

Assignment 1, Web Application Development

Put all deliverables into github repository in your profile. Share link to google form 24 hours before defense. Defend by explaining deliverables and answering questions.

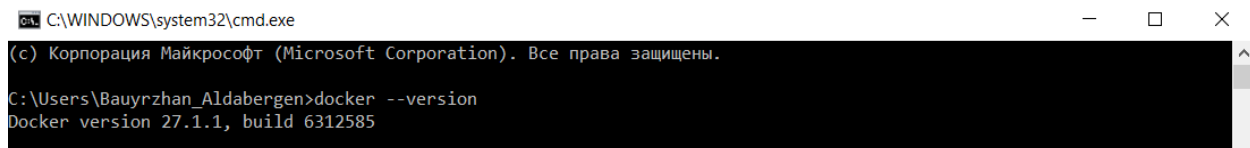
Deliverables: report in pdf

Google form: https://docs.google.com/forms/d/e/1FAIpQLSe0GyNdOYlvM1tX_I_CtlPod5jBf-ACLGdHYZq1gVZbUeBzlg/viewform?usp=sf_link

Intro to Containerization: Docker

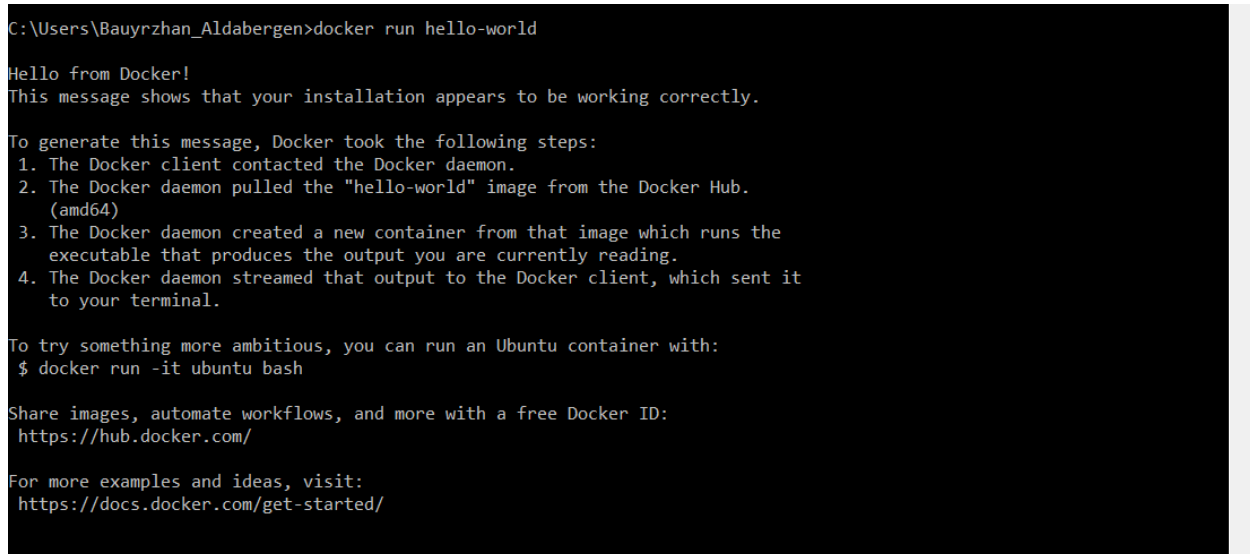
Exercise 1: Installing Docker

1. **Objective:** Install Docker on your local machine.
2. **Steps:**
 - Follow the installation guide for Docker from the official website, choosing the appropriate version for your operating system (Windows, macOS, or Linux).
 - After installation, verify that Docker is running by executing the command `docker --version` in your terminal or command prompt.



```
C:\WINDOWS\system32\cmd.exe
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.
C:\Users\Bauyrzhan_Aldabergen>docker --version
Docker version 27.1.1, build 6312585
```

- Run the command `docker run hello-world` to verify that Docker is set up correctly.



```
C:\Users\Bauyrzhan_Aldabergen>docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

3. **Questions:**
 - What are the key components of Docker (e.g., Docker Engine, Docker CLI)?
Docker Engine manages everything related to containers. It consists of:
 - Docker daemon. It is the background service that runs and manages containers.

- REST API that allows users and other applications to communicate with the Docker daemon.

Docker CLI is a command-line tool that lets you interact with Docker. It has run commands to create, start, stop, and manage containers and images

Docker Images. They contain everything needed to run an application, including the code, libraries, runtime environment.

Docker Containers. It is isolated environments where applications run; sharing the host system's kernel but keeping their processes separate.

Dockerfile. A script that contains instructions for building a Docker image.

Docker Compose is a tool for managing multi-container applications. Define all your services in a single YAML file, to start and configure everything at once.

Docker Hub a cloud service for sharing and storing Docker images. You can pull official images or upload your own to share with others.

Docker Network it allows containers to communicate with each other and with the outside world.

- How does Docker compare to traditional virtual machines?

If we compare by architecture, Docker uses containers that share the host OS kernel so it is lightweight and fast. VMs are running a full OS with its own kernel and they are heavier and slower.

Docker uses less memory and disk space on the other hand VMs use more memory due to full OS overhead.

Docker shares the OS, which can pose some security risks and VMs have stronger isolation and completely independent OS instances. Docker is quick to start and stop by performance, VMs are slower to boot due to the full OS loading. Docker is highly portable across environments and VMs can be moved, but are larger and less convenient. Docker used for microservices and fast deployment and VMs are better for legacy applications needing full OS features.

Docker is lightweight and efficient for modern applications, while VMs provide stronger isolation for specific use cases.

- What was the output of the `docker run hello-world` command, and what does it signify?

The output of the `docker run hello-world` command typically looks like this:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

Significance of the output

Installation confirmation. It shows that Docker is installed correctly and functioning as intended. It confirms that the Docker client can communicate with the Docker daemon.

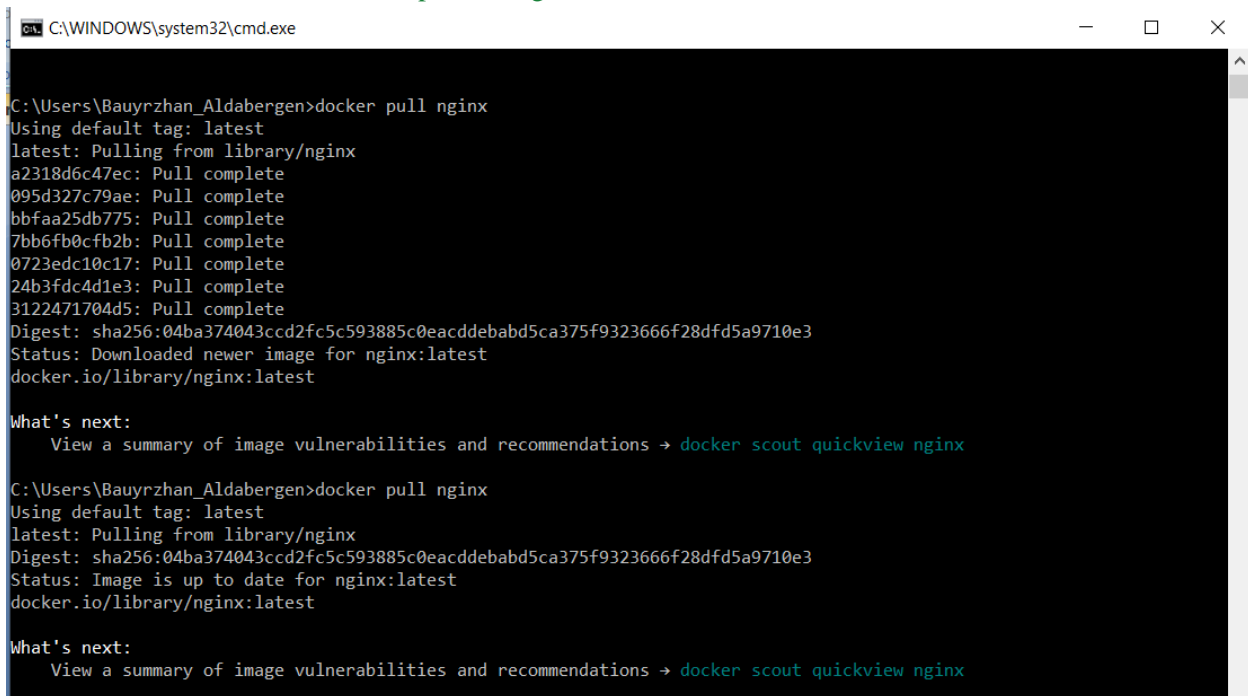
Image pulling. The output shows that Docker successfully pulled the hello-world image from Docker Hub.

Container execution. It shows that Docker can create and run containers from images, executing the application within the container.

Last steps. Provides guidance for exploring Docker documentation.

Exercise 2: Basic Docker Commands

1. **Objective:** Familiarize yourself with basic Docker commands.
2. **Steps:**
 - Pull an official Docker image from Docker Hub (e.g., `nginx` or `ubuntu`) using the command `docker pull <image-name>`.



```
C:\WINDOWS\system32\cmd.exe

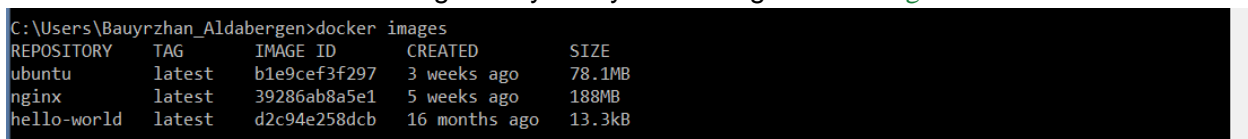
C:\Users\Bauyrzhan_Aldabergen>docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
a2318d6c47ec: Pull complete
095d327c79ae: Pull complete
bbfaa25db775: Pull complete
7bb6fb0cfb2b: Pull complete
0723edc10c17: Pull complete
24b3fdc4d1e3: Pull complete
3122471704d5: Pull complete
Digest: sha256:04ba374043ccd2fc5c593885c0eacddebabd5ca375f9323666f28dfd5a9710e3
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview nginx

C:\Users\Bauyrzhan_Aldabergen>docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
Digest: sha256:04ba374043ccd2fc5c593885c0eacddebabd5ca375f9323666f28dfd5a9710e3
Status: Image is up to date for nginx:latest
docker.io/library/nginx:latest

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview nginx
```

- List all Docker images on your system using `docker images`.



```
C:\Users\Bauyrzhan_Aldabergen>docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
ubuntu              latest          b1e9cef3f297   3 weeks ago    78.1MB
nginx               latest          39286ab8a5e1   5 weeks ago    188MB
hello-world         latest          d2c94e258dcf   16 months ago  13.3kB
```

- Run a container from the pulled image using `docker run -d <image-name>`.

```
C:\Users\Bauyrzhan_Aldabergen>docker run -d nginx
fbb60c94feced90be815a3cffff6836469099cdca1a1d68a4b4a46de9d199b16
C:\Users\Bauyrzhan_Aldabergen>
```

- List all running containers using `docker ps` and stop a container using `docker stop <container-id>`.

```
C:\Users\Bauyrzhan_Aldabergen>docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES
fbb60c94fece   nginx     "/docker-entrypoint..." About a minute ago Up About a minute 80/tcp       zealous_kare

C:\Users\Bauyrzhan_Aldabergen>docker stop fbb60c94fece
fbb60c94fece

C:\Users\Bauyrzhan_Aldabergen>docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES

C:\Users\Bauyrzhan_Aldabergen>
```

3. Questions:

- What is the difference between `docker pull` and `docker run`?
docker pull download a Docker image from a registry to your local machine. And *docker run* creates and starts a container from a image.
- How do you find the details of a running container, such as its ID and status?
Use *docker ps* for a quick overview of all running containers.
Use *docker inspect* for in-depth details about a specific container.
Use *docker stats* for real-time resource usage information.
- What happens to a container after it is stopped? Can it be restarted?

When a Docker container is stopped, it keeps all the changes and data that were made while it was running. And you can restart it whenever you want. If you decide you don't need it anymore, you can remove it completely.

Exercise 3: Working with Docker Containers

1. **Objective:** Learn how to manage Docker containers.

2. **Steps:**

- Start a new container from the `nginx` image and map port 8080 on your host to port 80 in the container using `docker run -d -p 8080:80 nginx`.

```
C:\Users\Bauyrzhan_Aldabergen>docker run -d -p 8080:80 nginx
ffaf6840e8d6b0b4455d2ede9ae287eb231e4f055ecc9e1ca6883c7e5a8b0c90
```

- Access the Nginx web server running in the container by navigating to `http://localhost:8080` in your web browser.



- Explore the container's file system by accessing its shell using `docker exec -it <container-id> /bin/bash`.

```
C:\Users\Bauyrzhan_Aldabergen>docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
ffaf6840e8d6   nginx    "/docker-entrypoint..." 29 minutes ago Up 29 minutes  0.0.0.0:8080->80/tcp    strange_clarke003e9f93e57e   nginx
" 30 minutes ago Up 30 minutes  80/tcp                inspiring_torvalds

C:\Users\Bauyrzhan_Aldabergen>docker exec -it ffaf6840e8d6 /bin/bash
root@ffaf6840e8d6:/#
```

- Stop and remove the container using `docker stop <container-id>` and `docker rm <container-id>`.

```
C:\Users\Bauyrzhan_Aldabergen>docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
ffaf6840e8d6   nginx    "/docker-entrypoint..." 35 minutes ago Up 35 minutes  0.0.0.0:8080->80/tcp    strange_clarke003e9f93e57e   nginx
003e9f93e57e   nginx    "/docker-entrypoint..." 36 minutes ago Up 36 minutes  80/tcp                inspiring_torvalds

C:\Users\Bauyrzhan_Aldabergen>docker stop ffaf6840e8d6
ffaf6840e8d6

C:\Users\Bauyrzhan_Aldabergen>docker rm ffaf6840e8d6
ffaf6840e8d6

C:\Users\Bauyrzhan_Aldabergen>docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
003e9f93e57e   nginx    "/docker-entrypoint..." 37 minutes ago Up 37 minutes  80/tcp                inspiring_torvalds

C:\Users\Bauyrzhan_Aldabergen>
```

3. Questions:

- How does port mapping work in Docker, and why is it important?
Port mapping in Docker is the process of connecting a container's internal ports to the host machine's ports, allowing you to access services running inside the container from outside. It lets you reach applications like web servers that are running inside containers. You can choose any available port on the host, so you don't have to worry about conflicts with other services. It makes it easier to test and interact with your applications during development.
- What is the purpose of the `docker exec` command?
The `docker exec` command is used to run commands inside a running Docker container. It's useful for troubleshooting, installing software, or checking logs while the container is active.

- How do you ensure that a stopped container does not consume system resources?

To ensure a stopped Docker container doesn't consume system resources we remove the Container freeing up resources. Prune Stopped Containers *docker container prune* to remove all stopped containers at once. Checking for any unused containers with *docker ps -a* and delete those you don't need. Clean up unused volumes with *docker volume prune* to avoid unnecessary space usage.

Dockerfile

Exercise 1: Creating a Simple Dockerfile

1. **Objective:** Write a Dockerfile to containerize a basic application.
2. **Steps:**
 - Create a new directory for your project and navigate into it.
 - Create a simple Python script (e.g., `app.py`) that prints "Hello, Docker!" to the console.
 - Write a Dockerfile that:
 - Uses the official Python image as the base image.
 - Copies `app.py` into the container.
 - Sets `app.py` as the entry point for the container.
 - Build the Docker image using `docker build -t hello-docker ..`
 - Run the container using `docker run hello-docker`.

```
C:\Users\Bauyrzhan_Aldabergen\hello-docker>docker build -t hello-docker .
[+] Building 3.9s (8/8) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile             0.0s
=> => transferring dockerfile: 121B                             0.0s
=> [internal] load metadata for docker.io/library/python:3.9-slim 3.4s
=> [internal] load .dockerignore                               0.0s
=> => transferring context: 2B                                    0.0s
=> [1/3] FROM docker.io/library/python:3.9-slim@sha256:2851c06da1fdc3c451784beef8aa31d1a313d8e3fc122e4a1891085a1 0.0s
=> [internal] load build context                               0.1s
=> => transferring context: 27B                                   0.0s
=> CACHED [2/3] WORKDIR /app                                   0.0s
=> CACHED [3/3] COPY app.py .                                  0.0s
=> exporting to image                                          0.0s
=> => exporting layers                                           0.0s
=> => writing image sha256:ce02e79d2aada8e10902694755e5274d31625bb8c97179330b774c5fcd3278ec 0.0s
=> => naming to docker.io/library/hello-docker                 0.0s

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview

C:\Users\Bauyrzhan_Aldabergen\hello-docker>docker run hello-docker
Hello, Docker!
```

3. Questions:

- What is the purpose of the **FROM** instruction in a Dockerfile?
- How does the **COPY** instruction work in Dockerfile?
- What is the difference between **CMD** and **ENTRYPOINT** in Dockerfile?

Exercise 2: Optimizing Dockerfile with Layers and Caching

1. **Objective:** Learn how to optimize a Dockerfile for smaller image sizes and faster builds.
2. **Steps:**
 - Modify the Dockerfile created in the previous exercise to:
 - Separate the installation of Python dependencies (if any) from the copying of application code.
 - Use a `.dockerignore` file to exclude unnecessary files from the image.
 - Rebuild the Docker image and observe the build process to understand how caching works.
 - Compare the size of the optimized image with the original.

```
C:\Users\Bauyrzhan_Aldabergen\hello-docker>docker build -t hello-docker .
[+] Building 2.3s (8/8) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile            0.0s
=> => transferring dockerfile: 318B                            0.0s
=> [internal] load metadata for docker.io/library/python:3.9  1.3s
=> [internal] load .dockerignore                               0.1s
=> => transferring context: 80B                                  0.0s
=> [internal] load build context                               0.1s
=> => transferring context: 27B                                   0.0s
=> [1/3] FROM docker.io/library/python:3.9@sha256:269252d0fb065c8e27ee77c1bade3fdded0c450ed3040c07c7d8256b592edf 0.0s
=> CACHED [2/3] WORKDIR /app                                   0.0s
=> [3/3] COPY app.py ./                                       0.1s
=> exporting to image                                         0.3s
=> => exporting layers                                          0.2s
=> => writing image sha256:7f0adec0ae56138328aa7e3a5bcf2f2ba2c59b477d88ea787a3b5d6b4123e7b7 0.0s
=> => naming to docker.io/library/hello-docker                 0.0s

What's next:
View a summary of image vulnerabilities and recommendations → docker scout quickview

C:\Users\Bauyrzhan_Aldabergen\hello-docker>docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-docker   latest    7f0adec0ae56   39 seconds ago 996MB
<none>        <none>    ce02e79d2aad   About an hour ago 125MB
<none>        <none>    317275782b35   About an hour ago 125MB
nginx         latest    39286ab8a5e1   5 weeks ago    188MB
hello-world    latest    d2c94e258dcb   16 months ago  13.3kB
```

3. Questions:

- What are Docker layers, and how do they affect image size and build times?

Docker layers are the building blocks of Docker images, where each layer represents changes made in the Dockerfile, such as installing software or adding files.

Effects on image size that layers are shared among images, so common layers don't take up extra space, keeping images smaller. Each layer only includes the changes from the previous one, minimizing redundancy. Effects for build times that Docker caches layers, allow it to reuse unchanged layers during builds, which speeds up the process. The order of commands in the Dockerfile is important; changing an early command can invalidate the cache for later layers, slowing down builds.

- How does Docker's build cache work, and how can it speed up the build process?

Docker's build cache helps speed up the image-building process by storing intermediate layers created during builds. Each command in a Dockerfile creates a layer that gets cached. If the same command is used again without changes, Docker can reuse the cached layer instead of rebuilding it. It Speeds Up Builds by reusing layers saving time and resources. Also by partial builds, if only later commands change, earlier layers can still be used, speeding up the build. And organizing your Dockerfile with less frequently changing commands at the top maximizes cache usage, further reducing build times.

- What is the role of the `.dockerignore` file?

It describes files and directories that you want to exclude when building a Docker image.

Exercise 3: Multi-Stage Builds

1. **Objective:** Use multi-stage builds to create leaner Docker images.
2. **Steps:**
 - Create a new project that involves compiling a simple Go application (e.g., a "Hello, World!" program).
 - Write a Dockerfile that uses multi-stage builds:
 - The first stage should use a Golang image to compile the application

```
C:\Users\Bauyrzhan_Aldabergen\hello-go>docker build -t hello-go .
[+] Building 10.5s (15/15) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile                0.0s
=> => transferring dockerfile: 294B                               0.0s
=> [internal] load metadata for docker.io/library/alpine:latest    2.2s
=> [internal] load metadata for docker.io/library/golang:1.23      2.3s
=> [auth] library/golang:pull token for registry-1.docker.io       0.0s
=> [auth] library/alpine:pull token for registry-1.docker.io      0.0s
=> [internal] load .dockerignore                                  0.1s
=> => transferring context: 2B                                     0.0s
=> [builder 1/4] FROM docker.io/library/golang:1.23@sha256:2fe82a3f3e006b4f2a316c6a21f62b66e1330ae211d039bb8d112 0.0s
=> [internal] load build context                                  0.1s
=> => transferring context: 119B                                   0.0s
=> [stage-1 1/3] FROM docker.io/library/alpine:latest@sha256:beefdbd8a1da6d2915566fde36db9db0b524eb737fc57cd1367 0.0s
=> CACHED [builder 2/4] WORKDIR /app                             0.0s
=> CACHED [builder 3/4] COPY . .                                  0.0s
=> [builder 4/4] RUN go build -o hello-go .                       7.6s
=> CACHED [stage-1 2/3] WORKDIR /app                             0.0s
=> CACHED [stage-1 3/3] COPY --from=builder /app/hello-go .      0.0s
=> exporting to image                                             0.1s
=> => exporting layers                                           0.0s
=> => writing image sha256:1ec37e0984543ac5c266a4fdf9f1293b1f414a299192d962e4008e87901b790c 0.0s
=> => naming to docker.io/library/hello-go                       0.0s

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview

C:\Users\Bauyrzhan_Aldabergen\hello-go>docker run --rm hello-go
Hello, World!
```

- The second stage should use a minimal base image (e.g., `alpine`) to run the compiled application.
- Build and run the Docker image, and compare the size of the final image with a single-stage build.


```
C:\Users\Bauyrzhan_Aldabergen\hello-go>docker build -t hello-go-single -f Dockerfile.single .
[+] Building 1.7s (9/9) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile.single      0.0s
=> => transferring dockerfile: 136B                             0.0s
=> [internal] load metadata for docker.io/library/golang:1.23  0.9s
=> [internal] load .dockerignore                               0.0s
=> => transferring context: 2B                                    0.0s
=> [1/4] FROM docker.io/library/golang:1.23@sha256:2fe82a3f3e006b4f2a316c6a21f62b66e1330ae211d039bb8d1128e12ed57 0.0s
=> [internal] load build context                                0.0s
=> => transferring context: 119B                                   0.0s
=> CACHED [2/4] WORKDIR /app                                    0.0s
=> CACHED [3/4] COPY . .                                        0.0s
=> CACHED [4/4] RUN go build -o hello-go .                      0.0s
=> exporting to image                                          0.4s
=> => exporting layers                                           0.3s
=> => writing image sha256:0e3627e6c87632c50b1c673980ee35f1e0990e5b2292e4ac4454aae62a55b90e 0.0s
=> => naming to docker.io/library/hello-go-single              0.0s

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview

C:\Users\Bauyrzhan_Aldabergen\hello-go>docker images
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
hello-go-single     latest      0e3627e6c876     About a minute ago 869MB
hello-go            latest      1ec37e098454     10 hours ago      9.93MB
```

3. Questions:

- What are the benefits of using multi-stage builds in Docker?

Multi-stage builds in Docker provide benefits like smaller image size, faster builds, cleaner dockerfiles, security that uses minimal base images, reducing the attack surface.

- How can multi-stage builds help reduce the size of Docker images?

Multi-stage builds reduce Docker image size by:

- Selective Copying. Only necessary files are copied to the final image.
- Lightweight Base Images. The final stage uses minimal base images, which decreases size.
- Fewer Layers. Avoids extra layers from build dependencies, keeping the image clean.
- Streamlined Dependencies. Installs all needed dependencies in the build stage without bloating the final image.

- What are some scenarios where multi-stage builds are particularly useful?

Multi-stage builds are particularly useful in:

- Complex Applications. They separate development dependencies from the final image.
- Heavy Tooling Languages. Languages like Go or Java benefit from compiling in one stage and running in a minimal environment.
- Microservices. Each service can have a lightweight image.
- CI/CD Pipelines. They speed up builds and reduce image size through layer caching.
- Security. They help minimize the attack surface by including only essential components.

Exercise 4: Pushing Docker Images to Docker Hub

1. **Objective:** Learn how to share Docker images by pushing them to Docker Hub.
2. **Steps:**
 - Create an account on Docker Hub.
 - Tag the Docker image you built earlier with your Docker Hub username (e.g., `docker tag hello-docker <your-username>/hello-docker`).

```
C:\Users\Bauyrzhan_Aldabergen\hello-docker>docker tag hello-docker baurzhana/hello-docker
```

- Log in to Docker Hub using `docker login`.
- Push the image to Docker Hub using `docker push <your-username>/hello-docker`.

```
C:\Users\Bauyrzhan_Aldabergen\hello-docker>docker login
Log in with your Docker ID or email address to push and pull images from Docker Hub. If you don't have a Docker ID, head
over to https://hub.docker.com/ to create one.
You can log in with your password or a Personal Access Token (PAT). Using a limited-scope PAT grants better security and
is required for organizations using SSO. Learn more at https://docs.docker.com/go/access-tokens/
```

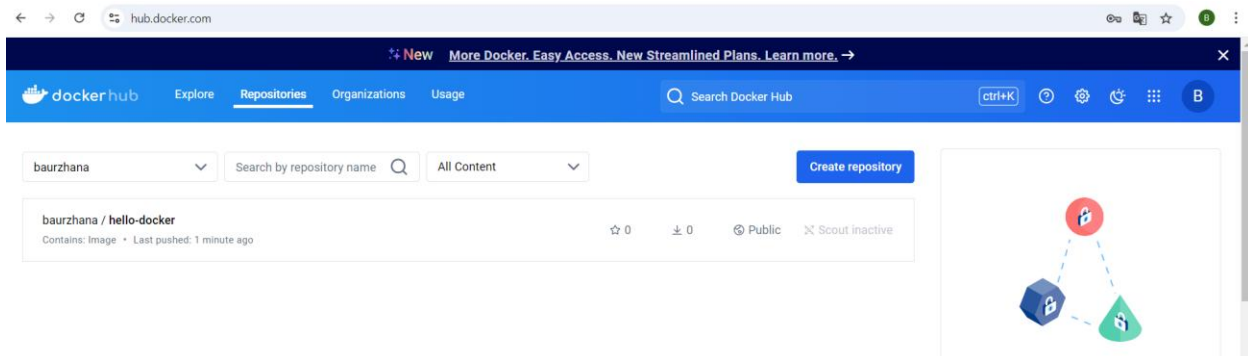
```
Username: baurzhana
Password:
```

```
Login Succeeded
```

```
C:\Users\Bauyrzhan_Aldabergen\hello-docker>docker push <your-username>/hello-docker
Не удается найти указанный файл.
```

```
C:\Users\Bauyrzhan_Aldabergen\hello-docker>docker push baurzhana/hello-docker
Using default tag: latest
The push refers to repository [docker.io/baurzhana/hello-docker]
1da0e52f706e: Pushed
aa91aa7271fb: Pushed
80afac98dfbd: Mounted from library/python
1c6c648b0363: Mounted from library/python
e97fd643c475: Mounted from library/python
3a8081ce85fa: Mounted from library/python
045d8b74bf0d: Mounted from library/python
25879f85bbb0: Mounted from library/python
6abe10f2f601: Mounted from library/python
latest: digest: sha256:0287180339233b17e53d94f26a5c671a9a91033b46e7d2f78af1ebab17003b29 size: 2209
```

- Verify that the image is available on Docker Hub and share it with others.



3. **Questions:**
 - What is the purpose of Docker Hub in containerization?

Docker Hub is a repository for Docker images that offers important features as

- Storage. Users can store and manage Docker images, both publicly and privately.
 - Sharing. Users can easily share their images with others.
 - Access to Official Images.
 - Version Control.
 - Automated Builds. Docker Hub can automatically build images from source code in GitHub or Bitbucket.
 - CI/CD Integration with CI/CD tools.
-
- How do you tag a Docker image for pushing to a remote repository?
Use the command: *docker tag <source-image> <your-username>/<repository-name>*
 - What steps are involved in pushing an image to Docker Hub?
 - 1.Tag the Image: *docker tag <source-image> <your-username>/<repository-name>*
 2. Log In: *docker login*
 - 3.Push the Image: *docker push <your-username>/<repository-name>*
 4. Verify: Check your Docker Hub profile for the image is there.