FE Dockerfile description

This Dockerfile builds a Docker image for a Quasar project. It has two stages, build and prod.

Stage 1: Build

1. The base image used for this stage is node:lts.
2. The API_URL and API_PUBLIC_PATH build arguments are declared. The default value of API_URL is http://team12-22.studenti.fiit.stuba.sk/api and the default value of API_PUBLIC_PATH is /turing/.
3. The working directory is set to /build.
4. The package.json and package-lock.json files are copied to the working directory.
5. The command npm install is run to install the project's dependencies.
6. The rest of the project files are copied to the working directory.
7. The command npx @quasar/cli build is run to build the Quasar project.

Stage 2: Production

1. The base image used for this stage is node:lts.
2. The environment variables HOST and PORT are declared. The default value of HOST is 0.0.0.0 and the default value of PORT is 8080.
3. The working directory is set to /turing.
4. The /build/dist/spa directory from the first stage is copied to the working directory.
5. The 8080 port is exposed.
6. The @quasar/cli package is installed globally using the npm install -g @quasar/cli command.
7. The default command for the image is ["quasar", "serve", "--history"], which starts a Quasar development server.

BE Dockerfile description

There are two Dockerfile files for the BE project. Dockerfile – for dev/test/prod environment and Dockerfile.local for local environment.

This Dockerfile builds a Docker image for a Python project.

1. The base image used for this image is python:3.10.
2. The environment variable PYTHONUNBUFFERED is set to 1.
3. The environment variable secrets_path is set to .secrets.
4. The ssh_key build argument is declared.
5. The working directory is set to /app.
6. The current directory's files are copied to the working directory.
7. The pip package manager is upgraded to the latest version with the pip install --upgrade pip command.
8. The /app/logs/ directory is created with the mkdir -p /app/logs/ command.
9. The requirements.txt file is copied to the /app/ directory.
10. The project's dependencies are installed with the pip install -r requirements.txt command.
11. The src directory is copied to the /app/ directory.
12. The default command for the image is python src/app.py, which runs the main Python script app.py in the src directory.

Usage of docker-compose

During deployment we are using **docker-compose** to create containers from build images.

There are several docker-compose files in each BE and FE repositories.

docker-compose.yml - used for local environment

docker-compose.dev.yml - used for DEV environment

docker-compose.prod-yml - used for PROD environment

For FE application main difference in all those files are build arguments:

API_URL - URL that is used for API calls (backend calls)

API_PUBLIC_PATH - public path to the root of the application. For example for team12.fiit.stuba.sk/deimos/ public path is "/deimos" and for deimos.fiit.stuba.sk/ public path is "/"

For BE applications we are defining two volumes that will be mounted in created container:

1. Path to docker socket - crucial for creating new containers from backend containers.
2. Path to projects folder where every student project is stored.

For the PROD BE environment we are also creating a postgresql container that contains all our data. This container uses external docker volume **deimos_data** and there two databases **deimos** - for PROD data and **deimos-dev** for DEV data. DO NOT DELETE this volume as all data will be gone. Making backups frequently is also advised. When starting the container we set ***max_connections=500*** for now it's only one start parameter and additional tweaks and adjustments would be necessary.

Usage of Makefile

Each Makefile has three targets and is managing a Docker Compose application.

build-prod/dev/local

Building docker containers for required environments.

down/down-prod

Stopping and recreating containers for required environments. **down-prod** is needed because we don't have to recreate postgresql containers in the PROD environment.

show_logs

The logs option displays the logs for the containers created by docker-compose up.

CI/CD with GitHub actions

GitHub Actions is a CI/CD (Continuous Integration and Continuous Deployment) platform built into GitHub that allows developers to automate software development workflows. With GitHub Actions, developers can automate tasks such as building, testing, and deploying code, without having to set up their own build infrastructure.

The workflows in GitHub Actions are defined in YAML files stored in a repository's **.github/workflows** directory. Workflows are triggered by events, such as pushes to a branch or the creation of a pull request. Each workflow is composed of one or more jobs, and each job is composed of one or more steps. Steps can run commands, run actions from the GitHub Actions Marketplace, or run actions from a Docker container.

With GitHub Actions, developers can perform a wide range of tasks, including:

- Building and testing code
- Automating deployments to various environments (e.g. development, staging, production)
- Building and publishing Docker images
- Automating the release process, including creating and publishing releases, updating documentation, and more.

GitHub Actions also integrates with other tools, including AWS, Google Cloud, and many others, making it easy to incorporate into existing workflows and tools. With GitHub Actions, developers can streamline their software development process and reduce the time and effort required to get code from development to production.

There are different triggers for different pipelines. Pipeline that is responsible for deploying applications to the DEV server is triggered automatically when a new commit is pushed to the **develop** branch. Deploy pipeline for PROD server is launched manually from the github repository (more information here Manually running a workflow - GitHub Docs).

CD pipeline workflow

Purpose: The purpose of this GitHub Action is to deploy the backend/frontend of a project to a remote server.

Backend deploy pipeline uses **env** variables defined in the pipeline that later will be written to the .env file and used by an application.

Steps:

1. Deploy to Server: This step deploys the backend code to the remote server.

- Uses: This step uses the **appleboy/ssh-action** GitHub Action, which is a simple SSH deployment action.
- Inputs: This step requires the following inputs to be provided as secrets in the GitHub repository. ${ENV_NAME} could be either DEV or PROD. All variables or secrets that are set from github repository are:
    - ${ENV_NAME}_DB_USER: username for used database
    - ${ENV_NAME}_DB_PASSWORD: password for used database
    - ${ENV_NAME}_PROJECTS_ABS_PATH: absolute path to STUDENTS projects (C code submitted by students)

    - ${ENV_NAME}_SSH_HOST: the hostname or IP address of the remote server
    - ${ENV_NAME}_SSH_USERNAME: the username to use for authentication with the remote server
    - ${ENV_NAME}_SSH_KEY: the private key for authentication with the remote server
    - ${ENV_NAME}_SSH_PORT: the port number for SSH on the remote server
    - ${ENV_NAME}_PROJECT_PATH: the path to the project on the remote server (Deimos code).
- Script: This step executes the following script:
    - set -e: The set -e command sets the exit status of a script to non-zero if any command in the script returns a non-zero value.
    - eval ssh-agent -s: The `eval `ssh-agent -s command starts an ssh-agent and sets the environment variables required by ssh.
    - ssh-add: The ssh-add command adds private keys to the ssh-agent for use when pulling from github.
    - The script starts by killing the supervisord process.
    - Changes the current directory to the project path on the remote server.
    - Pulls the latest changes from the Git repository.
    - Run the make down command.
    - Runs the make build command.
    - Restarts the nginx service.

Creating ssh keys for GitHub Actions.

Before running the pipeline for the first time we need to make sure we have all necessary SSH keys:

Firstly, we need to generate a key to connect to the server via **ssh-action.** Recommended practice is to generate this key pair on local computer and then upload them to server/GitHub actions

1. Generate key-pair with **ssh-keygen -t rsa -b 4096 -C "<your email>"**. Choose the appropriate name location for this key pair. Do not set any passphrases.
2. Add public key to authorized_keys on remote server with **cat /path/to/new_key_pair/public_key.pub | ssh -i ~/.ssh/team-key.private username@host' cat >> .ssh/authorized_keys'**
3. Copy private key and assign it to SSH_KEY GitHub Actions variable (inside GitHub repo – Settings->Secrets and variables->Actions->New repository secret).

Also, to git commands run without errors we also need to generate keypair on server and upload public key to GitHub. If you are not using GitHub premium you can just bind that key to your GitHub account (Settings->SSH and GPG keys).

DEV and PROD server environments.

The DEV server is assigned to a team by faculty. There is very few space under the root on this virtual server so it would be advisable to move docker data to mounted drive (How to Fix Docker's No Space Left on Device Error | Baeldung on Linux)

The PROD server is shared with another team (ASICDE). So developers always need to consult critical decisions with another team before applying them. Also there are some limitations/changes due to the fact (for example FE application is using different ports for DEV and different for PROD environment). Different domains were created for each application: http://deimos.fiit.stuba.sk/ and http://asicde.fiit.stuba.sk/. Both applications are using nginx to communicate with the client side. Navigate to nginx configuration files to adjust routing.