Worklab 3: Containerize all the apps !

Frontend

Let's us first start by containerizing our frontend. We will start with a really basic dockerfile, then upgrade it, test it, and update our CI pipeline.

Really dumb Dockerfile: `Dockerfile.basic`

Let's first write a `.dockerignore` file, it has the same purpose as a `.gitignore` file but for dockerfiles. That way when you type the `docker build ...` command, files that are listed in the `.dockerignore` file won't be added to the build context of the dockerfile. It will be very basic, but you'll be able to update it later with the other files you'll have created.

```
.github
.ruff_cache
.venv
.gitignore
.pre-commit-config.yaml
makefile
README.md
Dockerfile.basic
```

Nothing too fancy with our first dockerfile, it's pretty basic: we create à `WORKDIR`, install `uv`, `COPY` everything in it and run our frontend.

Dockerfile.basic

```
FROM python:3.12

WORKDIR /app

RUN python -m pip install uv

COPY . /app/

RUN uv sync

EXPOSE 8501

CMD [ "uv", "run", "-m", "streamlit", "run",  "main.py", "--server.port",
"8501"]
```

Note that, although being a best practice, creating a WORKDIR for streamlit apps is necessary. Otherwise Streamlit will through you an error.

Let's build it.

- `docker build -f Dockerfile.basic -t frontend:0.0.1 .`

You can see the size of your image by using the following command.

- `docker image ls | grep front`

This image is too big, around 1.45 GB, no way you can ship that to prod.

To communicate with the Ollama daemon

If you haven't modified your code to communicate with the backend api, to allow your Docker container to communicate with a local Ollma service on port 11434 you have the following method:

- Use the "host" network

This option is simple, but it has security risks because your container will have the same network access as your host.

To use this option, simply launch your container with the `--network="host"` option:

```
docker run --network="host" <image_name>
```

Your container will then be able to access your local service using `localhost:11434` or `127.0.0.1:11434`.

`docker run --rm -it --network="host" frontend:latest`

Standard Dockerfile

First version: `Dockerfile.standard`

Let's change the base image, we don't need a full python distribution, a slim one will do the job.

```
FROM python:3.12-slim-bullseye

WORKDIR /app

RUN python -m pip install uv

COPY . /app/

RUN uv sync

EXPOSE 8501

CMD [ "uv", "run", "-m", "streamlit", "run",  "main.py", "--server.port",
"8501"]
```

This one is better, it is about 560 MB, but it is still lacking on the security side.

But we do not need all the files: the `pyproject.toml` and `uv.lock` are only necessary to build virtual environment, not to run the application. Likewise, we do not need all the dependencies that come with the installation if `uv`, since `uv` is written in Rust, **we only need it's binary**.

Second version: `Dockerfile.standardv2`

```
FROM python:3.12-slim-bullseye
COPY --from=ghcr.io/astral-sh/uv:latest /uv /usr/local/bin/uv

WORKDIR /app

# Install dependencies
RUN --mount=type=cache,target=/root/.cache/uv \
    --mount=type=bind,source=uv.lock,target=uv.lock \
    --mount=type=bind,source=pyproject.toml,target=pyproject.toml \
    uv sync --frozen --no-install-project --no-editable


COPY /rest /app/rest
COPY main.py /app/main.py

EXPOSE 8501

CMD [ "uv", "run", "-m", "streamlit", "run",  "main.py", "--server.port",
"8501"]
```

Difference between `--mount=type=cache` and `--mount=type=bind`

Both `--mount=type=cache` and `--mount=type=bind` are used to mount directories into Docker containers, but they serve different purposes and have key distinctions:

- `--mount=type=bind`

- Functionality: This mounts a file or directory from your host machine directly into the container. Changes made in either location (host or container) are immediately reflected in the other.

- Use Cases:

  - Sharing source code between your development environment on the host and the container.
  - Persisting data generated within the container onto the host's filesystem.
  - Sharing configuration files from the host to containers.

- Characteristics:

  - Direct access to the host's filesystem.
  - Changes are immediately reflected in both locations.
  - Can have security implications if not used carefully (e.g., accidentally modifying important system files on the host).
  - Tied to the host's filesystem structure, making the container less portable.

`--mount=type=cache`

- Functionality: This creates a persistent cache location within the container, which can be used to store things like downloaded packages or compiled artifacts during builds. This cache is preserved across builds, speeding up subsequent builds by reusing cached data.
- Use Cases:
  - Optimizing Docker image builds by caching dependencies or intermediate build results.
  - Useful for package managers like npm, apt, or go mod to avoid redundant downloads.
- Characteristics:
  - Designed for build-time caching, not for general data persistence.
  - Cache is specific to the build context and not included in the final image.
  - Improves build speed by reusing cached data across builds.

Feature	`--mount=type=bind`	`--mount=type=cache`
Purpose	Sharing files/directories between host and container	Caching data during builds
Persistence	Changes are persistent and reflected in both locations	Cache is preserved across builds but not in the final image
Use	Case Development, data persistence, configuration sharing	Optimizing Docker builds
Security	Potential security risks if not used carefully	Less security concerns
Portability	Less portable due to dependency on host filesystem	More portable as cache is managed by Docker

You should also see a small improvement in the size of the image.

Let's improve it again, this time with multi-stage build.

Third version

Multistage builds in Docker are a powerful technique that allows you to use multiple stages within a single Dockerfile to build your application and then copy only the necessary artifacts into the final image. This results in significantly smaller and more efficient final images, as you avoid including build tools, intermediate files, and other unnecessary components in the production-ready container.

How Multistage Builds Work

- Multiple `FROM` instructions

A Dockerfile using multistage builds contains multiple `FROM` instructions. Each `FROM` instruction starts a new stage of the build process.

Think of each stage as a temporary container.

- Naming stages (optional but recommended)

You can give names to your stages using the `AS <stage_name>` syntax (e.g., `FROM node:16 AS builder`). This makes it easier to reference specific stages later in the Dockerfile.

- Copying artifacts between stages

The `COPY --from=<stage_name>` instruction allows you to copy files or directories from a previous stage to the current stage. This is the key to multistage builds.

You copy only the built application artifacts from the build stage to the final image stage.

- Final stage

The last `FROM` instruction in the Dockerfile defines the final image that will be created. Typically, this stage will be based on a minimal image (like `alpine`, `scratch`, or a `slim` variant of your base image) and will only contain the essential files needed to run your application.

Benefits of Multistage Builds

- **Smaller image size**: The most significant advantage. By discarding build tools and intermediate files, the final image becomes much smaller, leading to faster deployments, reduced storage costs, and improved security (fewer vulnerabilities).
- **Improved build performance**: Caching works more effectively with multistage builds. Docker can cache the intermediate stages, speeding up subsequent builds if those stages haven't changed.
- **Better security**: Smaller images have a reduced attack surface as they contain fewer unnecessary components.
- **Cleaner Dockerfiles**: Multistage builds help keep your Dockerfiles organized and readable by separating the build process from the final image creation.
- **Separation of concerns**: You can use different base images for different stages. For example, you might use a larger image with all the build tools in the build stage and a minimal image for the final production stage.

Let's write a dockerfile (`Dockerfile.advanced`) with a multistage build. It might still lack some security, but we will se of we can update it later with either distroless base images or by adding nonroot users.

```
FROM python:3.12-slim-bullseye AS builder

COPY --from=ghcr.io/astral-sh/uv:latest /uv /usr/local/bin/uv

WORKDIR /app

# Install dependencies
RUN --mount=type=cache,target=/root/.cache/uv \
  --mount=type=bind,source=uv.lock,target=uv.lock \
  --mount=type=bind,source=pyproject.toml,target=pyproject.toml \
  uv sync --frozen --no-install-project --no-editable

FROM python:3.12-slim-bullseye

WORKDIR /app

COPY /rest /app/rest
COPY main.py /app/main.py

# Copy the environment
COPY --from=builder --chown=app:app /app/.venv /app/.venv

# Set environment variables
ENV PATH="/app/.venv/bin:$PATH" PYTHONPATH="app/.venv/lib/python3.12/site-
packages"

EXPOSE 8501

# Let's be specific about which python we'll use.
# Note that we do not have uv anymore in this stage !
CMD ["app/.venv/bin/python3.12", "-m", "streamlit", "run",  "app/main.py",
"--server.port", "8501"]
```

Again, you might see a small improvement in the size of the image.

We need to go deeper, let's go distroless

Distroless images are Docker images stripped down to their bare essentials. They contain only the necessary components to run your application, such as the runtime and libraries, and exclude any unnecessary system tools, packages, or shells.

Why Use Distroless Images?

- **Security**: By reducing the attack surface, distroless images minimize the potential vulnerabilities that could be exploited.
- **Size**: These images are significantly smaller than traditional images, leading to faster downloads and deployments.
- **Performance**: The smaller size and fewer components contribute to faster container startup times.
- **Reliability**: With fewer components, there's less potential for conflicts or unexpected behaviors.

How They Work

Distroless images are typically built from a base image that contains only the fundamental components required for the target runtime (e.g., Node.js, Python, Java). Any unnecessary packages or tools are then removed, resulting in a highly minimal image.

Example

```
FROM gcr.io/distroless/nodejs:18

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

CMD ["node", "index.js"]
```

In this example, we use a Node.js-based distroless image. The image contains only the necessary components to run a Node.js application, providing a secure and lightweight environment.

Considerations and Limitations

- **Configuration**: Configuring distroless images can be more complex as you'll need to manage certain aspects manually that are typically handled by the underlying OS.
- **Dependencies**: Ensure all necessary dependencies are included in the image.
- **Tooling**: You might not have access to the same set of tools as in a full-fledged Linux distribution. In particular, **you will never have access to a terminal is a distroless image**.

Google distroless

Google Distroless images are a specialized type of container image that offers a minimalist approach to containerization. By removing unnecessary components, these images significantly reduce the attack surface, making them highly secure.

Canonical chisel

Chiseled Ubuntu Images are a specialized type of container image developed by Canonical, the company behind the Ubuntu operating system. They take the concept of "distroless" images a step further by focusing on creating extremely minimal and secure Ubuntu-based container images.

Key Characteristics:

- **Minimized Footprint**: Chiseled Ubuntu images are carefully crafted to include only the absolute essentials for your application to run. This results in significantly smaller image sizes compared to traditional Ubuntu images.
- **Enhanced Security**: By removing unnecessary packages and system utilities, these images minimize the attack surface, making them more secure against potential vulnerabilities.
- **Ubuntu Foundation**: Built upon the robust and well-supported Ubuntu Linux distribution, providing a solid foundation for your applications.
- **Chisel Tool**: Canonical provides a tool called "Chisel" which allows developers to fine-tune the image creation process. This enables you to precisely select the specific components required for your application, further minimizing the image size and enhancing security.

Benefits:

- **Reduced Image Size**: Smaller images lead to faster downloads and deployments, saving time and resources.
- **Improved Security**: Minimized attack surface translates to enhanced security for your applications.
- **Improved Performance**: Smaller images generally lead to faster startup times and reduced resource consumption.
- **Cost Savings**: Smaller images can significantly reduce storage costs and network bandwidth usage.
- **Enhanced Efficiency**: Faster deployments and improved resource utilization contribute to increased efficiency.

Use Cases:

- **Microservices**: Ideal for microservices architectures where small, efficient, and secure containers are crucial.
- **Edge Computing**: Suitable for edge deployments where bandwidth and storage are limited.
- **Cloud-Native Applications**: Well-suited for cloud environments where efficiency and scalability are paramount.

Chiseled Ubuntu images represent a significant advancement in container image optimization. By combining the security and reliability of Ubuntu with a focus on extreme minimalism, they offer a compelling solution for building and deploying secure, efficient, and lightweight containerized applications.

How to test your docker image

Terratest

Terratest is a Go library that makes it easier to write automated tests for your infrastructure code. It provides a variety of helper functions and patterns for common infrastructure testing tasks, including:

- Testing Terraform code
- Testing Packer templates
- Testing Docker images
- Executing commands on servers over SSH
- Working with AWS APIs
- Working with Azure APIs
- Working with GCP APIs
- Working with Kubernetes APIs
- Enforcing policies with OPA
- Testing Helm Charts
- Making HTTP requests
- Running shell commands
- And much more

While Terratest is a powerfull tool, it is a bit out of scope and we are gonna focus and something a bit more easier to understand: Container Structure Test.

Container Structure Tests

Container Structure Tests provide a powerful framework to validate the structure of a container image. These tests can be used to check the output of commands in an image, as well as verify metadata and contents of the filesystem.

Container Structure Test (CST) is a framework for validating the structure of Docker images.

Focus: CST primarily focuses on the structure of the image, not its functionality. It checks things like:

- File existence: Does the image contain the expected files and directories?
- File content: Do files contain the expected data or configurations?
- Command execution: Do specific commands within the image produce the expected output?
- Image metadata: Are the image's metadata (labels, entrypoint, etc.) as expected?

Testing Types: CST supports various types of tests:

- Command Tests: Verify the output of commands executed within the container.
- File Existence Tests: Check for the presence or absence of files and directories.
- File Content Tests: Check the contents of files for specific patterns or values.
- Metadata Tests: Verify the image's metadata, such as labels, entrypoint, and exposed ports.

Benefits:

- Early Detection of Issues: Identify problems early in the development cycle, before deploying the image.
- Improved Image Quality: Ensure consistent and reliable images by enforcing structural integrity.
- Enhanced Security: Help identify potential security vulnerabilities, such as unexpected files or misconfigurations.
- Simplified Auditing: Facilitate auditing and compliance checks by providing a clear record of image structure.

Example:

A simple CST test might check:

- If a specific configuration file exists in the image.
- If a particular command (like `ls -l`) returns the expected output.
- If the image exposes the correct ports.

CST acts as a quality gate for your Docker images, ensuring they adhere to defined standards and best practices. By incorporating CST into your CI/CD pipeline, you can significantly improve the reliability and security of your containerized applications.

Basic dockerfile

Let's create a `tests` subdirectory at the root of your frontend directory. We add a YAML file called `cst-basic.yaml` with the following content. This will test the basic Dockerfile `Dockerfile.basic` we just wrote.

```yaml
---
schemaVersion: 2.0.0

fileExistenceTests:
    - name: main
      path: /app/main.py
      shouldExist: true

commandTests:
    - name: python version
      command: python
      args: [--version]
      expectedOutput: [Python 3.12.*]

metadataTest:
    workdir: /app
    exposedPorts: ["8501"]
```

What are we doing here ?

1. Schema Version:

schemaVersion: 2.0.0: This line specifies the version of the CST schema being used. Adhering to a specific schema version ensures compatibility and maintainability of your tests.

2. File Existence Tests:

fileExistenceTests:, this section defines tests to check for the existence of specific files or directories within the image. **- name: main**, This is the name of the test, providing a human-readable identifier. **path: /app/main.py**, specifies the path to the file that should exist within the container. **shouldExist: true** This boolean indicates that the file at the specified path is expected to exist.

3. Command Tests:

commandTests: This section defines tests that execute commands within the container and verify their output. **- name: python version** The name of the command test. **command: python** The command to be executed within the container. **args: [--version]** Optional arguments to be passed to the command. **expectedOutput: [Python 3.12.*]** The expected output from the command. The * acts as a wildcard to match any version within the 3.12 series.

4. Metadata Test:

metadataTest: This section defines tests related to the image's metadata. **workdir: /app** Specifies the working directory that should be set within the container. **exposedPorts: ["8501"]** Defines the ports that the container should expose.

How to run CST ?

If you run on Linux, WSL2, or macOS, you can follow the following steps to test your container locally.

1. Install CST: https://github.com/GoogleContainerTools/container-structure-test
2. Build your docker image with the following command: `docker build -f Dockerfile.basic -t frontend:0.0.1 .`
3. Run the following command to run CST against your built image : `container-structure-test test --image frontend:0.0.1 --config tests/cst-basic.yaml`.

If every thing goes well, you will see a report in your terminal that all the tests have passed, or not.

If you're on windows, I'm sorry but you can't install CST on Windows. But you will see how to run it in your GitHub pipelines, so don't worry.

Your docker CI pipeline

Writing the pipeline

In the `.github/workflows` directory, let's add a new file called `docker-ci.yaml`. It will look like this.

```
name: Build and Push Docker Image

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build the Docker image
        run: docker build -f Dockerfile.basic -t ghcr.io/mathieu-
junia/frontend:${{ github.sha }} .

      - name: Log in to the registry
        run: docker login ghcr.io -u ${{ secrets.GITHUB_TOKEN }} -p ${{
secrets.GITHUB_TOKEN }}

      - name: Push the Docker image
        run: docker push ghcr.io/mathieu-junia/frontend:${{ github.sha }}
```

This pipeline is designed to:

- Trigger on pushes to the "main" branch.
- Check out the repository's code.
- Build a Docker image using the specified Dockerfile.
- Log in to the GitHub Container Registry: `ghcr.io`.
- Push the built image to the registry with a unique tag based on the commit SHA.

Note here that the location of the image `ghcr.io/mathieu-junia/frontend:${{ github.sha }}` depends on:

- the name of your GitHub organization, which is `mathieu-junia` for me,
- the name of the image you chose, which is `frontend` for me.

About the `GITHUB_TOKEN` secret

At the start of each workflow job, GitHub automatically creates a unique `GITHUB_TOKEN` secret to use in your workflow. You can use the `GITHUB_TOKEN` to authenticate in the workflow job by using the synthax `${{ secrets.GITHUB_TOKEN }}`.

When you enable GitHub Actions, GitHub installs a GitHub App on your repository. The `GITHUB_TOKEN` secret is a GitHub App installation access token. You can use the installation access token to authenticate on behalf of the GitHub App installed on your repository. The token's permissions are limited to the repository that contains your workflow.

By default, the `GITHUB_TOKEN` does not have the write to push packages in the container registry. You have to add this permission.

To do that, inside your repository, go to `settings`, click on `Actions` on the left menu and select `General`. Select "Read and write permissions" in the "Workflow permissions" and save. This will allow the `GITHUB_TOKEN` to push docker images on the github container registry.

Adding CST to the pipeline

Let's modify the `docker-ci` pipeline with the following lines.

```
name: Build and Push Docker Image

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Build the Docker image
        run: docker build -f Dockerfile.basic -t ghcr.io/mathieu-
junia/frontend:${{ github.sha }} .

      - name: Install and run Container Structure Tests
        run: |
          curl -LO https://github.com/GoogleContainerTools/container-
structure-test/releases/latest/download/container-structure-test-linux-
amd64
          chmod +x container-structure-test-linux-amd64
          mkdir -p $HOME/bin
          export PATH=$PATH:$HOME/bin
          mv container-structure-test-linux-amd64 $HOME/bin/container-
structure-test
          container-structure-test test --image ghcr.io/mathieu-
junia/frontend:${{ github.sha }} --config tests/cst-basic.yaml

      - name: Log in to the registry
        run: docker login ghcr.io -u ${{ secrets.GITHUB_TOKEN }} -p ${{
secrets.GITHUB_TOKEN }}

      - name: Push the Docker image
        run: docker push ghcr.io/mathieu-junia/frontend:${{ github.sha }}
```

The installation instructions of CST can be found on the repository page of CST.

Now that we have added CST to our CI pipeline, lets continue writing more tests.

CST for the standard Dockerfile

The `Dockerfile.standard` dockerfile is not that different from the basic one, we just changed the base image. Let's focus on the `Dockerfile.standardv2` dockerfile.

Look at the Dockerfile, what could we test ?

- There should be a uv binary file
- There should be an `app/rest` directory
- There should be an `app/main.py` file

- Exposed port should be 8501
- The base image should be from the 3.12.* series
- There shlouldn't be a `uv.lock` file
- There shlouldn't be a `pyproject.toml` file

We then have the following yaml file.

```yaml
---
schemaVersion: 2.0.0

fileExistenceTests:
    - name: main file
      path: /app/main.py
      shouldExist: true
    - name: rest directory
      path: /app/rest
      shouldExist: true
    - name: uv binary
      path: /usr/local/bin/uv
      shouldExist: true
    - name: uv.lock file
      path: /app/uv.lock
      shouldExist: false
    - name: pyproject.toml file
      path: /app/pyproject.toml
      shouldExist: false


commandTests:
    - name: python version
      command: python
      args: [--version]
      expectedOutput: [Python 3.12.*]

metadataTest:
    workdir: /app
    exposedPorts: ["8501"]
```

Then we can use to test our Dockerfile.

- `docker build -f Dockerfile.standardv2 -t frontend:0.0.2 .`
- `container-structure-test test --image frontend:0.0.2 --config tests/cst-standardv2.yaml`

CST for the advanced Dockerfiles

First version

- The `uv` binary shoudln't be here anymore
- We have added a new environment variable: `PYTHONPATH`, let's check it's here.

```
---
schemaVersion: 2.0.0

fileExistenceTests:
    - name: main file
      path: /app/main.py
      shouldExist: true
    - name: rest directory
      path: /app/rest
      shouldExist: true
    - name: venv directory
      path: /app/.venv
      shouldExist: true
    - name: uv binary
      path: /usr/local/bin/uv
      shouldExist: false
    - name: uv.lock file
      path: /app/uv.lock
      shouldExist: false
    - name: pyproject.toml file
      path: /app/pyproject.toml
      shouldExist: false


commandTests:
    - name: python version
      command: python
      args: [--version]
      expectedOutput: [Python 3.12.*]
    - name: python path
      command: which
      args: [python]
      expectedOutput: [app/.venv/bin/python*]

metadataTest:
    workdir: /app
    exposedPorts: ["8501"]
    envVars:
      - key: PATH
        value: /app/.venv/bin.*
        isRegex: true
      - key: PYTHONPATH
        value: "app/.venv/lib/python3.12/site-packages"
```

- `docker build -f Dockerfile.advanced -t frontend:0.0.3 .`
- `container-structure-test test --image frontend:0.0.3 --config tests/cst-advvanced.yaml`

Second version

In the second version of this Dockerfile, we also have added a nonroot user. This is one of the best practice that you should always enforce. No container should run as a root user.

We have done that by adding the following lines to the dockerfile.

```
# create nonroot user
ARG USERNAME=nonroot
ARG USER_UID=65532
ARG USER_GID=$USER_UID

RUN groupadd --gid $USER_GID $USERNAME \
  && useradd --uid $USER_UID --gid $USER_GID -m $USERNAME
```

This Dockerfile snippet creates a non-root user within the container. Let's break down the steps:

1. Define Arguments:

- `ARG USERNAME=nonroot`: Defines an argument named `USERNAME` with a default value of "nonroot". You can override this value during image build time using the --build-arg flag.
- `ARG USER_UID=65532`: Defines an argument named `USER_UID` with a default value of 65532. This will be the user ID of the non-root user.
- `ARG USER_GID=$USER_UID`: Defines an argument named `USER_GID` and sets its value to the same as USER_UID. This ensures the user and group IDs are the same.

2. Create User and Group:

- `RUN groupadd --gid $USER_GID $USERNAME`: This command creates a new group with the specified group ID ($USER_GID) and the group name ($USERNAME).
- `&& useradd --uid $USER_UID --gid $USER_GID -m $USERNAME`: This command creates a new user with the specified user ID ($USER_UID), group ID ($USER_GID), and username ($USERNAME). The `-m` flag creates the user's home directory.

Benefits of using a non-root user:

Security: Running applications as a non-root user minimizes the potential damage if the application is compromised. **Reduced attack surface**: Limiting privileges reduces the number of system resources the application can access. **Best practices**: Running applications as non-root users is considered a security best practice for containerized environments.

In a Linux host, you can check your user id, group id, and your name by using the commands `id -u`, `id -g`, and `whoami`. So let's do that in our test file.

```
---
schemaVersion: 2.0.0

fileExistenceTests:
    - name: main file
      path: /app/main.py
      shouldExist: true
    - name: rest directory
      path: /app/rest
      shouldExist: true
    - name: venv directory
      path: /app/.venv
      shouldExist: true
    - name: uv binary
      path: /usr/local/bin/uv
      shouldExist: false
    - name: uv.lock file
      path: /app/uv.lock
      shouldExist: false
    - name: pyproject.toml file
      path: /app/pyproject.toml
      shouldExist: false


commandTests:
    - name: python version
      command: python
      args: [--version]
      expectedOutput: [Python 3.12.*]
    - name: python path
      command: which
      args: [python]
      expectedOutput: [app/.venv/bin/python*]
    - name: whoami
      command: whoami
      expectedOutput: [nonroot]
    - name: check user id
      command: id
      args: [-u]
      expectedOutput: [65532]
    - name: check group id
      command: id
      args: [-g]
      expectedOutput: [65532]

metadataTest:
    workdir: /app
    exposedPorts: ["8501"]
    envVars:
      - key: PATH
        value: /app/.venv/bin.*
        isRegex: true
      - key: PYTHONPATH
        value: "app/.venv/lib/python3.12/site-packages"
```