**Лабораторна робота №3**

**Vikulin Volodymyr IM-11**

**Python**

1. First, we will create a container description for the application. We will add application code, dependencies to this image, and build our image. We can find out the necessary dependencies using the command “pip freeze”

Dockerfile:

FROM python:3.11.2-bullseye

WORKDIR /app

COPY . .

RUN pip install --no-cache-dir -r requirements/backend.in

CMD ["uvicorn", "spaceship.main:app", "--host=0.0.0.0", "--port=8080"]

Dependencies:

anyio==3.6.2

click==8.1.3

colorama==0.4.6

fastapi==0.95.0

h11==0.14.0

httptools==0.5.0

idna==3.4

pydantic==1.10.7

python-dotenv==1.0.0

PyYAML==6.0

sniffio==1.3.0

starlette==0.26.1

typing\_extensions==4.5.0

uvicorn==0.21.1

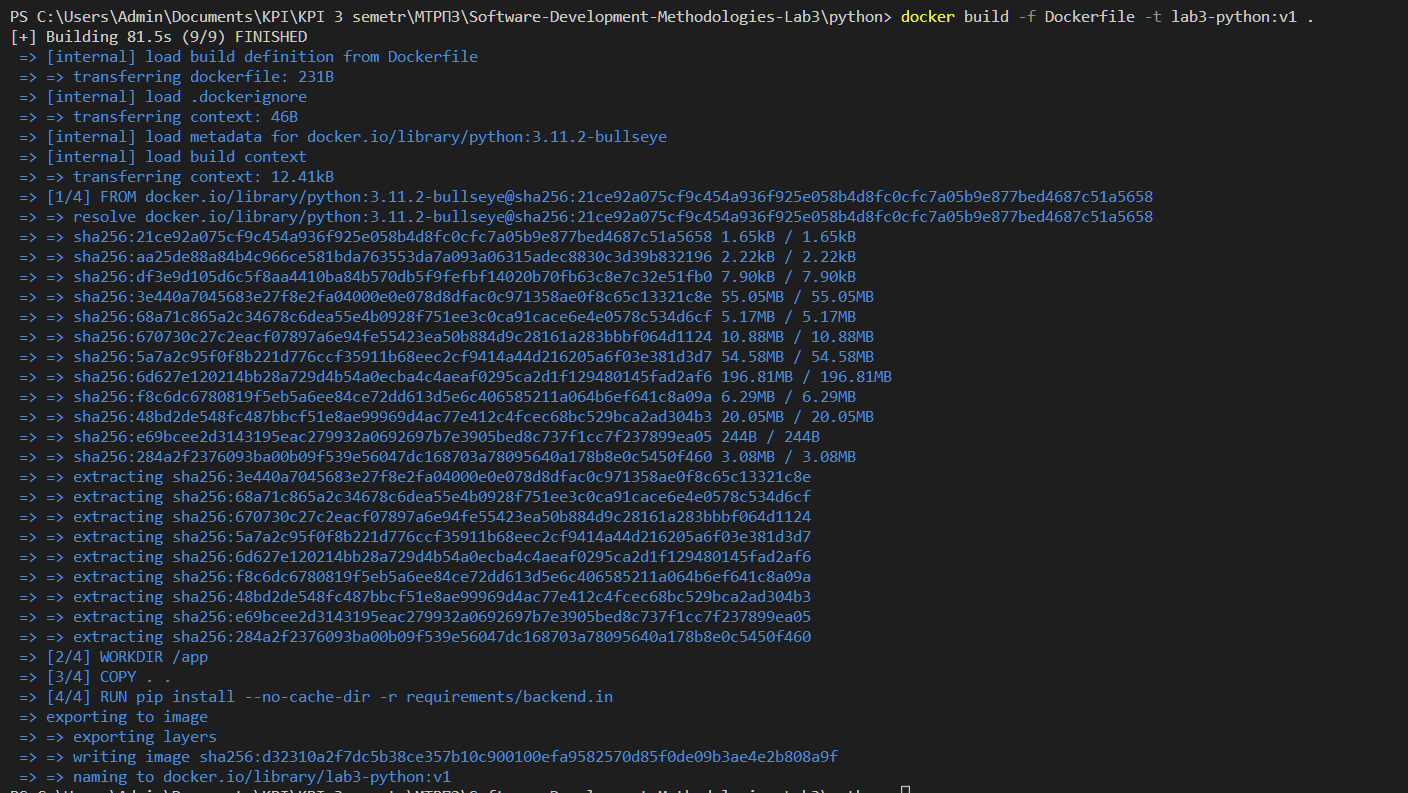
watchfiles==0.19.0

websockets==10.4

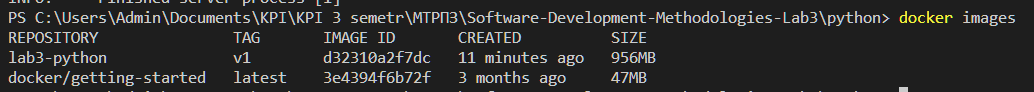
Added dockerignore

.venv/

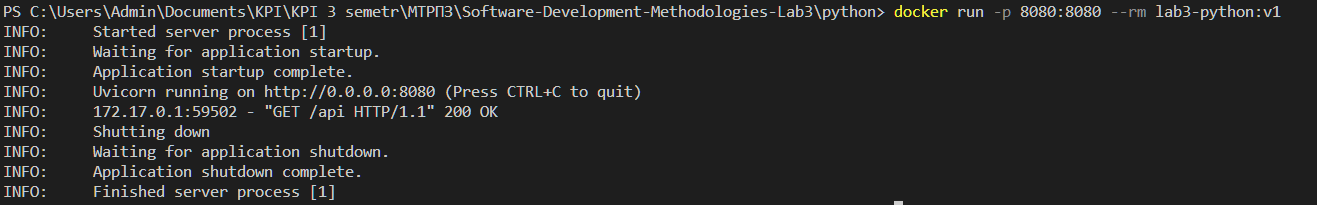
Build command: **docker build -f Dockerfile -t lab3-python:v1 .**



To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-python:v1**



Build results:

* Image size: 956 MB
* Image build time 81.5 s

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/eb51c9844782c38419fd243a40a2010f0c8d7329)

1. After that, we will make changes to the file spaceship/app.py (add a comment, force the program to display student info) and build the image with the new version of the code.

**Changed file app.py:**

@router.get('')

def hello\_world() -> dict:

    return {

        'student': 'Vikulin Volodymyr',

          'group': 'IM-11',

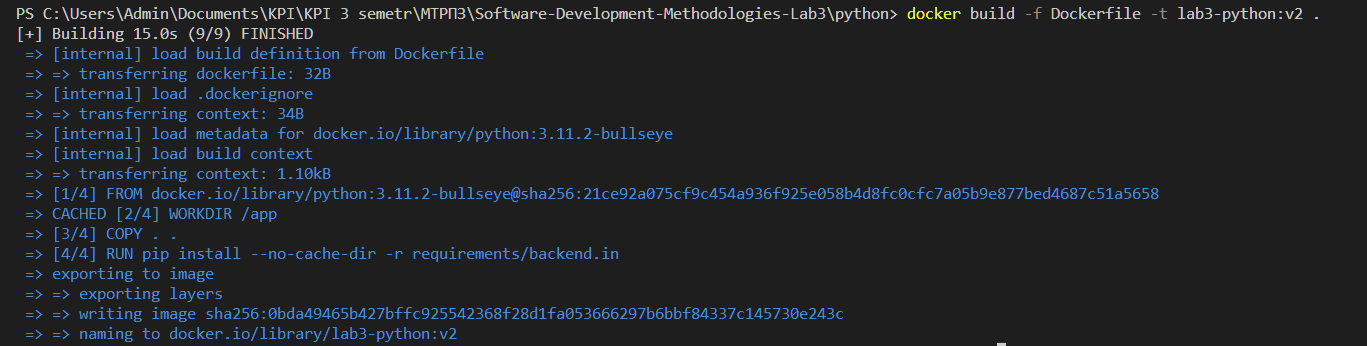
        'Country': 'Ukraine',

        'City': 'Kyiv',

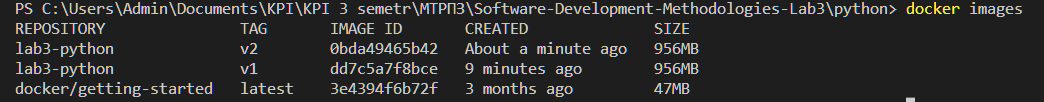
        'University': 'KPI',

    }

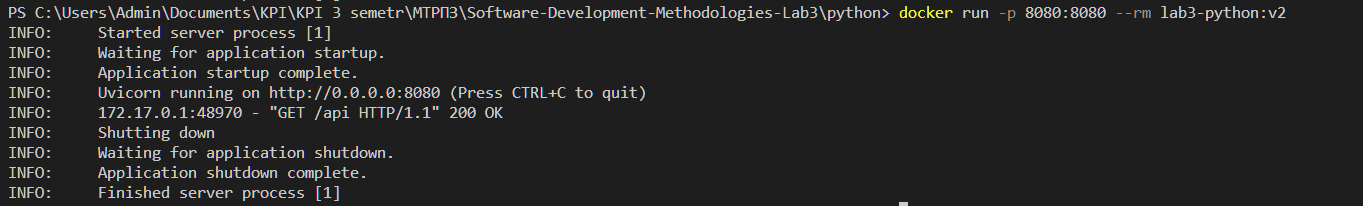
Build command: **docker build -f Dockerfile -t lab3-python:v2 .**



To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-python:v2**



Build results:

* Image size: 956 MB
* Image build time 15 s

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/cdedc0fdbe548b81e5ca5d31df016eecea7b2723)

1. We will rewrite the image description file (dockerfile) in such a way as to effectively use layers for building the image. To do this, we first add to the image what will change least often (project dependencies), and those files that will change more often (our code) add last.

Dockerfile:

FROM python:3.11.2-bullseye

WORKDIR /app

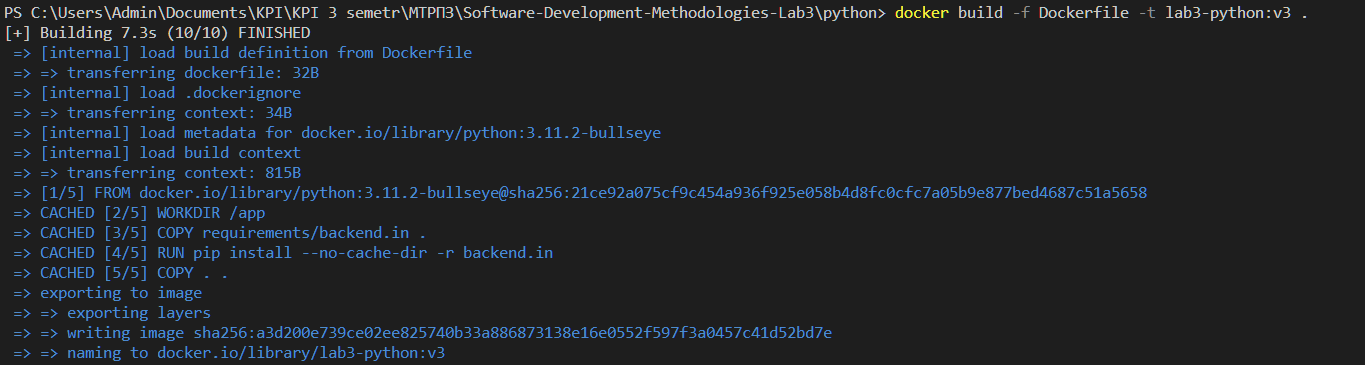
COPY requirements/backend.in .

RUN pip install --no-cache-dir -r backend.in

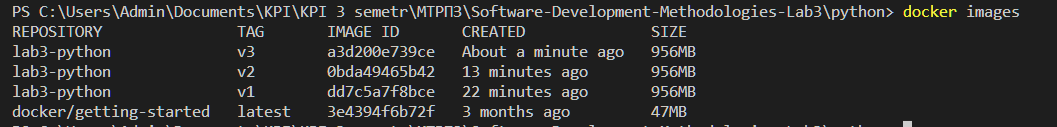
COPY . .

CMD ["uvicorn", "spaceship.main:app", "--host=0.0.0.0", "--port=8080"]

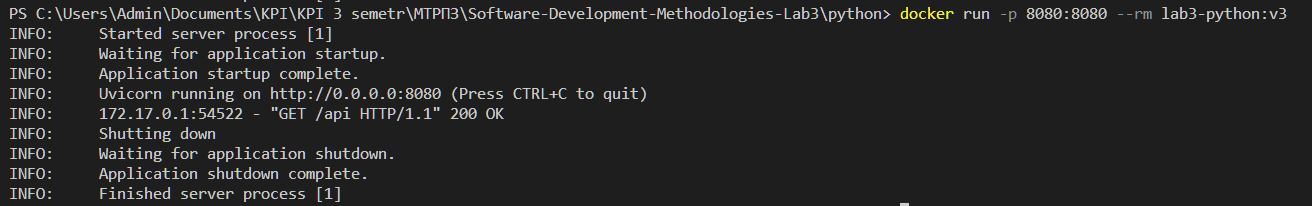
Build command**: docker build -f Dockerfile -t lab3-python:v3 .**



To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-python:v3**



Build results:

* Image size: 956 MB
* Image build time 7.3 s

You can see that the layer with dependencies is used from the cache, and does not copy along with the contents of the directory

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/e3c828f78310e0561d0d2c8ea98bab8da79a4f9f)

1. Let's take a smaller basic image as the basis of the project. For example, instead of python:3.11.2-bullseye (based on debian), we will try python:3.11.2-alpine (based on alpine).

Dockerfile:

FROM python:3.11.2-alpine

WORKDIR /app

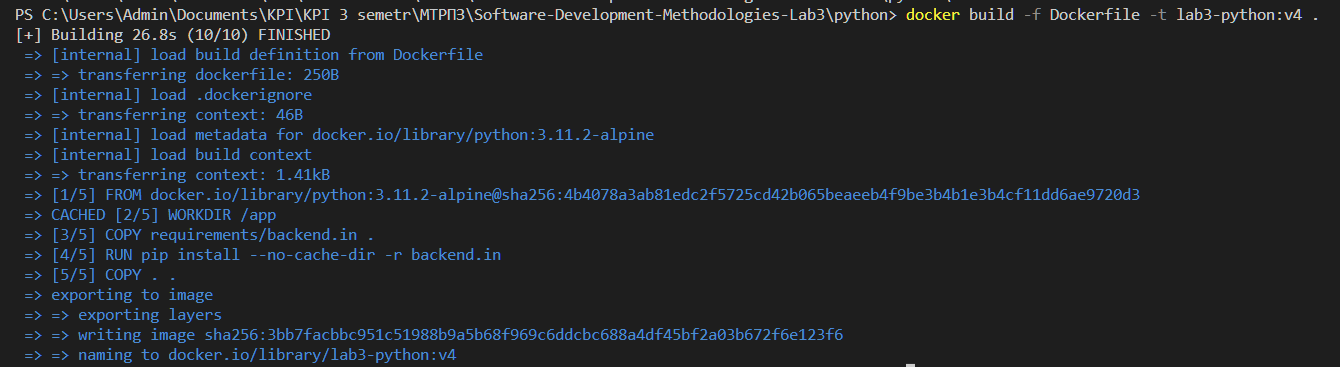
COPY requirements/backend.in .

RUN pip install --no-cache-dir -r backend.in

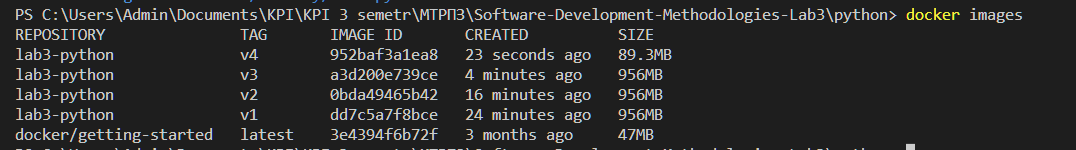
COPY . .

CMD ["uvicorn", "spaceship.main:app", "--host=0.0.0.0", "--port=8080"]

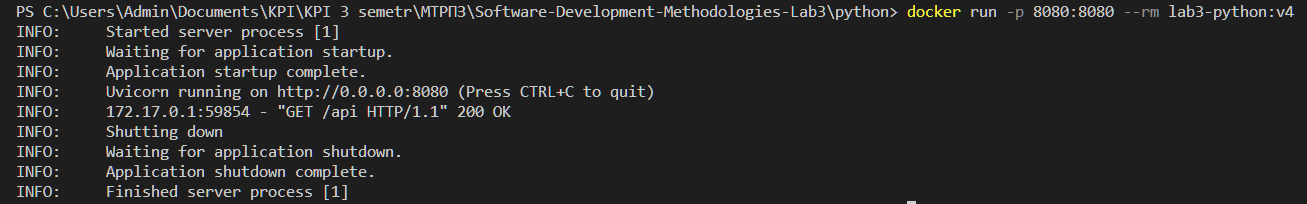
Build command: **docker build -f Dockerfile -t lab3-python:v4 .**



To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-python:v4**



Build results:

* Image size: 89.3 MB
* Image build time 26.8 s

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/dd2f4193c07a2b99dcd6f0120cc45537d8aaa164)

1. Let's add a numpy dependency to the project, and add an endpoint to spaceship/routers/api.py that will generate 2 random 10x10 matrices and multiply them together. The endpoint will return a dictionary of the following structure:

{

“matrix\_a”: [[...], ...],

“matrix\_b”: [[...], ...],

“product”: [[...], ...],

}

Added dependence:

numpy==1.24.2

**Alpine**

Dockerfile:

FROM python:3.11.2-alpine

WORKDIR /app

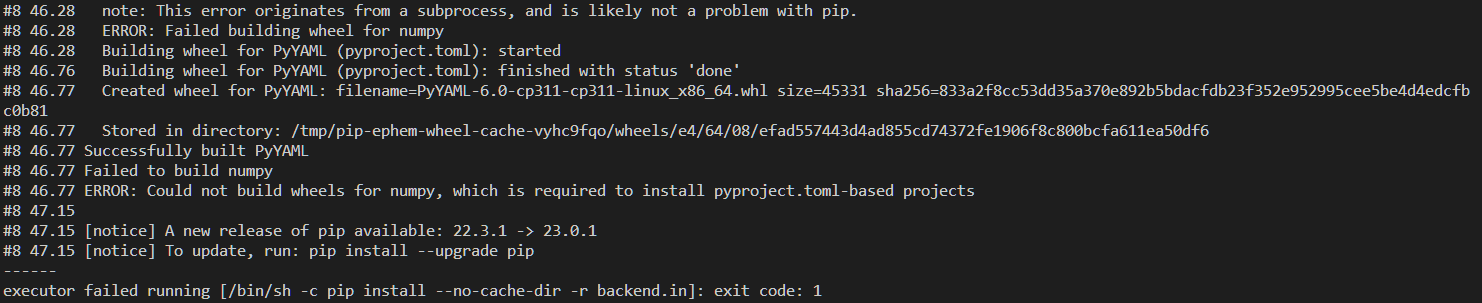
COPY requirements/backend.in .

RUN pip install --no-cache-dir -r backend.in

COPY . .

CMD ["uvicorn", "spaceship.main:app", "--host=0.0.0.0", "--port=8080"]

Build command: **docker build -f Dockerfile -t lab3-python:apline .**



After adding numpy to the dependency list, the installation failed, so I added one more line to the docker to solve it:

FROM python:3.11.2-alpine

RUN apk add --no-cache g++ gcc lapack-dev musl-dev

WORKDIR /app

COPY requirements/backend.in .

RUN pip install --no-cache-dir -r backend.in

COPY . .

CMD ["uvicorn", "spaceship.main:app", "--host=0.0.0.0", "--port=8080"]

app.py endpoint:

@router.get('/matrix\_mult')

def matrix() -> dict:

    matrix\_a = np.random.rand(10, 10).tolist()

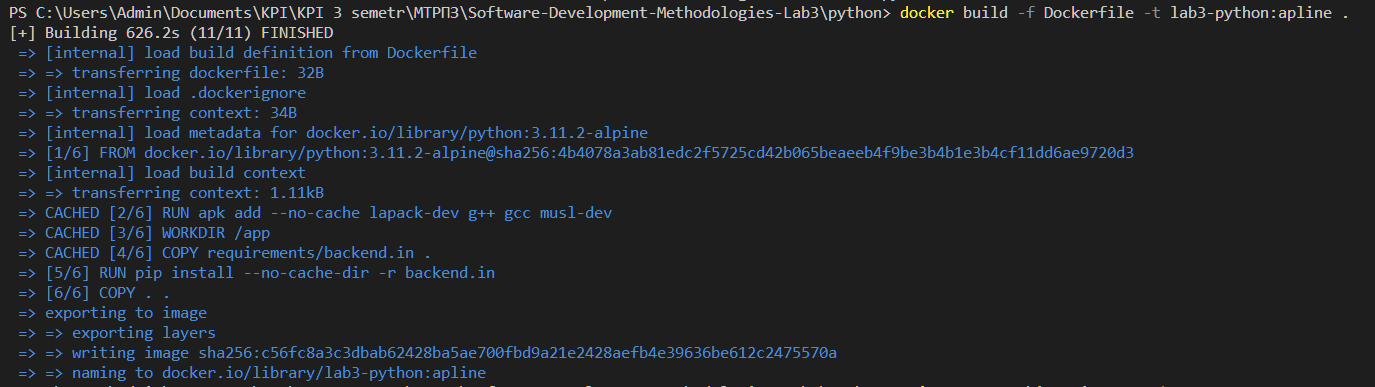
    matrix\_b = np.random.rand(10, 10).tolist()

    product = np.dot(matrix\_a, matrix\_b).tolist()

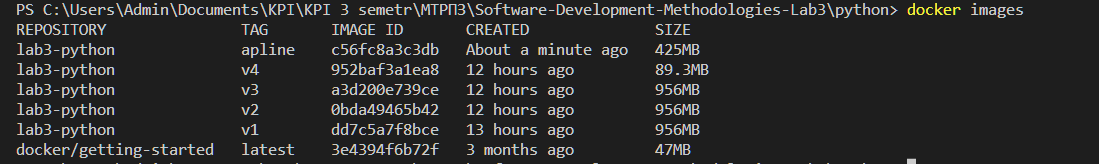
    result = {"matrix\_a": matrix\_a, "matrix\_b": matrix\_b, "product": product}

    return result

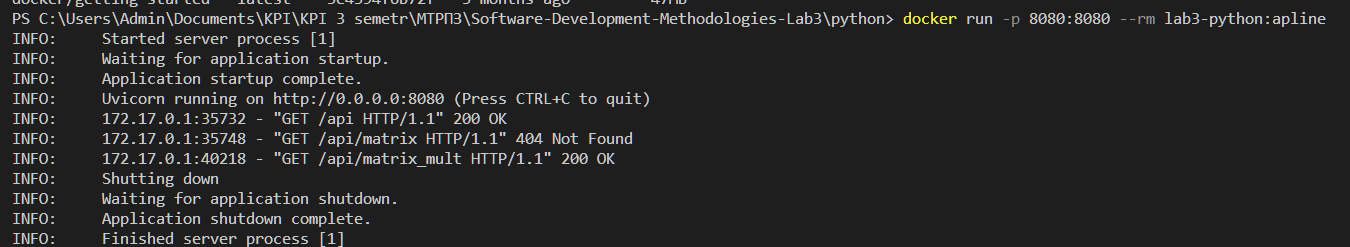
Build command: **docker build -f Dockerfile -t lab3-python:apline .**



To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-python:apline**



Build results:

* Image size: 425 MB
* Image build time 626.2 s

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/bc53ea7a56e953c5250eea7aa120dbd224cb129a)

**Bullseye**

Dockerfile:

FROM python:3.11.2-bullseye

WORKDIR /app

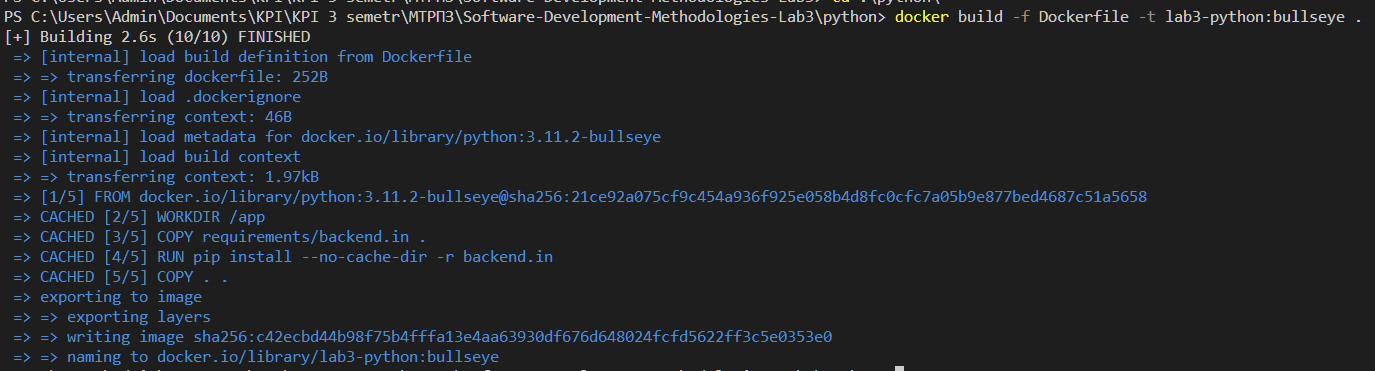
COPY requirements/backend.in .

RUN pip install --no-cache-dir -r backend.in

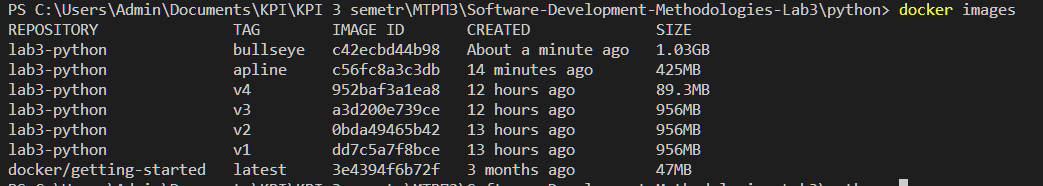
COPY . .

CMD ["uvicorn", "spaceship.main:app", "--host=0.0.0.0", "--port=8080"]

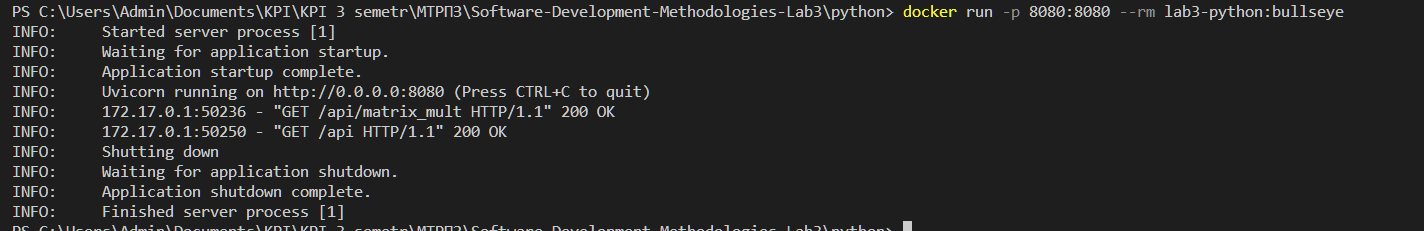
Build command: **docker build -f Dockerfile -t lab3-python:bullseye .**



To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-python:apline**



Build results:

* Image size: 1.03 GB
* Image build time 2.6 s

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/58a6b04b228e066db25508a0c439c7e03922db67)

**Comparison of the build of the two images:** The size and time of the alpine build turned out to be much larger compared to the bullseye-based image. The build time in the bullseye image is minimal, since in my case all layers were taken from the cache, but at the first build, all python dependencies will be installed again, since the backend.in file was updated. The size of the image does not reflect reality, since compared to previous images, the directory of the virtual environment was not copied here, which significantly reduced the size of the image.

**Golang**

1. Let's create an image of the project, add dependencies to the container, build the executable file

Dockerfile:

FROM golang:1.20.2

WORKDIR /app

COPY go.mod go.sum /app/

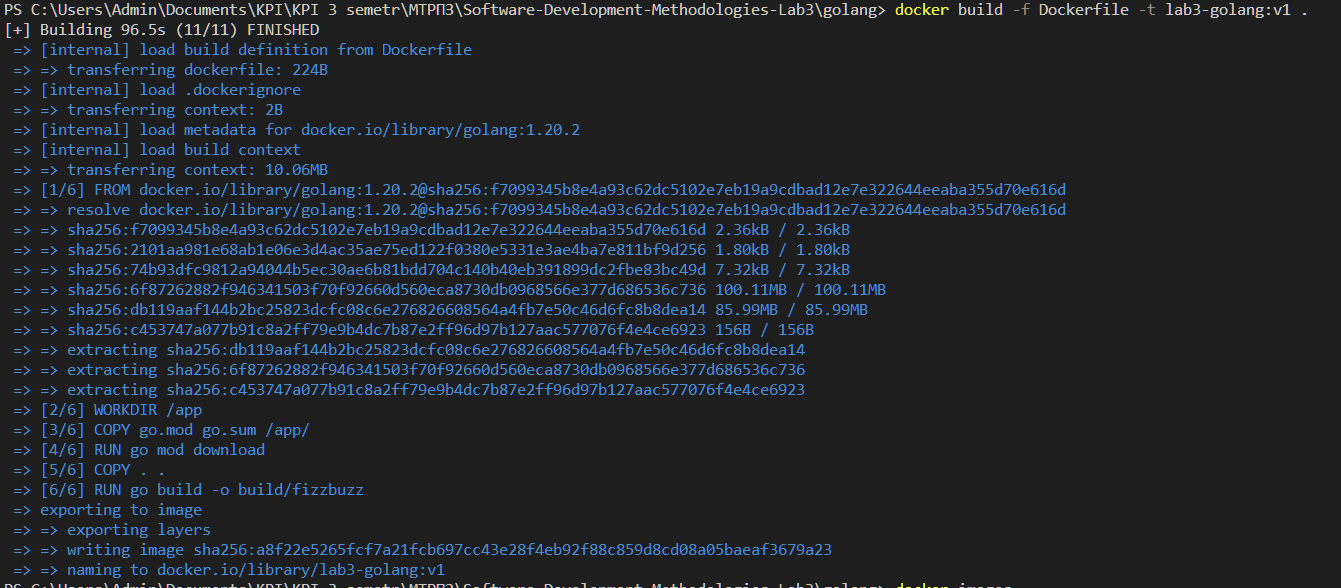
RUN go mod download

COPY . .

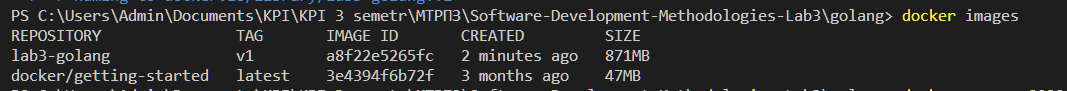
RUN go build -o build/fizzbuzz

CMD ["./build/fizzbuzz", "serve", "--port=8080"]

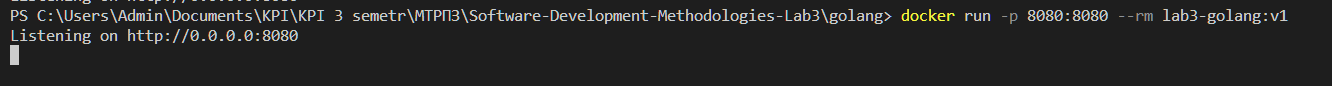
Build command: **docker build -f Dockerfile -t lab3-golang:v1 .**

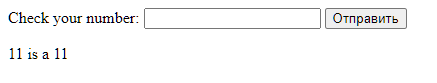


To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-golang:v1**





Build results:

* Image size: 871 MB
* Image build time 96.5 s

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/45cf87a907cf0a105fa2397436e7ea4d6fd2bf12)

**Are all the files there needed to run the project?**

No, because to run the application, only the binary file /build/fizzbuzz is needed, which we created with the command “go build -o build/fizzbuzz”

1. Let's make a multi-stage assembly. To do this, at the first stage, we collect a binary file, and at the next stage, we copy this build file into a new empty image (FROM scratch).

Dockerfile:

FROM golang:1.20.2 AS builder

WORKDIR /app

COPY go.mod go.sum /app/

RUN go mod download

COPY . .

RUN CGO\_ENABLED=0 go build -o build/fizzbuzz

FROM scratch

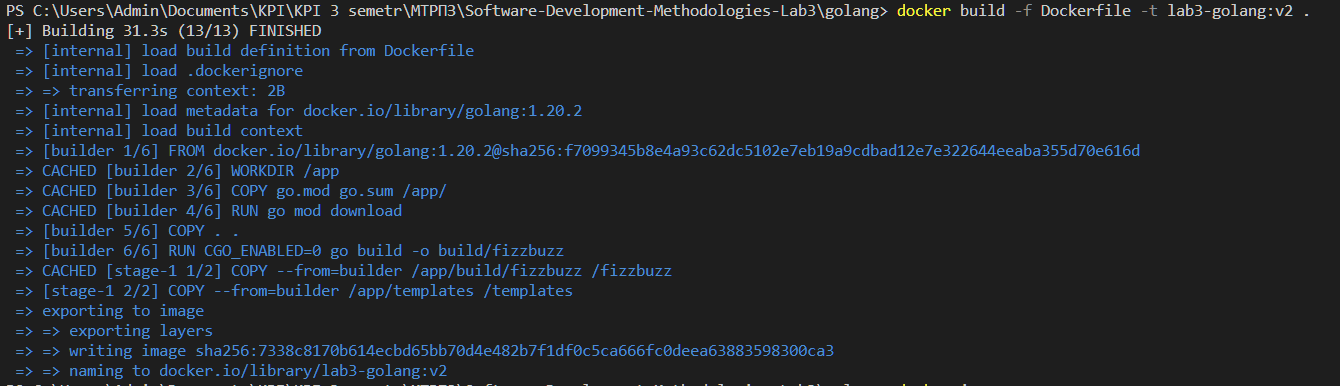
COPY --from=builder /app/build/fizzbuzz /fizzbuzz

COPY --from=builder /app/templates /templates

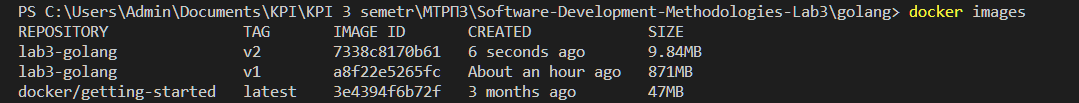
CMD ["/fizzbuzz", "serve", "--port=8080"]

We changed the image creation in docker because the executable was statically compiled and had no external dependencies:

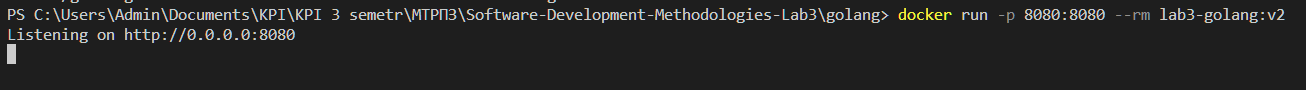
Build command: **docker build -f Dockerfile -t lab3-golang:v2 .**

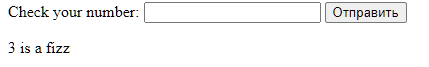


To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-golang:v2**





Now, when you run Dockerfile, a multi-stage container will be assembled that contains only the necessary files for the application to work, which can reduce the size of the image and increase the speed of its deployment.

Build results:

* Image size: 9.84 MB
* Image build time 31.3 s

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/5ec484deda9537de3094c6380d80d575621025ca)

**Are there enough files to run our project?**

Yes, the files are enough, all dependencies are included in the executable because the executable is compiled statically.

**Is it convenient to use this way?**

A multi-stage build based on scratch can be quite a convenient solution in some cases, especially if you want to create a very lightweight Docker image that does not contain any redundant OS components.

In addition, using a multi-stage build allows to reduce the size of the Docker image, as you can remove all the redundant components that are only used to build the application.

However, using the base scratch image can be difficult because it doesn't have any OS components and you have to manually add all the required libraries and other components that are required for the application to run.

So, using a multi-stage scratch-based build may be convenient if you want to create a very lightweight and efficient Docker image, but it may not be the best choice for all scenarios.

1. Instead of scratch, we will use images from the distroless project

Dockerfile:

FROM golang:1.20.2 AS builder

WORKDIR /app

COPY go.mod go.sum /app/

RUN go mod download

COPY . .

RUN go build -o build/fizzbuzz

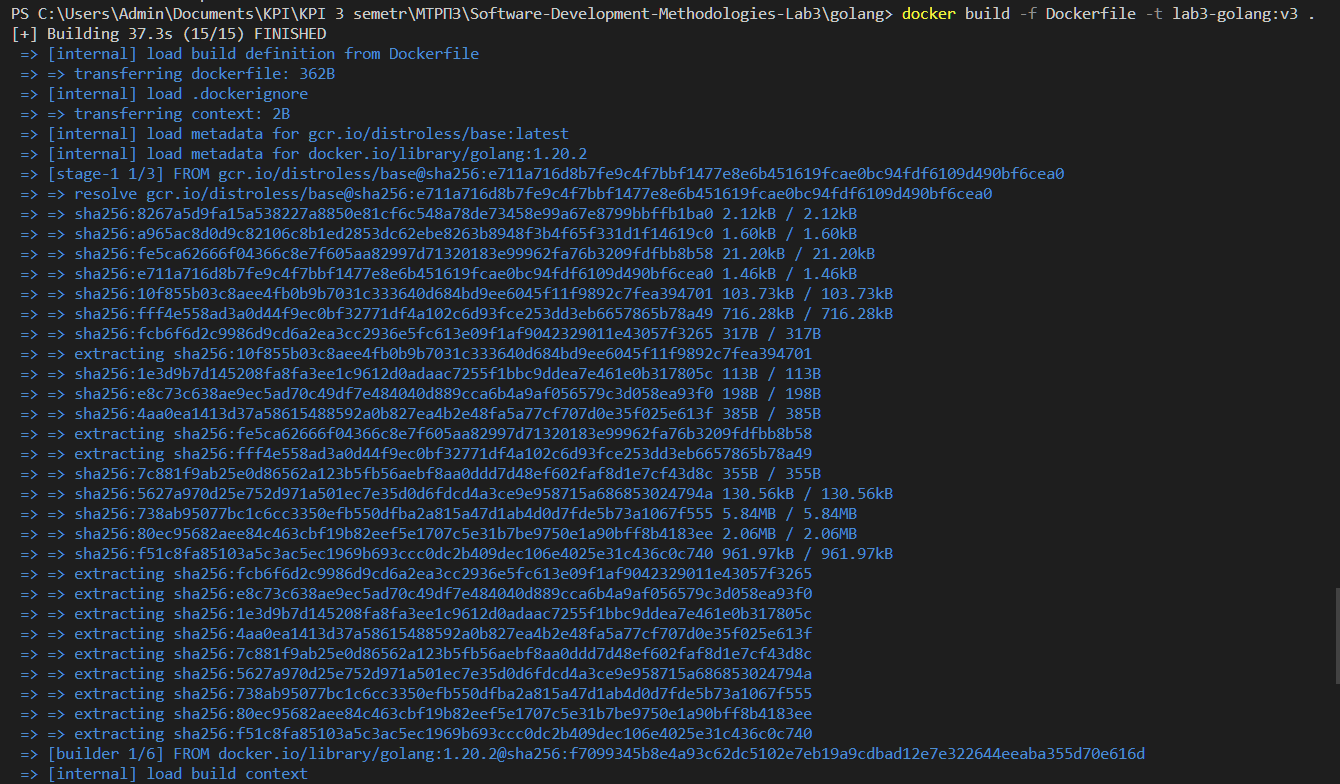
FROM gcr.io/distroless/base

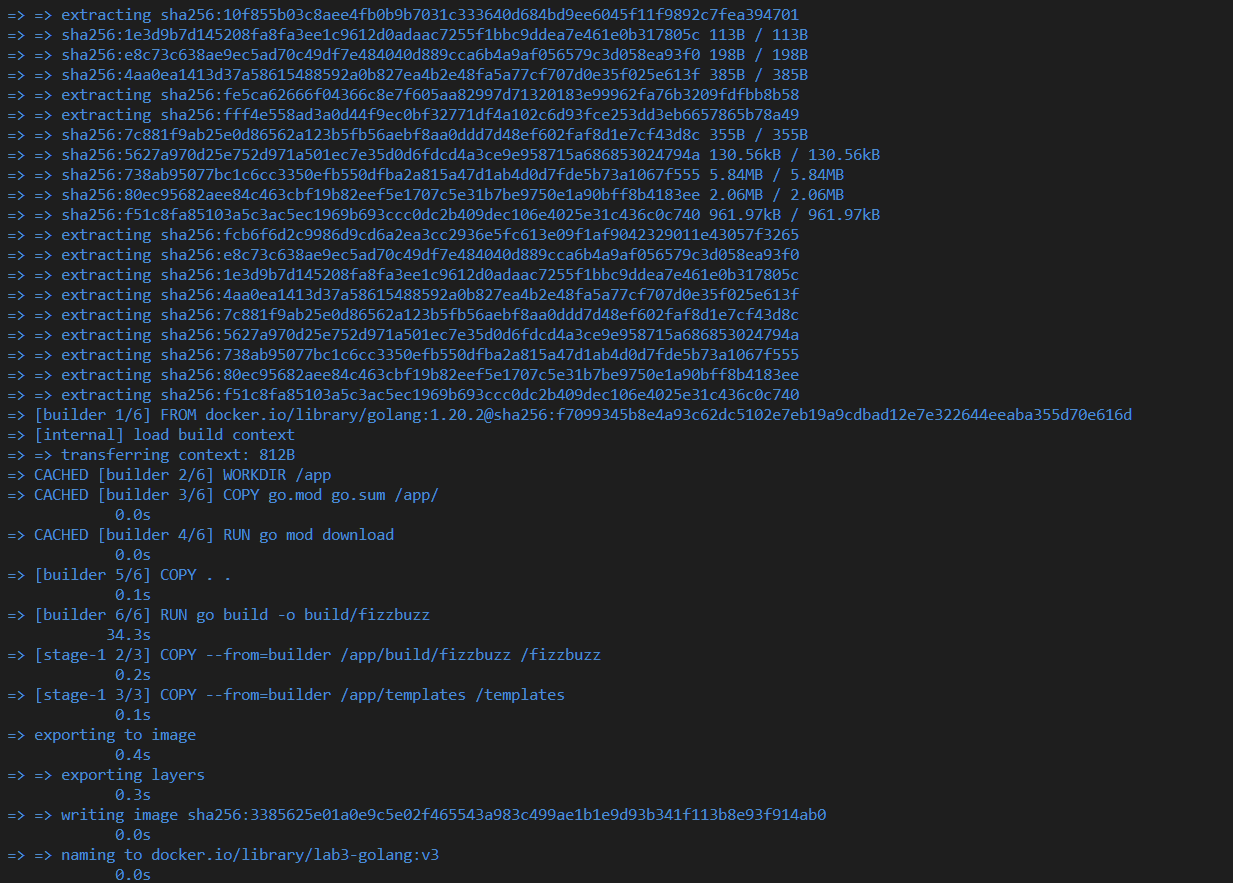
COPY --from=builder /app/build/fizzbuzz /fizzbuzz

COPY --from=builder /app/templates /templates

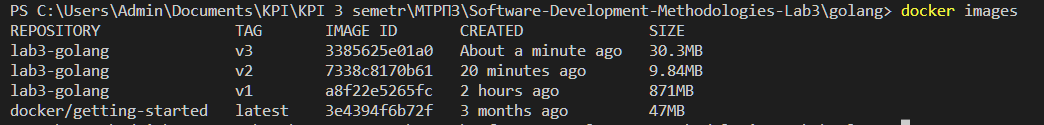
CMD ["/fizzbuzz", "serve", "--port=8080"]

Build command: **docker build -f Dockerfile -t lab3-golang:v2 .**

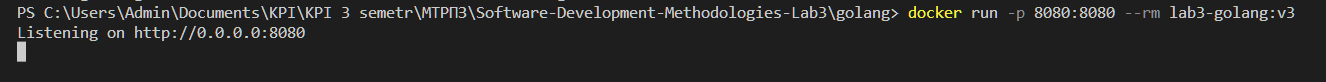


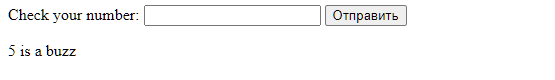


To see images: **docker images**



Run command: **docker run -p 8080:8080 --rm lab3-golang:v2**





Using a distroless image allows you to increase security and reduce the size of the Docker image, because it does not contain unnecessary OS components and dependencies that are necessary to build and run the application. Also, using a distroless image allows to get rid of the need to manually add all the necessary libraries and other components.

Build results:

* Image size: 30.3 MB
* Image build time 37.3 s

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/713a3ed8a0e41cd415216ad4905d9b6d0446e06b)

**Compare:**

Scratch is the most minimal base image available in Docker and contains only the bare minimum required to run a Docker container. It has no package manager or shell, and its size is just a few megabytes. Distroless, on the other hand, is also minimal, but it provides some functionality by including some necessary libraries and utilities.

Since Scratch has a minimal size, it has a very small attack surface, and it is less likely to have vulnerabilities. However, it requires manually including all the required dependencies, which can be a security risk if some dependencies are not included or are outdated. Distroless provides a higher level of security by including necessary libraries and utilities that are required to run the application.

**JavaScript**

1. Create an image of the project, add dependencies to the container, build the executable file

Dockerfile:

FROM node:18.15.0

WORKDIR /app

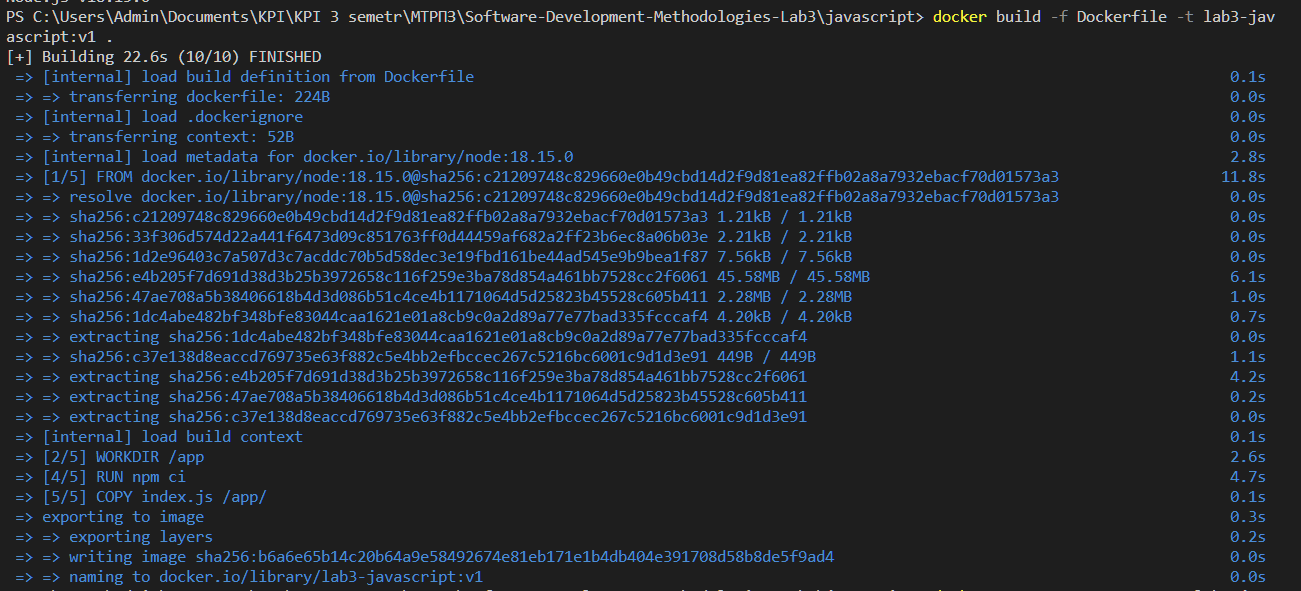
COPY package.json package-lock.json /app/

RUN npm ci

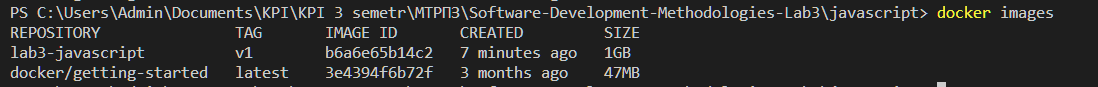
ENV NODE\_ENV=production

COPY index.js /app/

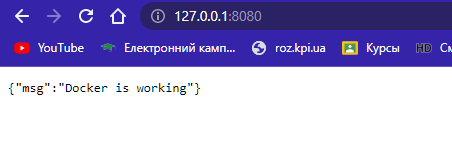
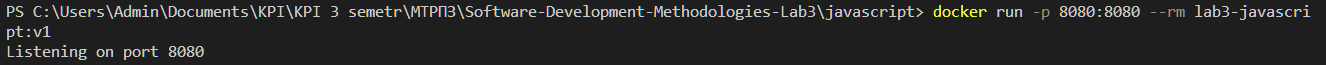
CMD ["node", "index.js"]

Build command: **docker build -f Dockerfile -t lab3-javascript:v1 .**

To see images: **docker images**



Run command**: docker run -p 8080:8080 --rm lab3-javascript:v1**



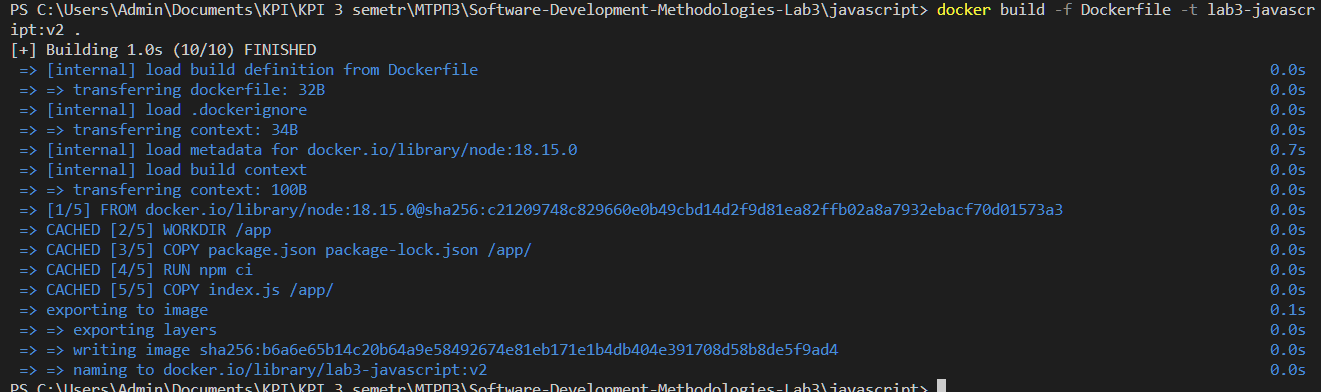
Build results:

* Image size: 1 GB
* Image build time 22.6 s

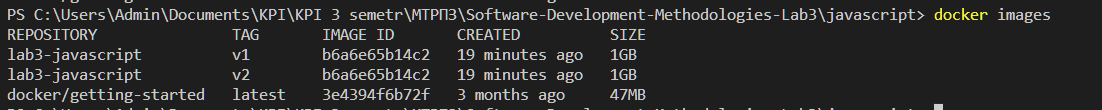
[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/3431b746696a9b1dc1928df88ce42e98e8dd21c3)

1. Build the project again to check the caching

Build command: **docker build -f Dockerfile -t lab3-javascript:v2**



To see images: **docker images**



Build results:

* Image size: 1 GB
* Image build time 1.0 s

1. Build the project using scratch

Dockerfile:

FROM node:18.15.0 as builder

WORKDIR /app

COPY package.json package-lock.json /app/

RUN npm ci

ENV NODE\_ENV=production

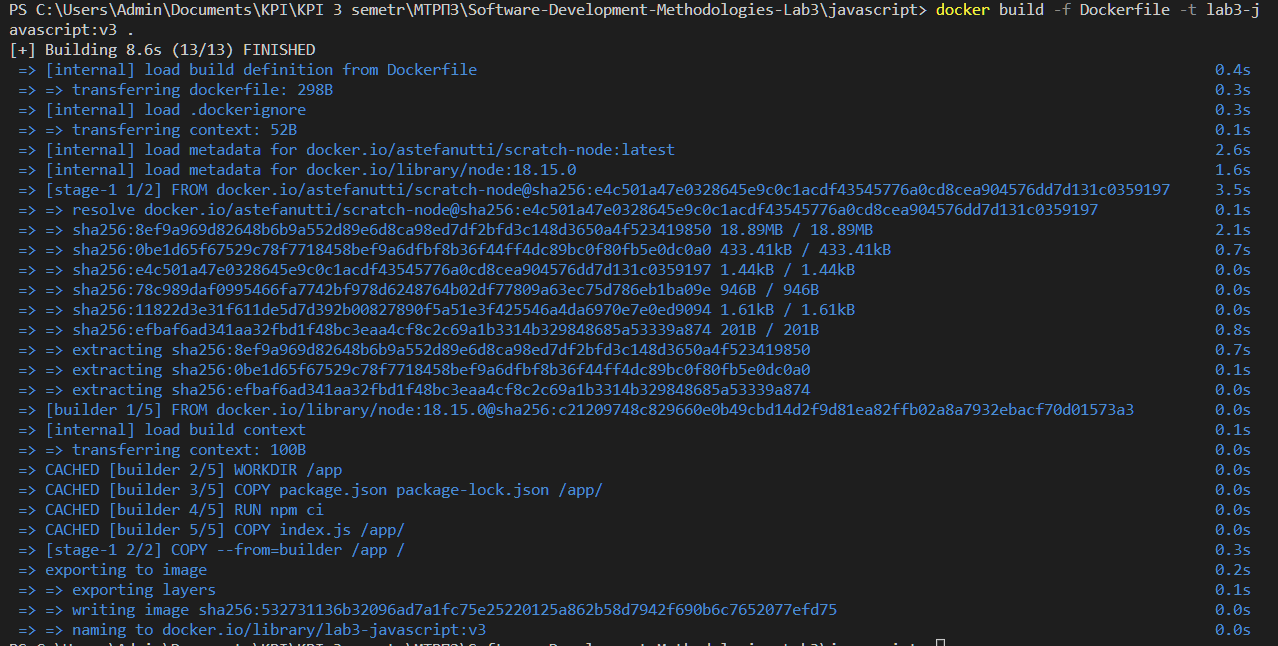
COPY index.js /app/

FROM astefanutti/scratch-node

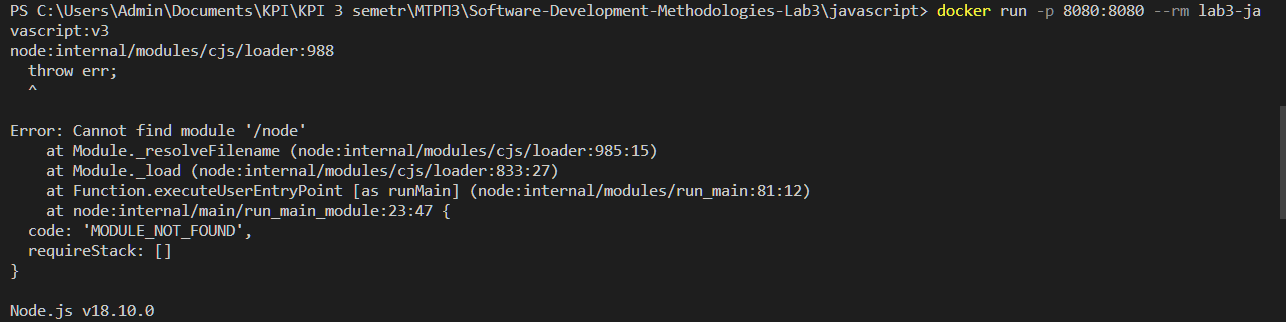
COPY --from=builder /app /

CMD ["node", "index.js", "--port=8080"]

Build command: **docker build -f Dockerfile -t lab3-javascript:v2**



Run command**: docker run -p 8080:8080 --rm lab3-javascript:v2**



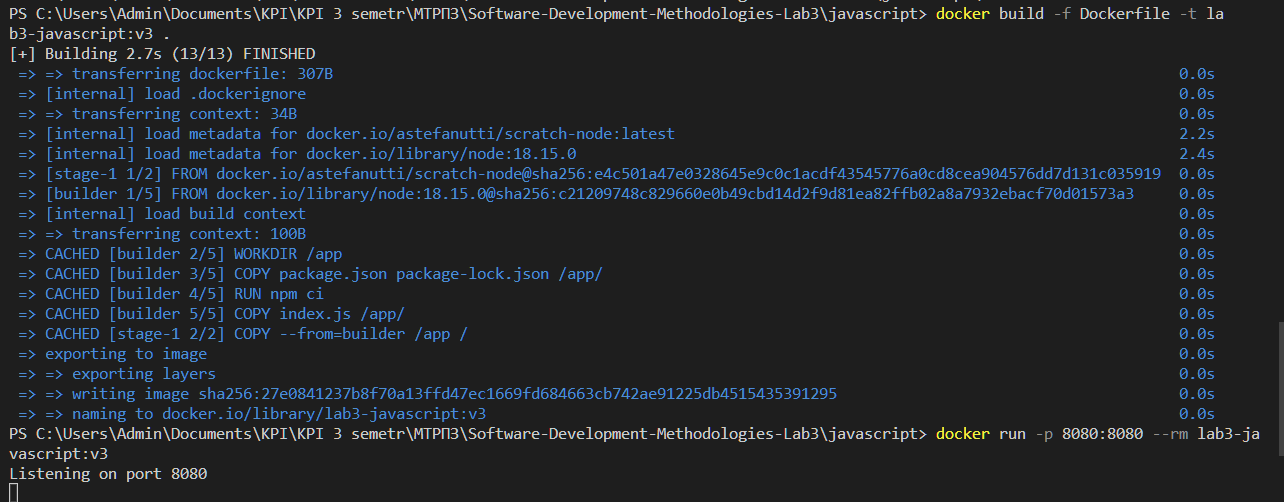
An error occurred during the starting of the image that can be fixed by adding line

ENTRYPOINT  ["node", "index.js", "--port=8080"]

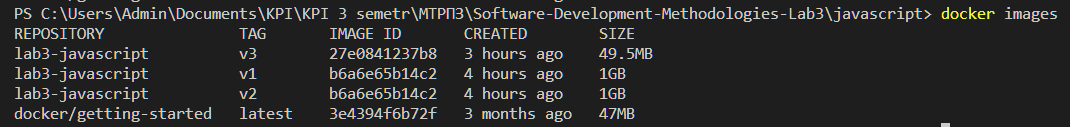
to Dockerfile instead of

CMD  ["node", "index.js", "--port=8080"]

Building one more and start:



To see images: **docker images**



Build results:

* Image size: 49.5 MB
* Image build time 8.6 s (first)
* Image build time 2.7 s (caching)

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/1ea7af96880d27d4c9ba94919af33d011f101a90)

1. Build the project using distroless

Dockerfile:

FROM node:latest as builder

WORKDIR /app

COPY package.json package-lock.json /app/

RUN npm ci

ENV NODE\_ENV=production

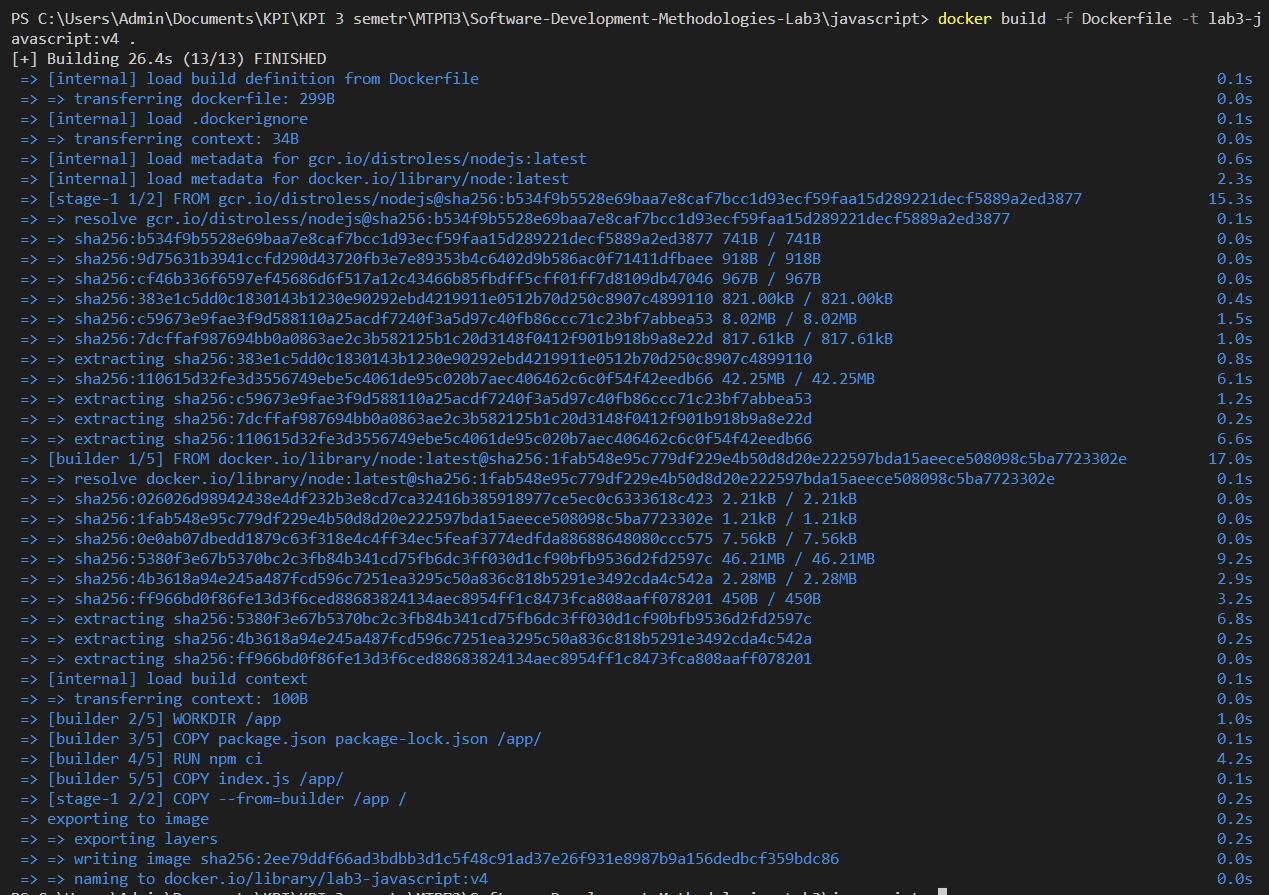
COPY index.js /app/

FROM gcr.io/distroless/nodejs

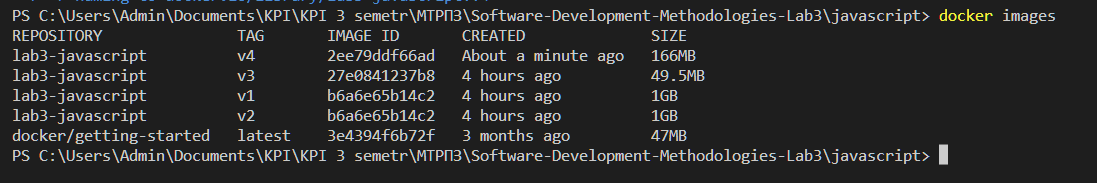
COPY --from=builder /app /

CMD  ["node", "index.js", "--port=8080"]

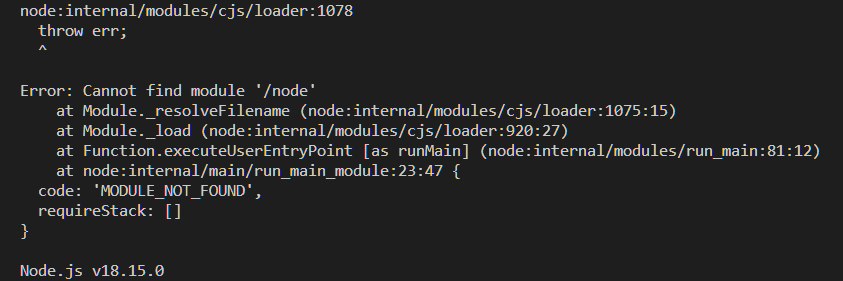
Build command: **docker build -f Dockerfile -t lab3-javascript:v4**



To see images: **docker images**



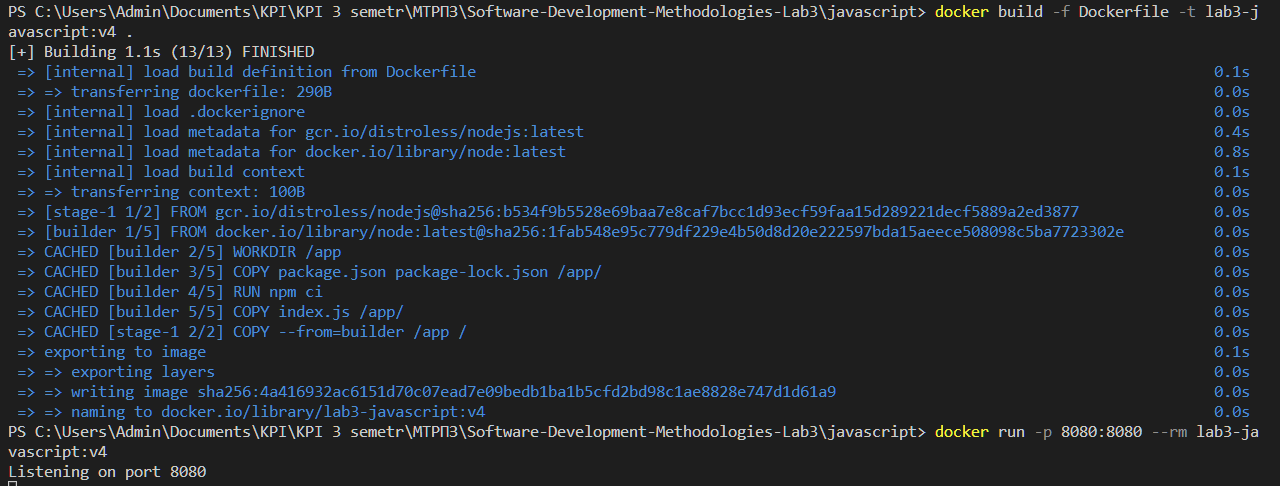
Run command**: docker run -p 8080:8080 --rm lab3-javascript:v2**



An error appeared again, to remove it, it is enough to change the last line and remove the node

CMD ["index.js", "--port=8080"]

Building one more and start:



Build results:

* Image size: 166 MB
* Image build time 26.4 s (first)
* Image build time 1.1 s (caching)

[Commit link](https://github.com/vladimirvikulin/Software-Development-Methodologies-Lab3/commit/b0f3660b4269b50359e83fb6de6b38b5f0477dc5)

The results obtained indicate that the image based on Scratch takes up less space than the image based on Distroless. However, to use the image based on Scratch, a statically linked executable file is required. Therefore, the choice depends on the specific situation and requirements of the program.

**Conclusion**

The laboratory work consisted of three parts, focusing on three different programming languages: Python, Golang, and Javascript. Each part aimed to explore the process of creating container images for applications, optimizing the image size and build time, and analyzing the image content.

In the Python part, we started by creating a container image for the application, measuring the image size and build time. Then we made changes to the code and re-built the image, measuring the size and build time again. After that, we optimized the Dockerfile by using layers and repeated the experiment. Finally, we compared the image size and build time between two different base images and added a dependency to the project, measuring the impact on the image size and build time.

In the Golang part, we followed a similar process, creating a container image for the project, measuring the build time and image size, and analyzing the image content. Then we experimented with multi-stage builds, where we separated the build process into two stages and analyzed the image content of the resulting image. Finally, we tried to create images using distroless, a more secure approach to container images, and analyzed their size and content.

In the Javascript part, we created a simple application and containerized it, measuring the image size and build time. We also experimented with different base images and analyzed their impact on the image size and build time.

Overall, the laboratory work provided a valuable hands-on experience in creating container images for applications and optimizing their size and build time. We also gained insights into the importance of choosing the right base image and the impact of dependencies on the image size and build time. Finally, we learned about the importance of analyzing the image content and using secure container image practices.