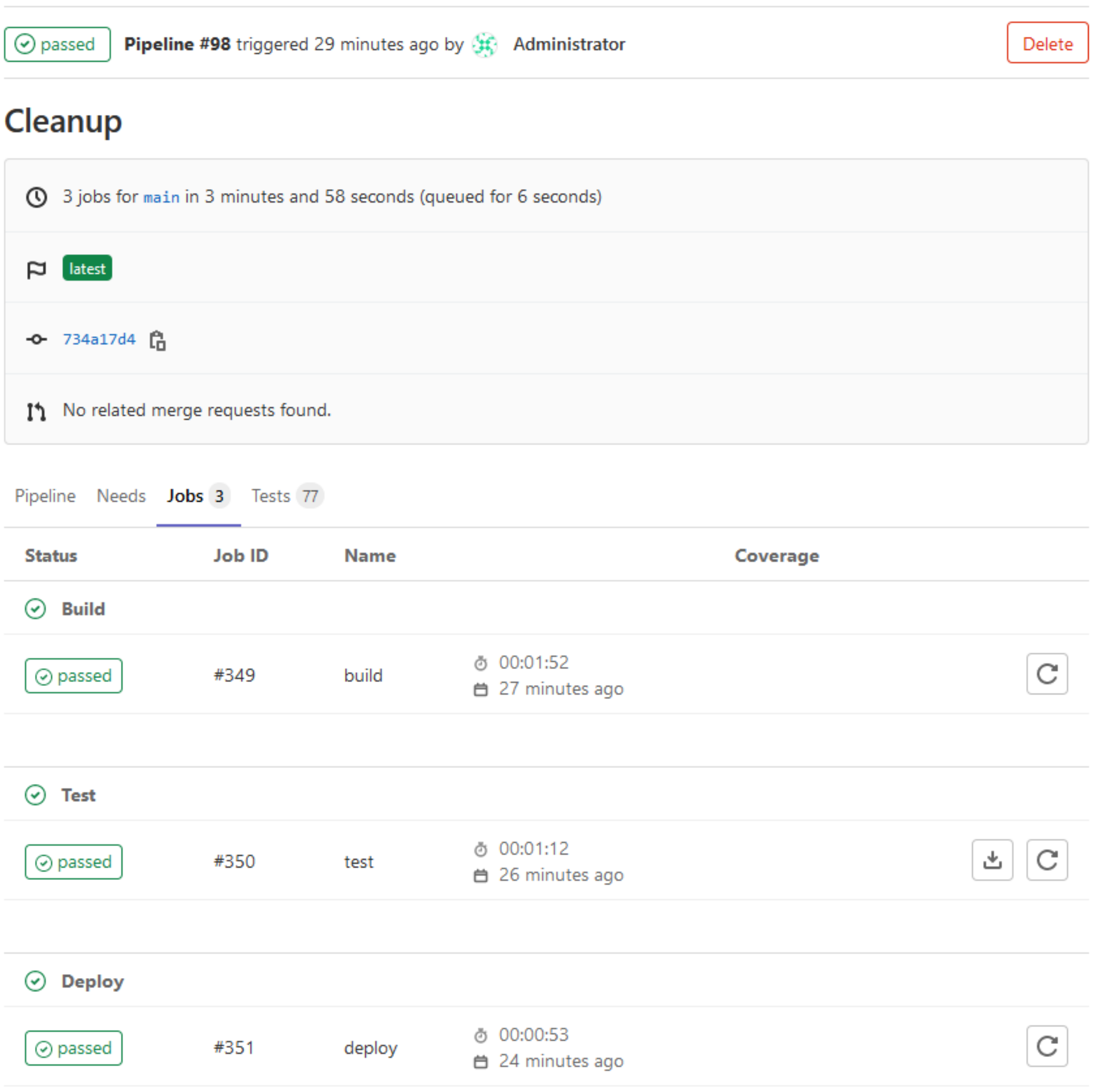
- Implementing new features by writing tests first required thinking about the minimum amount of effort to create the basic tests
    - Once the initial set of tests had been written adding further tests for each step of the process became much easier
- Refactoring became much easier since tests for the functionality already existed
    - For example performing large refactoring from using simple strings for messages logged by the Observer into using a stronger type (TopicMessage) turned out to be fairly simple and with tests the refactoring was done with great confidence

Test-driven development is not without drawbacks. For example development velocity in implementing new features was found to be slightly slower than normally. However, in a longer project this would not matter so much as having tests would be of great benefit when it comes to the maintenance of the application, so the overall development speed would likely be similiar, as you would have to write tests anyway at some point. This is also likely due to not being as familiar with test-driven development prior to this project. Overall test-driven development is something worth considering when starting a project with some sort of specification. It's a useful tool in a developers toolbox.

Documeting the code and scripts that were utilized is something I wish I had the time and energy to do. As in many projects, documentation is often the last thing to be done.

Examples from running the pipeline

Successful run - jobs view



passed **Pipeline #98** triggered 29 minutes ago by Administrator

Cleanup

3 jobs for `main` in 3 minutes and 58 seconds (queued for 6 seconds)

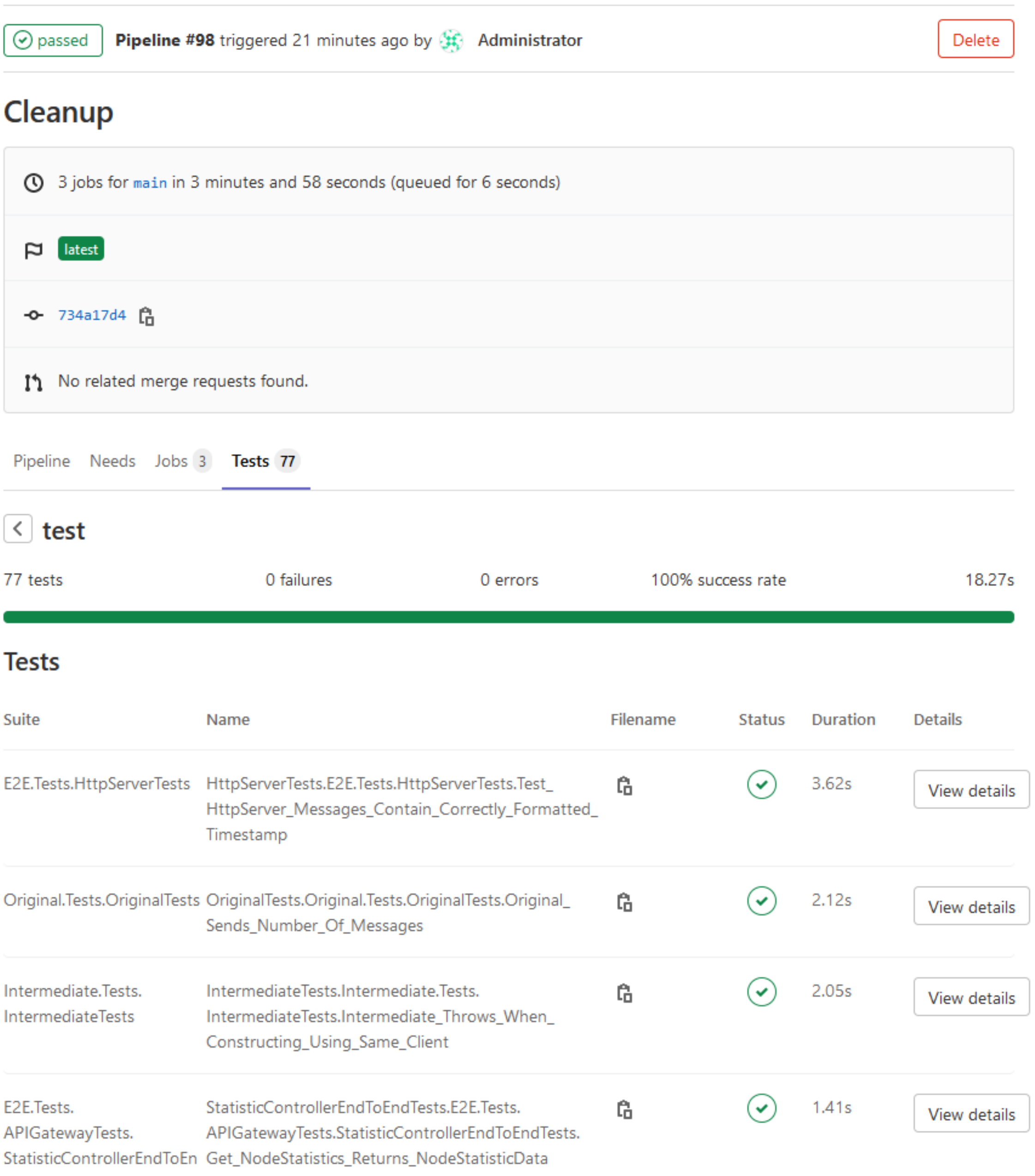
latest

734a17d4

No related merge requests found.

Status	Job ID	Name		Coverage
Build				
passed	#349	build	00:01:52 27 minutes ago	
Test				
passed	#350	test	00:01:12 26 minutes ago	
Deploy				
passed	#351	deploy	00:00:53 24 minutes ago	

Successful run - tests view


passed **Pipeline #98** triggered 21 minutes ago by Administrator

Delete

Cleanup

3 jobs for `main` in 3 minutes and 58 seconds (queued for 6 seconds)

latest

`734a17d4`

No related merge requests found.

Pipeline | Needs | Jobs 3 | **Tests 77**

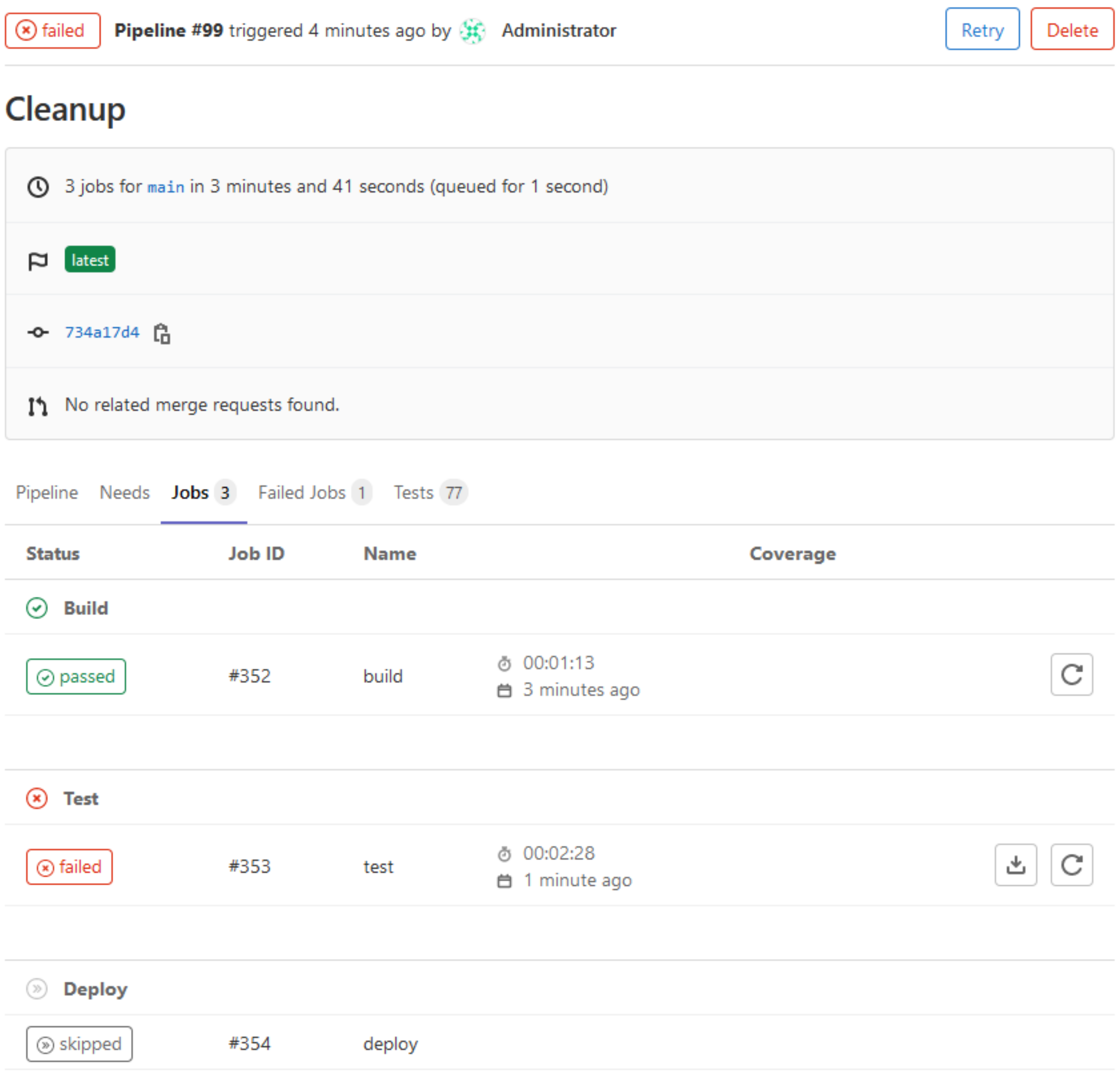
test

77 tests | 0 failures | 0 errors | 100% success rate | 18.27s

Tests

Suite	Name	Filename	Status	Duration	Details
E2E.Tests.HttpServerTests	HttpServerTests.E2E.Tests.HttpServerTests.Test_HttpServer_Messages_Contain_Correctly_Formatted_Timestamp		✓	3.62s	View details
Original.Tests.OriginalTests	OriginalTests.Original.Tests.OriginalTests.Original_Sends_Number_Of_Messages		✓	2.12s	View details
Intermediate.Tests.IntermediateTests	IntermediateTests.Intermediate.Tests.IntermediateTests.Intermediate_Throws_When_Constructing_Using_Same_Client		✓	2.05s	View details
E2E.Tests.APIGatewayTests.StatisticControllerEndToEn	StatisticControllerEndToEndTests.E2E.Tests.APIGatewayTests.StatisticControllerEndToEndTests.Get_NodeStatistics_Returns_NodeStatisticData		✓	1.41s	View details

Failed run - jobs view



failed **Pipeline #99** triggered 4 minutes ago by **Administrator**

Retry Delete

Cleanup

3 jobs for `main` in 3 minutes and 41 seconds (queued for 1 second)

latest

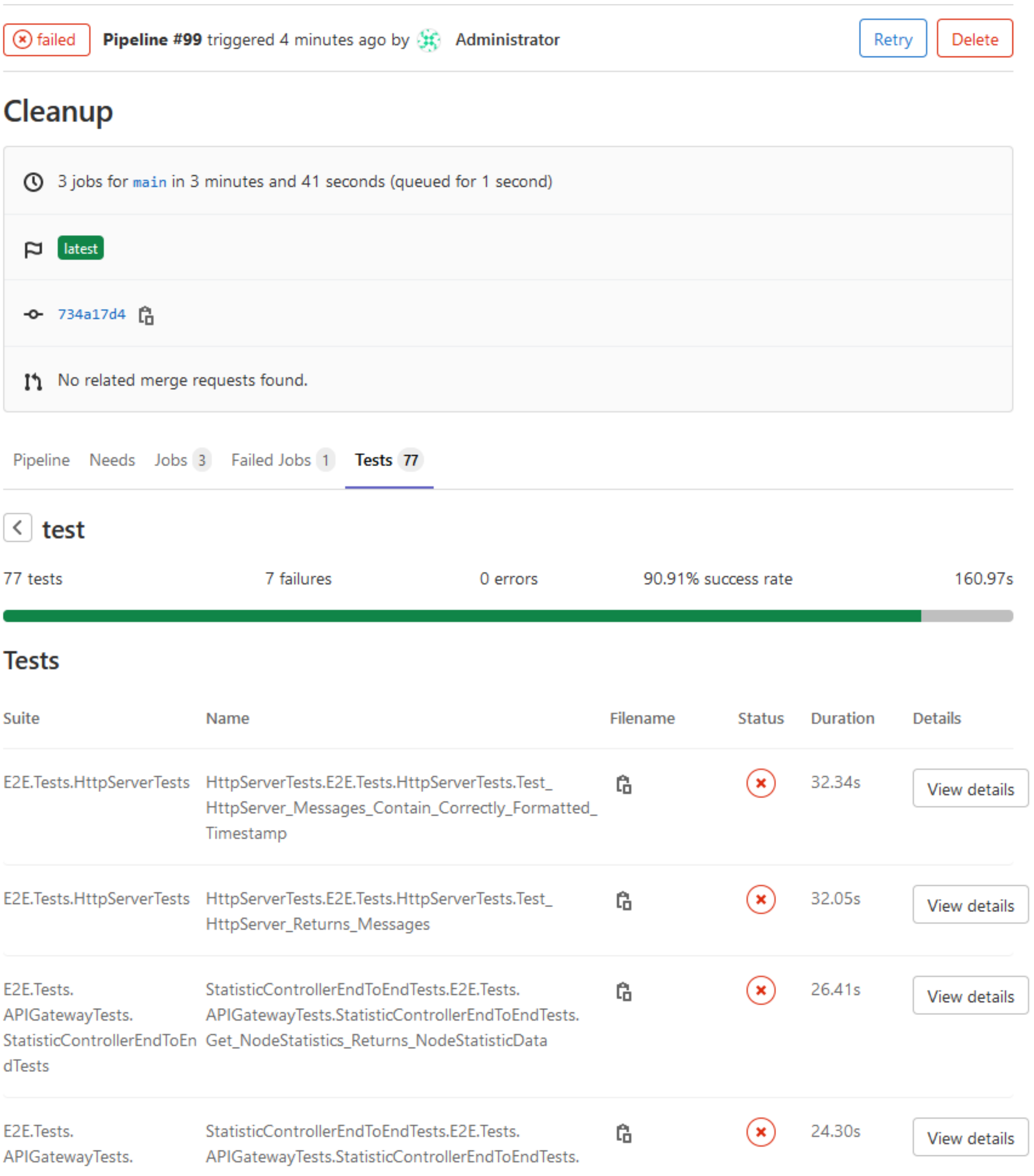
734a17d4

No related merge requests found.

Pipeline Needs **Jobs 3** Failed Jobs 1 Tests 77

Status	Job ID	Name		Coverage
Build				
passed	#352	build	00:01:13 3 minutes ago	
Test				
failed	#353	test	00:02:28 1 minute ago	
Deploy				
skipped	#354	deploy		

Failed run - tests view



failed **Pipeline #99** triggered 4 minutes ago by Administrator

Cleanup

3 jobs for `main` in 3 minutes and 41 seconds (queued for 1 second)

latest

`734a17d4`

No related merge requests found.

test

77 tests 7 failures 0 errors 90.91% success rate 160.97s

Tests

Suite	Name	Filename	Status	Duration	Details
E2E.Tests.HttpServerTests	HttpServerTests.E2E.Tests.HttpServerTests.Test_HttpServer_Messages_Contain_Correctly_Formatted_Timestamp		✗	32.34s	View details
E2E.Tests.HttpServerTests	HttpServerTests.E2E.Tests.HttpServerTests.Test_HttpServer_Returns_Messages		✗	32.05s	View details
E2E.Tests.APIGatewayTests.StatisticControllerEndToEndTests	StatisticControllerEndToEndTests.E2E.Tests.APIGatewayTests.StatisticControllerEndToEndTests.Get_NodeStatistics_Returns_NodeStatisticData		✗	26.41s	View details
E2E.Tests.APIGatewayTests.	StatisticControllerEndToEndTests.E2E.Tests.APIGatewayTests.StatisticControllerEndToEndTests.		✗	24.30s	View details

Appendix A: `.gitlab-ci.yml`

```yaml
# The image keyword is the name of the Docker image
# which the Docker executor runs to perform the CI/CD tasks.
# In this case we utilize an image with docker-compose installed
image: docker/compose:1.27.4

# Define common variables utilized in the CI/CD pipeline
variables:
  # This is utilized as a docker-compose --project-name
  # to allow running the containers even if the actual application
  # is running.
  E2E_PROJECT_NAME: e2e-test

  # The name of the docker-compose --project-name that will be deployed
  # in the 'deploy'-job.
  DEFAULT_PROJECT_NAME: ex-5-amqp

  # Docker image tag
  tag: "latest"

# Define the stages supported by the pipeline
# The stages contain groups of jobs run in each stage.
# Stages are run in the definition order and jobs inside a stage
# can be run in parallel
stages:
  - build
  - test
  - deploy

# Use docker-in-docker service to allow running of
# docker-commands from inside the Docker executor
services:
  - docker:dind

# This script is shared between the jobs and run before each job
before_script:
  - docker info
  - docker-compose version

# Define a build job
# The build job is set to run in the 'build'-stage
build:
  stage: build
  script:
    - ./DevOps/scripts/build-images.sh --tag "${tag}"

# Define a test job
# The test job is set to run in the 'test'-stage
test:
  stage: test
  # Define variables specific to the 'test'-job
  variables:
    # Override defaults
    # Since the application may be running
    # in the default ports
    RABBITMQ_PUBLIC_PORT: 5777
    RABBITMQ_MANAGEMENT_PUBLIC_PORT: 15677
    REDIS_PUBLIC_PORT: 16379
    HTTP_SERVER_PORT: 9512
    APIGATEWAY_PORT: 9513
    # Ensure we run end-to-end tests as well
    E2E: "true"
    ENV_FILE: ".env.e2e"

  script:
    - ./DevOps/scripts/create-test-env-file.sh ${ENV_FILE}
    - docker-compose --project-name $E2E_PROJECT_NAME down -v
    - docker-compose --project-name $E2E_PROJECT_NAME up -d --no-build
    - docker-compose --project-name $E2E_PROJECT_NAME ps
    - 'export EXTRA_RUN_ARGS="--network ${E2E_PROJECT_NAME}_backend"'
    - ./DevOps/scripts/run-docker-tests.sh ./Dockerfile . ${ENV_FILE} ./testresults

  # after_script can be utilized to define a set of commands
  # which are executed after the job, even if the job fails.
  # In this case the after_script is utilized to ensure the e2e-test
  # containers are properly stopped and removed even if the test run fails.
  # In addition the after_script also prints relevant logs to the console
  after_script:
    - docker-compose --project-name $E2E_PROJECT_NAME logs --tail="all" original
    - docker-compose --project-name $E2E_PROJECT_NAME logs --tail="all" observer
    - docker-compose --project-name $E2E_PROJECT_NAME logs --tail="all" httpserver
    - docker-compose --project-name $E2E_PROJECT_NAME logs --tail="all" apigateway
    - docker-compose --project-name $E2E_PROJECT_NAME down -v

  # artifacts define list of files & directories that are attached
  # to the job on success. In this case the artifacts are used to upload
  # the test results from the 'test'-job.
  artifacts:
    when: always
    paths:
      - ./testresults/**/*.xml
    reports:
      junit:
        - ./testresults/**/*.xml

# Define a deploy job
# The deploy job is set to run in the 'deploy'-stage
deploy:
  stage: deploy
  # The deploy job brings down the containers if they are running.
  # Then the containers are restarted using the previously built and tested
  # docker images.
  script:
    # Ensure we re-start the containers
    - docker-compose --project-name $DEFAULT_PROJECT_NAME down
    - docker-compose --project-name $DEFAULT_PROJECT_NAME up -d --no-build
    - docker-compose --project-name $DEFAULT_PROJECT_NAME ps
```