Sensor Recording Software (SRS)

This repository is used to build and deploy the `SRS` pipeline to create a sensor system. Preview Version

The Pipeline is composed of the following components:

- `Reverse Proxy`: the reverse proxy acts as the entry point of the backend and route the requests to the various components.
- `SensorRecordingSoftwareBackend` (https://gitlab.com/srs/srs-backend) (modules/srs-backend): service that exposes a REST API to receive data from sensors, websockets to control some of them (like the camera) and save the data in a `MongoDB`.
- `SensorRecordingSoftwareFrontend` (https://gitlab.com/srs/srs-frontend) (modules/srs-frontend): web application (nuxt) to monitor sensors and perform some actions on the pipeline.
- `MongoDB`: a Mongo database that stores the measurements (raw data).
- `DataExtractor`: a Python application triggered by the `SensorRecordingSoftwareBackend` to export raw data from `MongoDB` as `Apache Parquet` files.
- `TICK Stack`: a monitoring tool based on `InfluxData` TICK Stack.

The InfluxData TICK stack is composed of the following modules:

- **T**elegraf: agent for collecting and reporting metrics and events
- **I**nfluxDB: the time series database
- **C**hronograph: the interface to manage the entire InfluxData platform
- **K**apacitor: real-time streaming data processing engine, also used for alerting

Detailed information about each module of the TICK stack can be found here (https://www.influxdata.com/time-series-platform/).

Software Prerequisites

- `docker`: https://docs.docker.com/engine/install/ubuntu/ (https://docs.docker.com/engine/install/ubuntu/)
- `docker-compose`: https://docs.docker.com/compose/install/ (https://docs.docker.com/compose/install/)
- `k6` (required for load testing). Installation: https://k6.io/docs/getting-started/installation/ (https://k6.io/docs/getting-started/installation/)
- `python3` and `bcrypt` (required to inject users and their credentials in DB). Installation:

```
pip install bcrypt
```

Initial Setup

1. After having cloned the repo, initialize its submodules:

```
git submodule update --init --recursive
```

2. Create the `.env` file for development:

```
cp README.env.dev .env.dev
```

Adapt the values in the env file if needed.

3. Create the `.env` file for the `SensorRecordingSoftwareFrontend`:

```
cp modules/srs-frontend/.env.README modules/srs-frontend/.env
```

4. Create the JWT certificates in the `SensorRecordingSoftwareBackend` submodule. See chapter Certificates for JWT (https://gitlab.com/srs/srs-backend#certificates-for-jwt) of the sumodule's README.

5. Create the default config for the sensors for the `SensorRecordingSoftwareBackend`. See chapter Sensors config file (https://gitlab.com/srs/srs-backend#sensors-config-file) of the sumodule's README.

6. Install (https://k6.io/docs/getting-started/installation/) `K6` for load testing.

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys C5AD17C747E3415A3642D57D77C6C491D6AC1D69
echo "deb https://dl.k6.io/deb stable main" | sudo tee /etc/apt/sources.list.d/k6.list
sudo apt-get update
sudo apt-get install k6
```

Building and running the pipeline

Building and running the pipeling locally in DEV

You can make use of the `Makefile` (./Makefile) as follow:

```
make up
```

See the content of the `Makefile` for all the possible simplified commands.

Building and running the pipeline on the Test Machine

1. Clone the repo on the machine and follow the initial setup chapter above to create required certificates.
2. Run the stack on the machine.

```
ssh test-pipeline@192.168.67.29

# on the test machine
cd repos/srs-pipeline-platform

# build the stack
make build

# run the stack
make up

# list the services
make ps
```

3. Create the initial admin user that will be able to login into the `SensorRecordingSoftwareFrontend`:

```
# SSH the machine
# On the machine, run the following script which will ask you some information to create the initial admin user
./init_admin_user.sh
```

Building and running the pipeline on the Prod Machine

Initial server IP configuration

Some client devices (like the door sensors) hardcode the IP of the production pipeline server to which to sent the sensor data. On the server, `nginx` routes the incoming HTTP requests to the `SensorRecordingSoftwareBackend`.

The IP address of the production machine is reserverd in the IP list of the router. In case the IP address needs to be changed (replacement of the network card for example), you can update the new IP of the ubuntu server to the desired IP by updating the file `/etc/netplan/00-installer-config.yaml` as follow.

```
# File: /etc/netplan/00-installer-config.yaml
network:
  ethernets:
    enp45s0:
      dhcp4: no
      addresses:
        - 192.168.47.144/24
      gateway4: 192.168.47.1
      nameservers:
        addresses: [1.1.1.1]
  version: 2
  renderer: networkd
```

Once updated, apply the new plan using the following command:

```
sudo netplan apply
```

Deployment

Deployment of the pipeline on the production machine is configured through Gitlab CI. See chapter Deployment using Gitlab-CI below.

Accessing the applications once the services are up

In DEV, the services run on your localhost and you can directly access them through your browser, e.g. `http://localhost:3000` for the frontend.

For the Test/Prod setup, you need to tunnel the needed ports through SSH first. Adapt the user and IP of the server for the Test or Prod setup.

```
# Example for the production machine
ssh -L 3000:localhost:3000 -L 4000:localhost:4000 -L 8888:localhost:8888 -L 27017:localhost:27017 pipeline@192.168.47.144
```

Then access the applications as follow:

- The `SensorRecordingSoftwareFrontend` is ava http://localhost:3000
- `Chronograf` (monitoring): http://localhost:8888
- `SensorRecordingSoftware` API documentation: http://localhost:4000
- Access MongoDB using MongoPass and connect to: http://localhost:27017

Load testing

We use K6 (https://k6.io/docs/) for load testing. https://k6.io/docs/getting-started/installation/ (https://k6.io/docs/getting-started/installation/)

The load test consists of 50 Virtual Users (VUs) performing a request (sending data) for each sensor type every 500ms. The test runs for a duration of 60s.

```
cd ./performance_testing
./run_test.sh
```

Below is the result of the load test on the test setup.

K6 will also output the details of the load test in `InfluxDB`. The result of the load test can be visualized using `Chronograf`. A custom frontend for `Chronograf` is avaiable in `./performance_testing/chronograph-dashboard` and will display the following charts.

How to

Configure an experiment and record data

Once the `srs Pipeline` is running, use the `SensorRecordingSoftwareFrontend` to configure and experiment and record data.

This process consists of:

1. Create a new config for the sensors to use:

   This will create a new default config from the file specified with ENV variable: `PIPELINE_SENSORS_FILEPATH` (see file `.env`)

2. Load the sensors config and edit the sensors you want to use
3. Create a new experiment, assign the config to be used and the owner of the experiment:
4. Configure the experiment:

   Here you can define the study protocol:

   - tasks
   - operators
   - participants
5. Set the experiment as the active experiment (only one can be active at a time):

   Note that the database must be empty before switching the active experiment! We don't want to mix data from different experiment. If the DB is not empty, export the data and backup it (if needed), then drop the content of the DB by visiting the "data management tab".
6. Visualize the sensors activity:
7. Start recording:

Important notes:

- A recording can only be started if the DB is empty.
- You can delete the DB from the `data management`'s page (admin only).

Exporting data

The data of the active experiment must be exported before starting another experiment. In other words, a new experiment recording can only be started with an empty database (samples collections).

You have 2 options to export data:

- (Preferable way) Use the `SensorRecordingSoftwareFrontend` when the amount of data recorded and to export is reasonable (< 50 GB per export),
- Use the export script when the amount of data is bigger, note that this requires an access via SSH to the production machine.

Exporting data using the SensorRecordingSoftwareFrontend

In order to export data, navigate to `Admin -> Experiments -> Data management` and click the `Export data active experiment` button. You will be asked to select the date for which to export the data. It is responsibility of the experiment owner to organize and backup the data exported via the frontend.

The exported experiment data is available in folder `./data/exports/[EXPERIMENT_UUID]/`. (The `SensorRecordingSoftwareBackend` exports the data in the `./exports` folder which is mounted in the `./data` folder on the host, alongside the other persistent data.)

The structure of the exported data is as follow:

- [EXPERIMENT_UUID]/:
  - experiment_[current_datetime].json: contains the details of the experiment at the time the export is executed (current_datetime). This includes the sensors used, participants, owner, operators, tasks, ...
  - experiment_log_[current_datetime].json: contains the part of the logbook associated to the experiment being exported at the time the export is executed (current_datetime).
  - progress_participant_[current_datetime].json: contains the progress of the participants for the experiment at the time the export is executed (current_datetime).
  - [PARTICIPANT_UUID]:
    - [date_selected_for_export]_[collection_type]_samples.dump: mongo dump of the samples collections for the date being selected in export.

Exporting data using the export script

When there is a lot of data to export, the export feature via the `SensorRecordingSoftwareFrontend` can fail due to requests that timeout to preconfigure the export job. In such case, the script utils/export_for_participant.sh (./utils/export_for_participant.sh) can be used as an alternative.

Important note: This script is not complete and only exports the samples collections as single dumps using `mongodump` tool. The script doesn't export the configuration of the experiment, neither the logbook or other metadata related to the experiment. However, it is robust and will work with the database containing hundreds of GB of data.

In order to use this script, copy it first to the production machine:

```
# Copy to the production machine
scp export_for_participant.sh deployer@192.168.47.144:.
```

Then run it by specifying the participant uuid:

```
./export_for_participant.sh e33f6d1b-43a7-4a4d-a83e-137db79edd11
```

The data will be exported in the folder `./output`. Note that a single dump is created for each collection, as illustrated below.

Then copy the dumps to the `RainbowDash` NAS and don't forget to document the experiment configuration and metadata manually.

Importing some dumps

See folder import_from_dumps (./utils/import_from_dumps/) for an example that show how to restore dumps created during export in a mongo DB.

This script loops through all the *.dump files in a directory and restores them in MongoDB.

In order to test the script:

1. Launch a mongo db instance if you don't have one yet. You can use the file `./utils/import_from_dumps/docker-compose.yml`.

```
cd ./utils/import_from_dumps/

cp example.env .env

docker-compose up -d mongo-data-analysis
```

2. Run the import script:

```
./import_samples_from_dumps.sh ../../data/exports/[EXPERIMENT_UUID]/[PARTICIPANT_UUID]/
```

Monitoring

We use the TICK stack from `InfluxData` to monitor the production machine and some processes of the Pipeline.

The name TICK comes from the intial of those components:

- **T**elegraf: agent for collecting and reporting metrics and events
- **I**nfluxDB: the time series database
- **C**hronograph: the interface to manage the entire InfluxData platform
- **K**apacitor: real-time streaming data processing engine, also used for alerting

Chronograf authentication

Authentication and authorization to access the dashboard is done using OAuth 2.0 via Gitlab authentication (https://docs.influxdata.com/chronograf/v1.9/administration/managing-security/#configure-gitlab-authentication).

In order to enable OAuth via Gitlab, we must create a Group Application (https://gitlab.com/groups/srs/-/settings/applications). We have actually created two for the group `srs`, one for dev and one for production:

- `Monitoring Chronograf Production`
- `Monitoring Chronograf Development`

This will allow Gitlab to perform the authentication and redirect to the correct url to access the `Chronograf` application. The correct application credentials are injected via environment variables.

TICK Stack configuration

The configuration of the TICK stack is done through configuration files. See the following files:

- influxdb.conf (./srs/tick-stack/influxdb/config/influxdb.conf) for `InfluxDB`
- telegraf.conf (./srs/tick-stack/telegraf/telegraf.conf) for `Telegraf`
- kapacitor.conf (./srs/tick-stack/kapacitor/config/kapacitor.conf) for `Kapacitor`
- influxdbconnection.src (./srs/tick-stack/chronograf/resources/influxdbconnection.src) and kapacitorconnection.kap (./srs/tick-stack/chronograf/resources/kapacitorconnection.kap) for `Chronograf`

Those files are copied during build by the respectice Dockerfiles.

The configuration files use environment variables which are injected by the docker-compose.yml (./docker-compose.yml) file. See variables provided to the services `srs-influxdb`, `srs-telegraf`, `srs-chronograf`, `srs-kapacitor`.

Kapacitor alerts and notifications

`Kapacitor` allows to analyze the time series collected by `Telegraf` and saved in `InfluxDB`. `TICKScripts` are scripts that you can develop to analyze this data and send alerts if some metrics goes over some thresholds. See the TICKScript language reference (https://docs.influxdata.com/kapacitor/v1.5/tick/) for more details.

Currently, we have one `tickscript` to monitor the disk size.

You can add new tick scripts through the interface but this would be lost when you remove/rebuild the container unless, you mounted the appropriate volumes. We recommand to put your scripts in folder `kapacitor/config` once ready and copy them using the Dockerfile so that they will be available directly when rebuilding the container.

`Kapacitor` is configured to send alerts via SMTP and using SLACK channel (see kapacitor.conf (./srs/tick-stack/kapacitor/config/kapacitor.conf)). You can test your notification channels via the `Chronograf` interface via Configuration -> My Kapacitor Connection -> Edit button.

Here you can send test alerts to check that the credentials for the various channels are properly configured.

When pressing "Send test alert" for the SLACK channel, you should receive the following test message:

Seeing outputs (logs) of the Sensors srs backend or Nginx

You can check that everything is running fine on the machine and service also by visualizing the output of the `srs-backend` or the logs of nginx.

Example:

```
# ssh the machine

# attach the log of srs-backend
docker-compose --env-file .env.prod -f docker-compose.yml -f docker-compose.prod.yml logs -f --tail=100 srs-backend
```

```
# ssh the machine

# check the nginx logs
tail -f data/logs/nginx/nginx_access.log

# or the error log
tail -f data/logs/nginx/nginx_error.log
```

Deployment using Gitlab CI/CD

We make use of `Gitlab CI/CD` (https://docs.gitlab.com/ee/ci/README.html) for Continuous Integration (CI) and Continuous Deployment/Delivery (CD).

Process

The process consists of two stages:

- Build stage:
  - During build stage the various docker images required by the project will be built and pushed to our gitlab container registry
- Deployment stage:
  - For the deployment, we have setup a user called `deployer` on the machine. The stack will be deployed in the home folder of this user, i.e. `/home/deployer/`.
    During the deployment stage, we will perform the following actions on the production machine:
    - create the required files: environement file (.env.prod), certificates (JWT, SSL, ...), docker-compose file, ...
    - pull the required images from our gitlab container registry
    - stop and re-run the stack

Note on deployment strategy

Why do we use SSH to run the deploy commands (see gitlab-ci.yaml) on our own server? From How To Set Up a Continuous Deployment Pipeline with GitLab CI/CD on Ubuntu 18.04 (https://www.digitalocean.com/community/tutorials/how-to-set-up-a-continuous-deployment-pipeline-with-gitlab-ci-cd-on-ubuntu-18-04):

> *It may seem odd to use SSH to run these commands on your server, considering the GitLab runner that executes the commands is the exact same server. Yet it is required, because the runner executes the commands in a Docker container, thus you would deploy inside the container instead of the server if you'd execute the commands without the use of SSH. One could argue that instead of using Docker as a runner executor, you could use the shell executor to run the commands on the host itself. But, that would create a constraint to your pipeline, namely that the runner has to be the same server as the one you want to deploy to. This is not a sustainable and extensible solution because one day you may want to migrate the application to a different server or use a different runner server. In any case it makes sense to use SSH to execute the deployment commands, may it be for technical or migration-related reasons.*

CI/CD setup

The CI/CD setup consists of:

- creating the user for deployment
- registering a gitlab runner to execute the jobs for the two stages (build and deploy)
- configure the jobs in the gitlab-ci.yaml file

Creating the user for deployment

Follow the process below to create the user on the server that will be used to deploy the pipeline:

```
# ssh the machine
sudo adduser deployer
```

Add the user to the Docker group:

```
sudo usermod -aG docker deployer
```

This permits deployer to execute the docker command, which is required to perform the deployment.

Then generate a ssh key pair (https://docs.gitlab.com/ee/ssh/index.html#generate-an-ssh-key-pair) and add the public key to the authorized key:

```
su deployer

# Generate ssh key pair
ssh-keygen -t rsa -b 2048

# Authorize the SSH key for the deployer user
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Copy the content of the id_rsa (private key) as a CI/CD variable (see variable called ID_RSA). See the reference, how To Set Up a Continuous Deployment Pipeline with GitLab CI/CD on Ubuntu 18.04 (https://www.digitalocean.com/community/tutorials/how-to-set-up-a-continuous-deployment-pipeline-with-gitlab-ci-cd-on-ubuntu-18-04) for more details.

Register a gitlab runner

The next step is to register a specific runner that will execute our jobs. We've opted to have a the `gitlab-runner` running as a docker container (https://docs.gitlab.com/runner/install/docker.html). We've decided to register the runner on the Production machine directly, but the runner could be register virtually anywhere, as it will deploy the pipeline by connecting to the production machine over SSH. For example, it is possible to register additional runners on your personnal machine.

On the production machine, we will run and register the gitlab runner with the `pipeline` user which has sudo rights so that we can edit the config of the runner after registration. Follow the steps below:

1. Create and run a gitlab-runner service:

```
# Copy (if not there yet) the docker-compose file that we will use to run and register a gitlab runner
# A folder "gitlab_runner" already exists on the home of the pipeline user. If not create the folder before.
scp docker-compose.gitlab-runner.yml pipeline@192.168.47.144:~/gitlab_runner/.

# SSH the production machine
ssh pipeline@192.168.47.144

# Go to the gitlab_runner folder
cd gitlab_runner

# Creates the mp-gitlab-runner service
docker-compose -f docker-compose.gitlab-runner.yml up -d mp-gitlab-runner

# You can check that the service is running with the following command
docker-compose -f docker-compose.gitlab-runner.yml ps
```

2. Register the runner

```
# SSH the machine
ssh deployer@192.168.47.144

cd ~/gitlab_runner

# Register the runner. When prompted, answer with the questions below
docker-compose -f docker-compose.gitlab-runner.yml exec -T mp-gitlab-runner gitlab-runner register
```

When registering the runner, answer with the following questions:

- Gitlab instance: "https://gitlab.com/"
- Enter the registration token: [REGISTRATION_TOKEN (https://gitlab.com/srs/srs-backend/-/settings/ci_cd)]
- Enter a description for the runner: "SRSBackend Machine Runner Deployer"
- Enter tags for teh runner (comma-separated): "build,deployment_prod"
- Enter an executor: docker
- Enter the default Docker image: "docker:stable"

3. Update the config of the runner:

```
# SSH the machine
ssh deployer@192.168.47.144

cd ~/gitlab_runner

# Stop the runner
docker-compose -f docker-compose.gitlab-runner.yml stop mp-gitlab-runner

# Edit the gitlab runner config file (config.toml)
sudo vi config/gitlab-runner/config.toml
```

Replace the line `volumes = ["/cache"]` with `volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock"]` to fix this issue (https://serverfault.com/a/1074274))

After the edit, the `config.toml` file should look like this:

```
concurrent = 1
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "SRS Production Machine Runner Deployer"
  url = "https://gitlab.com/"
  token = "u_wG5KsHxQUiz5KzHKnH"
  executor = "docker"
  [runners.custom_build_dir]
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    image = "docker:stable"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock"]
    shm_size = 0
```

4. Restart the runner

```
# Still on the prod machine

# Re-run the runner
docker-compose -f docker-compose.gitlab-runner.yml up -d mp-gitlab-runner
```

Upon successfull registration, the gitlab runner will display in the CI/CD settings, under specific runner (https://gitlab.com/srs/srs-backend/-/settings/ci_cd):

The runner is now ready to execute CI/CD jobs.

Configure the Gitlab CI/CD variables

Gitlab-CI will create the `.env.prod` that contains the environment variables for production and deploy it on the production machine, before running the docker-compose commands: `docker-compose --env-file .env.prod -f docker-compose.yml -f docker-compose.prod.yml ps`.

The creation and deployment of the env file is done in the `deploy-pipeline-production` job of the `deploy` stage. See the file gitlab-ci.yml (.gitlab-ci.yml) for configuration. This job will create the env file from the CI/CD variables (https://gitlab.com/srs/srs-backend/-/settings/ci_cd) that are configure in Gitlab-CI.

The following variables are expected:

- `GITLAB_OAUTH_GENERIC_CLIENT_ID`: the ID of the gitlab group application used for gitlab authentication (in `chronograf`),
- `GITLAB_OAUTH_GENERIC_CLIENT_SECRET`: the secret of the gitlab group application used for gitlab authentication (in `chronograf`),
- `GITLAB_OAUTH_TOKEN_SECRET`: a token secret to define between `chronograf` and Gitlab,
- `ID_RSA`: the private ssh key which is used to ssh and deploy the solution on the production machine (the one used by the deployer user),
- `SERVER_IP`: the IP of the server on which to deploy the containers (production machine),
- `SERVER_USER`: the name of the user which is used to deploy the solution on the production machine, i.e. `deployer`
- `INFLUXDB_ADMIN_PASSWORD`: the password for the admin user of the influxdb,
- `INFLUXDB_READ_USER_PASSWORD`: the password for the user with read only access to the influxdb,
- `INFLUXDB_WRITE_USER_PASSWORD`: the password for the user with write access to the influxdb,
- `KAP_SMTP_FROM`: the email address used to send alerts by `kapacitor`,
- `KAP_SMTP_HOST`: the SMTP host used to send alerts by `kapacitor` (`cicero.metanet.ch`),
- `KAP_SMTP_PASSWORD`: the password of the email account used to send alerts by `kapacitor`,
- `KAP_SMTP_PORT`: the port used for SMTP service,
- `KAP_SMTP_TO`: the email recipient to which are sent the alerts (forwarding to specific persons can be configured in `cicero.metanet.ch`)
- `KAP_SMTP_USERNAME`: the username of the SMTP user,
- `MONGODB_USER`: the root user of the mongodb database,
- `MONGODB_PASSWORD`: the password of root user of the mongodb database,
- `MONGODB_DB_NAME`: the name of the mongodb database,

- `NGINX_SSL_CERT_FULLCHAIN`: the cert fullchain of the SSL wildard certificate,
- `NGINX_SSL_CERT_PRIVKEY`: the private key of the SSL wildcard certificate,
- `PIPELINE_SENSORS_FILEPATH`: relative path from the `srs-backend` to a `*.sensors.json` file
- `SRS_BACKEND_API_URL`: URL of the srs backend service API
- `SRS_BACKEND_AUTHENTICATED_WEBSOCKETS_URL`: URL to open the authenticated websockets (sensors and cameras websockets)
- `SRS_BACKEND_OPEN_WEBSOCKETS_URL`: URL to open the non-authenticated websockets (active experiments status websocket)
- `SLACK_CHANNEL`: slack channel to which monitoring alerts can be sent by `kapacitor`,
- `SLACK_URL`: the SLACK url used for sending alerts by `kapacitor`,
- `SLACK_WORKSPACE`: the SLACK workspace used for sending alerts by `kapacitor`,
- `SMS_JWT_PRIVATE_KEY`: private key for JWT management,
- `SMS_JWT_PRIVATE_KEY_FILEPATH`: path to the private key for the JWT management,
- `SMS_JWT_PUBLIC_KEY`: public key for JWT management,
- `SMS_JWT_PUBLIC_KEY_FILEPATH`: path to the public key for the JWT management,
- `SMS_JWT_VALIDITY_LENGTH`: validity period for the JWT,
- `SMS_SMTP_HOST`: the host used to send mails for the `srs-backend` (bookings),
- `SMS_SMTP_PASSWORD`: the password of the account used to send mails for the `srs-backend` (bookings),
- `SMS_SMTP_USER`: the user used to send mails for the `srs-backend` (bookings),
- `SMS_SMTP_FROM`: the from address used to send mails for the `srs-backend` (bookings),

Using Gitlab-CI pipelines for the deployment

Once Gitlab-CI is configured properly, when you push your commits to the repository, the commits will appear in the CI/CD pipelines.

The jobs are configured to be triggered manually.

Click on the `skipped` button to display the stages and jobs of the pipeline. You can then use GUI of Gitlab-CI to trigger the jobs.

The build stage will build the docker images and push them to the container registry.

The deploy stage will:

- create the required files in the home of the deployer user `/home/deployer`, i.e.:
  - the `.env.prod` file,
  - the `docker-compose.yml` and `docker-compose.prod.yml`,
  - the required certificates for JWT management,
- replace the version of the the docker images in the image url with the `CI_COMMIT_SHORT_SHA`,
- pull the docker images tagged with the `CI_COMMIT_SHORT_SHA` from the container registry
- and run the docker-compose stack.

See the content of the `gitlab-ci.yml` file for the jobs definition.

Rollback to a previous version using Gitlab-CI

Once the deployment job is completed, if you scroll to the top of the page, you will find the `This job is deployed to production` message. GitLab recognizes that a deployment took place because of the job's environment section. Click the production link to move over to the production environment.

You can also access this page through `Deployments` -> `Environments` which tracks the deployments for the various environments configured.

For each deployment there is a re-deploy button available to the very right. A re-deployment will repeat the deploy job of that particular pipeline.

Whether a re-deployment works as intended depends on the pipeline configuration, because it will not do more than repeating the deploy job under the same circumstances. Since you have configured to deploy a Docker image using the commit SHA as a tag (see the docker-compose.prod.yml (./docker-compose.prod.yml) file), a re-deployment will work for your pipeline.

> *Note: Your GitLab container registry may have an expiration policy. The expiration policy regularly removes older images and tags from the container registry. As a consequence, a deployment that is older than the expiration policy would fail to re-deploy, because the Docker image for this commit will have been removed from the registry. You can manage the expiration policy in Settings > CI/CD > Container Registry tag expiration policy. The expiration interval is usually set to something high, like 90 days. But when you run into the case of trying to deploy an image that has been removed from the registry due to the expiration policy, you can solve the problem by re-running the publish job of that particular pipeline as well, which will re-create and push the image for the given commit to registry.*

Source: Article on Digital Ocean: How To Set Up a Continuous Deployment Pipeline with GitLab CI/CD on Ubuntu 18.04 (https://www.digitalocean.com/community/tutorials/how-to-set-up-a-continuous-deployment-pipeline-with-gitlab-ci-cd-on-ubuntu-18-04#step-7-validating-the-deployment)

Cleanup on the production machine

From time to time, you can delete the obsolete and dangling images on the production machine to free up space like this:

```
# SSH the machine
ssh deployer@192.168.47.144

# on the machine

# List the docker images
docker images

# Remove the dangling images
docker image prune
```

Backup and Restore

Strategy

A daily backup of some important collections (users, bookings, experiments, configs, ...) (see `./utils/daily_backup.sh` for the full list) runs daily on the production machine.
Note that the daily backup script **does not backup any samples collected**, this is from the responsibility of the owner of the experiment.

The full backup process consists of two steps:

1. An archive is created by the `./utils/daily_backup.sh` script, which is run as a cronjob by the `deployer` user (see below) at 02:15am.
2. The created archive is then copied (rsync) to the `Rainbow_Dash NAS` every day at 05:15am by a backup job configured on the NAS, in shared folder: `/DailyBackupProductionMachine/home/deployer/backups`

Daily backup script

The `./utils/daily_backup.sh` script is aimed to daily backup some collections that have to persists like the users definitions (for users management), experiments definitions, log book, ...

The script is copied on the home folder of the `deployer` user and the following entry can be installed in the crontab to run the backup script daily. This has to be done manually once.

```
# Copy the daily backup script in the home of the deployer user
scp utils/daily_backup.sh deployer@192.168.47.144:.

# Login with the deployer user on the machine on which to perform the daily backup
ssh deployer@192.168.47.144

# Edit the crontab
crontab -e

# In the crontab editor, enter the following line (will run the script every day at 3:15am)
15 3 * * * cd /home/deployer/ && ./daily_backup.sh >> ./daily_backup.log 2>&1
```

The daily backup script will create a compressed archive (bz2) of the exported mongo collections in folder `./backups/YYYY/MM` (e.g. `./backups/2022/02/2022-02-07_backup.tar.gz`).

Backup job set up on Synology NAS (Rainbow_Dash)

Pre-requisite: add a dedicated user for the backup on the production machine

1. Create a dedicated user `remotebkp` on the production machine that we will use to connect and rsync some folders (backup job on the Synology NAS)

```
# SSH the production machine with a sudo user
ssh pipeline@192.168.47.144

# Create new user account for the user that will connect for the backup
sudo adduser remotebkp
```

2. Generate a SSH key pair and add the public key `remotebkp_id_rsa.pub` to the authorized keys for the `remotebkp` user on the production machine. This will allow the Synology NAS to ssh and run some rsync commands.

```
# SSH the production machine with the remotebkp user
ssh remotebkp@194.182.165.107

# Generate a new key pair and name it remotebkp_id_rsa
ssh-keygen -t rsa -b 2048

# Add remotebkp_id_rsa.pub to /home/remotebkp/.ssh/authorized_keys
cat ~/.ssh/remotebkp_id_rsa.pub >> ~/.ssh/authorized_keys
```

Configure the backup job on the NAS

Add the backup job on Synology NAS as follows:

1. From the DSM start menu, choose **Active Backup for Business**
2. In **File Server**, click on **Add Server**
3. Remote server information:
   - Server address: `192.168.47.144` (IP production machine)
   - Connection mode: **rsync shell mode via SSH**
   - Port: `22`
   - Account: `remotebkp`
   - Authentication method: **By SSH key**
   - SSH key: Upload *remotebkp_id_rsa*
4. Create a new task for the server
   - Backup type: **Incremental**
   - Select the files to back up, i.e. */home/deployer/backups*
5. Task settings:
   - Task name: `Copy daily backup to Rainbow_Dash NAS`
   - Local path: Browse to `/DailyBackupProductionMachine` shared folder (only accessible by admins)
   - Enable schedule: Enabled
   - Run on the following days: **Daily**
   - First run time: **05:15** (anything sufficiently after the cron job created on the machine)
6. Review the summary and save the task

Restoration from a daily backup archive

The `./utils/restore_from_backup.sh` script allows to restore a backup that has been created by the `daily_backup.sh` script. Please make sure the mongo database is empty before running the restore script.

Important note: if the databases names are different between the backup and the environment in which you want to restore, edit the `nsFrom` and `nsTo` parameters of the `mongorestore` command in the `restore_from_backup.sh` script. For example, if the mongo dumps created by the `daily_backup.sh` script have been exported from the production db (`pipeline_prod`) and that you are trying to restore the dump in your local environment (`pipeline_dev` by default), adapt the command in the script as follow:

```
mongorestore --uri="mongodb://$MONGODB_USER:$MONGODB_PASSWORD@localhost:27017/?authSource=admin" --archive="$dump_file" --nsFrom="pi
```

```
# Extract the archive
tar -xf 2022-02-07_backup.tar.bz2

# Run the restore script
cd ./utils
./restore_from_backup.sh path/to/extracted/2022-02-07_backup
```

References

CI/CD:

- Deploying applications to the Docker/NGINX server using GitLab CI (https://olex.biz/2020/01/deploying-to-docker-nginx-via-gitlab-ci/)
- How To Set Up a Continuous Deployment Pipeline with GitLab CI/CD on Ubuntu 18.04 (https://www.digitalocean.com/community/tutorials/how-to-set-up-a-continuous-deployment-pipeline-with-gitlab-ci-cd-on-ubuntu-18-04)
- How to fix Gitlab CI error during connect: Post http://docker:2375/v1.40/auth: dial tcp: lookup docker on … no such host (https://techoverflow.net/2021/01/12/how-to-fix-gitlab-ci-error-during-connect-post-http-docker2375-v1-40-auth-dial-tcp-lookup-docker-on-no-such-host/)

Important notes

- In order to make submodules work correctly in CI/CD jobs, it is needed to use relative URLs for submodules located in the same Gitlab server. See documentation here (https://docs.gitlab.com/ce/ci/git_submodules.html#configuring-the-gitmodules-file).

Coding guidelines

- JSON fields: follow snake case

Useful commands (cheatsheet)

Docker

Here below are some usefull commands you can inspire while working with `Docker` and `docker-compose`. Those are example commands you can run on the production environment.

```
# List running containers
docker-compose --env-file .env.prod -f docker-compose.yml -f docker-compose.prod.yml ps

# Display log output from a specific service (e.g. srs-backend)
docker-compose --env-file .env.prod -f docker-compose.yml -f docker-compose.prod.yml logs -f --tail=100 srs-backend

# Entering a container (e.g. srs-backend)
docker-compose --env-file .env.prod -f docker-compose.yml -f docker-compose.prod.yml exec srs-backend /bin/sh

# Restarting a service (e.g. srs-backend)
docker-compose --env-file .env.prod -f docker-compose.yml -f docker-compose.prod.yml restart srs-backend
```

Working with MongoDB container

Enter container

```
# Enter MongoDB container
docker-compose --env-file .env.dev -f docker-compose.yml -f docker-compose.dev.yml exec mongo /bin/bash

# From inside containenr

# Open MongoDB connection
mongo -u "$MONGO_INITDB_ROOT_USERNAME" -p "$MONGO_INITDB_ROOT_PASSWORD"

# Usefull commands
show dbs
show collections
# List users
db.users.find()
# List configs
db.experimentconfigs.find()
```

Export dump manually:

```
docker-compose --env-file .env.dev -f docker-compose.yml -f docker-compose.dev.yml exec -T mongo sh -c 'mongodump --username=${MONGO_
```

Restore dump

```
docker-compose --env-file .env.dev -f docker-compose.yml -f docker-compose.dev.yml exec -T mongo sh -c 'mongorestore --username=${MO
```