


Activity 6: Make Dockerfiles Production Ready

 **Objective:** Learn techniques to build smaller, faster, and more secure container images by applying production-ready Dockerfile patterns — using a compact, hands-on Go worked example and a short Node exercise.

Prerequisites

- Local **clab** workspace (you'll run `./system-clean-init.sh` from your local clab terminal).
 - An **Ubuntu 24.04 LTS EC2** instance with SSH access and Docker installed (Docker Engine + Docker Compose available).
 - `secret-key.pem` (SSH key) and the EC2 public IP address.
 - Basic familiarity with Docker CLI, `docker build`, `docker run`, and `tar` commands.
 - Recommended: Enable BuildKit when building (`DOCKER_BUILDKIT=1`) for cache & `--mount=type=cache` support.
-

Lab Setup

Goal: Reset the EC2 `docker_lab` workspace and automatically ship the **Activity6** source archive to the EC2 instance.

Important: From your **local clab terminal** (`/home/labDirectory`), simply run the provided script:

```
Shell
bash ./system-clean-init.sh
```

This script will:

1. **Clean/reset** your EC2 `docker_lab` workspace (remove old containers, images, volumes).

2. **Transfer and extract** the `Activity6_source.tar.xz` archive into the correct location on EC2.

After running it, SSH into your EC2 instance and verify:

```
Shell
ssh -i secret-key.pem ubuntu@<EC2_PUBLIC_IP>
cd /home/ubuntu/docker_lab/Activity6
```

The EC2 instance should look like this:

```
None
$ pwd
/home/ubuntu/docker_lab/Activity6

$ tree -a
.
├── go-app
│   ├── Dockerfile.naive
│   ├── README.md
│   ├── go.mod
│   └── main.go
└── node-exercise
    ├── Dockerfile.naive
    ├── README.md
    ├── app.js
    └── package.json

3 directories, 8 files
```

If the tree above matches, you are ready to proceed.

What's in the tarball

- `go-app/` – tiny Go web app + **naive** Dockerfile (`Dockerfile.naive`) and `README.md` with build/run/test instructions.

- `node-exercise/` — tiny Express app + **naive** Dockerfile (`Dockerfile.naive`) and `README.md` with build/run/test instructions.

Students will use these naive Dockerfiles as the starting point for optimization.




Tasks (overview)

This short activity contains two main items:

1. **Worked example (Go):** step-by-step build of a production-ready Dockerfile demonstrating:
 - Multi-stage builds
 - `FROM scratch` (and distroless) final stages
 - Base-image slimming (ubuntu vs alpine vs distroless)
 - Build cache optimization (`--mount=type=cache`)
 - Layer ordering & minimizing layers
 - Reducing attack surface (minimal deps + drop root)
 - Security scanning (`docker scan` / `trivy`)
 - Benchmarking & metrics (build time, image size, layer info, startup time)
 2. **Student exercise (Node):** given a naive `Dockerfile.naive`, the student must produce `Dockerfile.prod` that applies similar optimizations (multi-stage, cache, layer ordering, non-root, healthcheck) and verify by building and running the optimized image.
-



Task 1 — Worked example: Make the Go app production-ready (step-by-step)

 **Goal:** Starting from the naive `go-app/Dockerfile.naive`, we will apply one optimization at a time so you can see *why* each change helps. By the end you'll have a production-ready Dockerfile and a clear comparison (size, build speed, layers, startup) between the naive and optimized images.

GO app — general setup & optimization targets

Where you are (files / location):

```
None
/home/ubuntu/docker_lab/Activity6/go-app
├─ Dockerfile.naive
├─ README.md
├─ go.mod
└─ main.go
```

How the Go build works (brief):

- `go build` compiles your Go sources into a single binary.
- Go supports modules (`go.mod` / `go.sum`) — module downloads are a significant cost during builds.
- Static builds are possible (`CGO_ENABLED=0`) and desirable for `scratch` images because they don't depend on `libc`.
- The final artifact that needs to run is usually just a single binary — everything else (compiler, toolchain, caches) is only needed during build.

Why optimize the naive Dockerfile (targets):

1. **Image size** — naive images often contain the full Go toolchain (hundreds of MB). Shipping only the binary drastically reduces size.
 2. **Build speed (iterative)** — separate dependency download and use caching (`BuildKit --mount=type=cache`) so repeated builds are fast.
 3. **Layer caching** — ordering `COPY` to maximize cache re-use (copy `go.mod` first).
 4. **Attack surface** — final image should not include compilers or shells; run as non-root.
 5. **Debuggability vs minimality** — provide variants (`alpine` for debugging, `scratch`/distroless for production).
 6. **Reproducibility & metadata** — add labels, pin base images, strip binary symbols.
 7. **Health & observability** — include `HEALTHCHECK` and expose port in Dockerfile or compose.
-



Baseline: inspect the naive Dockerfile

`Dockerfile.naive` (baseline):

```
None
# Dockerfile.naive
FROM golang:1.20

WORKDIR /app
COPY . .
RUN go build -o server .

EXPOSE 8080
CMD ["/server"]
```

Baseline drawbacks:

- Final image contains full `golang:1.20` runtime (toolchain, package manager, shells) — large (~700–900MB depending on base) (Check `image size`).
- Layer cache is poor because `COPY . .` invalidates cache for dependency installation even if only unrelated **source files** changed (Check `image build time`).
- No stripping of binary; likely larger binary size.
- Runs as root by default.
- No BuildKit cache mounts used for modules.

Step 1 — Improve layer ordering & separate dependency download (for making docker build faster)

Goal: Copy `go.mod` or `go.sum` first, download modules (so this layer stay cached when only source changes), then copy the rest of the source.

Dockerfile snippet (`Dockerfile.step1`):

```
None
# step1: ordering to improve cache
FROM golang:1.20 AS builder
WORKDIR /src

# copy go.mod/go.sum first and download go dependencies
COPY go.mod ./
RUN go mod download

# copy the rest and build
COPY . .
RUN go build -o /server ./...
```

Build & verify:

```
Shell
docker build -t go-app:step1 -f Dockerfile.step1 .
# Change a source file and rebuild: dependency layer will remain cached
docker build -t go-app:step1 -f Dockerfile.step1 .
```

Why this helps: `go mod download` becomes cacheable. If you only change `main.go`, Docker can reuse the module-download layer and rebuild faster.

Step 2 — Use BuildKit Cache Mounts for Faster Builds (for faster iterative builds)

Goal: Leverage Docker's BuildKit to cache dependencies and build artifacts, making repeated builds much faster. Requires BuildKit enabled (`DOCKER_BUILDKIT=1`).

Dockerfile (`Dockerfile.step2`):

```
None
# syntax=docker/dockerfile:1.4
FROM golang:1.20 AS builder
WORKDIR /src

# cache Go module downloads and build cache
```

```
COPY go.mod ./
RUN --mount=type=cache,target=/go/pkg/mod \
    --mount=type=cache,target=/root/.cache/go-build \
    go mod download

COPY . .
RUN --mount=type=cache,target=/go/pkg/mod \
    --mount=type=cache,target=/root/.cache/go-build \
    go build -o /server ./...
```

The `# syntax=docker/dockerfile:1.4` line is crucial as it enables the advanced BuildKit features like cache mounts. The `--mount=type=cache` flag creates a temporary, isolated cache volume that persists between builds. This is a significant improvement over traditional caching methods, which rely on layers and are less granular.

How It Works

The `go mod download` and `go build` commands use these cache mounts to speed up the process:

- `--mount=type=cache,target=/go/pkg/mod`: This caches Go modules, so they only need to be downloaded once.
- `--mount=type=cache,target=/root/.cache/go-build`: This caches compiled objects and build artifacts.

Unlike the traditional Docker cache where a single change to `go.mod` invalidates the entire cache layer, **BuildKit's cache mounts are more intelligent and granular**, only rebuilding what's absolutely necessary. This results in more efficient and reproducible builds.

Build Commands

```
Shell
# First build (downloads modules and populates the cache)
time DOCKER_BUILDKIT=1 docker build -t go-app:step2 -f Dockerfile.step2 .
```

```
# Subsequent builds (much faster, as the cache is used)
time DOCKER_BUILDKIT=1 docker build -t go-app:step2 -f Dockerfile.step2 .
```

Step 3 — Make the Binary Smaller: Static Build & Strip Symbols (Size Compression)

Goal: Reduce the final binary size by creating a statically linked executable and removing unnecessary metadata.

Dockerfile (**Dockerfile.step3**):

```
None
# syntax=docker/dockerfile:1.4
FROM golang:1.20 AS builder
WORKDIR /src

COPY go.mod ./
RUN --mount=type=cache,target=/go/pkg/mod \
    go mod download

COPY . .

# build static, stripped binary
ENV CGO_ENABLED=0
RUN --mount=type=cache,target=/go/pkg/mod \
    go build -trimpath -ldflags="-s -w" -o /server ./...
```

This process creates a self-contained, lightweight binary that is easier to deploy and has a smaller attack surface.

How It Works

This build process uses two key flags to reduce the binary size:

- **ENV CGO_ENABLED=0**: This creates a **statically linked** binary. It tells the Go compiler not to use the host C library (libc) but instead to package all necessary C code directly into the final executable. The resulting binary is self-sufficient and does not require any external dependencies.
- **-ldflags="-s -w"**: These linker flags remove metadata from the binary.
 - **-s**: This flag strips the **symbol table**. The symbol table contains information used by debuggers to link addresses back to function names, which is unnecessary for a production binary.
 - **-w**: This flag strips the **DWARF debug information**, further reducing the size by removing data used for stack traces and debugging.
- **-trimpath** : This Go linker flag removes all file system paths from the compiled binary.
 - Without -trimpath, the final binary contains absolute file paths from the build machine.
 - This is significant because it makes your Go builds reproducible and more secure across different machines while using **statically linked binary**.

Build & Check Binary Size (Optional)

```
Shell
DOCKER_BUILDKIT=1 docker build --target=builder -t go-app:builder -f
Dockerfile.step3 .
# Inspect builder container filesystem
docker run --rm --entrypoint ls go-app:step2 -l /server || true
# vs
docker run --rm --entrypoint ls go-app:builder -l /server || true

# Or extract binary from image to check size:
docker create --name tmp go-app:builder
docker cp tmp:/server ./server_tmp
docker rm tmp
ls -lh server_tmp
```

Why this helps: A statically linked and stripped Go binary can be extremely small—often just a few megabytes—as it no longer relies on system libraries like **libc** and has all non-essential debugging information removed. The **--target=builder** flag is used to stop the build at this stage, creating a temporary image that contains only the builder

environment and the final binary. This allows you to inspect the binary's size before proceeding to the final, minimal image.

***Note:** The `docker create` command is used to create a new Docker container from a specified image, without starting it. This differs from `docker run`, which creates the container and immediately starts it.*

Step 4 — Multi-stage Builds for Minimal Images (Size Compression)

Goal: This step uses a **multi-stage build** to create a final, production-ready image that contains only the necessary application binary, stripping away all build dependencies.

Dockerfile (`Dockerfile.step4`):

```
None
# syntax=docker/dockerfile:1.4
FROM golang:1.20 AS builder
WORKDIR /src
COPY go.mod ./
RUN --mount=type=cache,target=/go/pkg/mod go mod download
COPY . .
ENV CGO_ENABLED=0
RUN --mount=type=cache,target=/go/pkg/mod \
    go build -trimpath -ldflags="-s -w" -o /server ./...

# Final stage: scratch
FROM scratch AS runtime
COPY --from=builder /server /server
```

Alternative final stage (`Dockerfile.step4_distroless`):

```
None
# syntax=docker/dockerfile:1.4
FROM golang:1.20 AS builder
WORKDIR /src
COPY go.mod ./
RUN --mount=type=cache,target=/go/pkg/mod go mod download
COPY . .
```

```
ENV CGO_ENABLED=0
RUN --mount=type=cache,target=/go/pkg/mod \
    go build -trimpath -ldflags="-s -w" -o /server ./...

# Final stage: distroless
FROM gcr.io/distroless/static
COPY --from=builder /server /server
```

How It Works

A multi-stage build uses multiple **FROM** instructions in a single **Dockerfile**. Each **FROM** starts a new stage, and you can selectively copy artifacts from a previous stage without bringing along its entire filesystem.

- **AS <stagename>**: This keyword assigns a name to a build stage (e.g., **FROM golang:1.20 AS builder**). This allows you to reference the stage later.
- **--from=<stagename>**: This flag, used with the **COPY** instruction, specifies that you want to copy files from a named stage instead of the current one. Here, **COPY --from=builder /server /server** copies only the compiled **/server** binary from the **builder** stage into the new **runtime** stage.
- **--target=<stagename>**: This flag is a build-time option (**docker build --target=builder**) that allows you to stop the build at a specific stage, which is useful for debugging or creating intermediate images.

Minimal Base Images: **scratch** vs. **distroless**

The final stage uses an extremely small base image, either **scratch** or **distroless**, to minimize the final image size and reduce the attack surface.

- **scratch**: This is the smallest possible base image, an **empty image**. It contains no files, no directories, no operating system, and no shell. It's ideal for statically-linked binaries like the one we've built, as it requires no external dependencies.
- **distroless**: An alternative to **scratch** from Google. While slightly larger than **scratch**, it contains a minimal set of system libraries (**glibc**, **ssl**) and is

useful for dynamically-linked binaries that might need these libraries. It provides a tiny, secure runtime environment with no package managers or shells, making it safer than traditional base images.

Build Commands

Shell

```
# Build with the scratch final stage
```

```
DOCKER_BUILDKIT=1 docker build -t go-app:prod-scratch -f Dockerfile.step4 .
```

```
# Build with the distroless final stage
```

```
DOCKER_BUILDKIT=1 docker build -t go-app:prod-distroless -f  
Dockerfile.step4_distroless .
```

Why this helps: The final image contains only your application binary and its required runtime dependencies, removing the large build environment (compiler, development tools, package manager, and shell). This drastically reduces the image size, improves security by eliminating unnecessary software, and makes the image faster to push and pull.

Step 5 — Add Metadata, Healthcheck, and a Non-Root User

Goal: This step focuses on adding crucial operational metadata and security configurations to the final production image.

Complete Dockerfile (**Dockerfile.semifinal**):

None

```
# syntax=docker/dockerfile:1.4
```

```
FROM golang:1.20 AS builder
```

```
WORKDIR /src
```

```
COPY go.mod ./
```

```
RUN --mount=type=cache,target=/go/pkg/mod \  
go mod download
```

```

COPY . .
ENV CGO_ENABLED=0
RUN --mount=type=cache,target=/go/pkg/mod \
    go build -trimpath -ldflags="-s -w" -o /server ./...

# -----
# Final stage (scratch) with metadata and healthcheck
FROM scratch AS runtime

LABEL org.opencontainers.image.title="go-app" \
    org.opencontainers.image.version="v1.0.0" \
    org.opencontainers.image.description="Tiny Go service"

COPY --from=builder /server /server

# numeric non-root UID
USER 10001

EXPOSE 8080
HEALTHCHECK --interval=10s --timeout=2s --start-period=5s CMD
["/server", "--healthcheck"]

# Add ENTRYPOINT to run the server binary
ENTRYPOINT ["/server"]

```

How It Works

These final additions enhance the image's security and operability:

- **LABEL:** This adds **Open Container Initiative (OCI) labels**, providing standardized metadata like title, version, and description. This helps with image discoverability and management.
- **USER 10001:** This sets the container to run as a **non-root user**, a key security practice. Using a numeric UID like **10001** is standard for **scratch** images since they contain no `/etc/passwd` file.
- **EXPOSE 8080:** This declares that the container listens on port **8080**. While it doesn't publish the port to the host, it serves as documentation and is used by other Docker features.

- **HEALTHCHECK:** This defines a command to check the container's health. For a `scratch` image, you cannot use a shell or tools like `curl`, so the health check must be a **statically compiled binary that returns a non-zero exit code on failure**. Our example assumes the `/server` binary has a `--healthcheck` flag for this.

Build and Run the Final Image

With the complete `Dockerfile` saved as `Dockerfile.semifinal`, you can now build the image and run the container.

```
Shell
# Build the final image with a tag
DOCKER_BUILDKIT=1 docker build -t go-app:semifinal -f Dockerfile.semifinal .

# Run the container, mapping the port
docker run -d --name go-server -p 8080:8080 go-app:semifinal

# Check the container's health status
docker inspect --format='{{.State.Health.Status}}' go-server
```

The `docker run` command launches the container, publishing the internal port `8080` to the host's `8080`. The `docker inspect` command then checks the container's health status, which Docker determined by running the `HEALTHCHECK` command you defined in the `Dockerfile`.

Step 6 — Final: combine all steps into `Dockerfile.prod`

Now assemble the previous progressive steps into the final `Dockerfile.prod`. (You built it incrementally above; this file is the final product that glues everything together.)

```
None
# Dockerfile.prod (final)
# syntax=docker/dockerfile:1.4
#####
# Builder
```

```
#####
FROM golang:1.20 AS builder
WORKDIR /src

# Copy go.mod first to leverage cache
COPY go.mod ./

# Use BuildKit cache for module downloads and go build cache
RUN --mount=type=cache,target=/go/pkg/mod \
    --mount=type=cache,target=/root/.cache/go-build \
    go mod download

# Copy source and build static, stripped binary
COPY . .
ENV CGO_ENABLED=0
RUN --mount=type=cache,target=/go/pkg/mod \
    --mount=type=cache,target=/root/.cache/go-build \
    go build -trimpath -ldflags="-s -w" -o /server ./...

#####
# Runtime (scratch)
#####
FROM scratch AS runtime

LABEL org.opencontainers.image.title="go-app" \
    org.opencontainers.image.version="v1.0.0" \
    org.opencontainers.image.licenses="MIT"

COPY --from=builder /server /server

# Run as non-root numeric UID
USER 10001

EXPOSE 8080
HEALTHCHECK --interval=10s --timeout=2s --start-period=5s CMD
["/server", "--healthcheck"]

ENTRYPOINT ["/server"]
```

Note: If your binary does not accept `--healthcheck`, modify the `HEALTHCHECK` to call the binary appropriately or use a minimal `alpine` final stage with `curl` to `/health`.

✓ How to build & measure (commands you will run on EC2)

Build the baseline (naive) image:

```
Shell
# baseline
time docker build -t go-app:naive -f Dockerfile.naive .
```

Build the final optimized image (use BuildKit):

```
Shell
# optimized (first build may download modules)
DOCKER_BUILDKIT=1 docker build -t go-app:prod -f Dockerfile.prod .
# second build (should be faster due to cached mounts)
time DOCKER_BUILDKIT=1 docker build -t go-app:prod -f Dockerfile.prod .
```

Measure image size and layers:

```
Shell
# sizes
docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"
go-app:naive

docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}" go-app:prod

# layers (count)
docker history go-app:naive --no-trunc --format '{{.ID}} {{.Size}}' | wc -l
docker history go-app:prod --no-trunc --format '{{.ID}} {{.Size}}' | wc -l
```

Startup / healthcheck / non-root verification:

```
Shell
# run baseline and optimized on different ports
docker run -d --name go-naive -p 8085:8080 go-app:naive
docker run -d --name go-prod -p 8086:8080 go-app:prod

# wait a few seconds then check
```



```
curl -fsS http://localhost:8085/health || echo "naive no response"
curl -fsS http://localhost:8086/health || echo "prod no response"

# verify non-root (inspect image config)
docker image inspect go-app:prod --format='User={{.Config.User}}'

# for running container (if it has shell): docker exec -it go-prod id -u
```

Security scanning (Trivy):

```
Shell

# run trivy (containerized) and capture output
docker run --rm -v /var/run/docker.sock:/var/run/docker.sock
aquasec/trivy:latest image go-app:prod > trivy_goapp_prod.txt

docker run --rm -v /var/run/docker.sock:/var/run/docker.sock
aquasec/trivy:latest image --skip-db-update go-app:naive >
trivy_goapp_naive.txt
```

✓ Testing and Verification

Testing an optimized Docker image involves more than just ensuring it runs. You must verify that the optimizations, such as a reduced size and enhanced security, have been successfully applied. The process involves a **comparison test** between a **naive image** (built without any optimizations) and the **final optimized image**. This difference is the proof that your work has paid off.

- **Size & Layer Measurement:** The `docker images` and `docker history` commands allow you to directly compare the size and number of layers. The naive image will be significantly larger and have more layers due to its large `golang` base and a history of build commands, while the optimized image will be tiny and have only one or two layers.
- **Startup & Healthcheck:** Running the two containers side-by-side verifies that both are functional. The `curl` command confirms the application's availability, and `docker inspect` validates that the `HEALTHCHECK` from the `Dockerfile.prod` is correctly configured and passing.

- **Non-Root Verification:** The `USER` instruction is a critical security feature. `docker image inspect` is used to confirm that the `Config.User` is set to `10001`, proving that the container will run with reduced privileges. This is a simple but vital security check.

Security Scanning with Trivy

Trivy is a popular, open-source **container security scanner**. It analyzes container images for vulnerabilities, misconfigurations, and other security issues. The `docker run` command provided runs Trivy in a container, mounts the host's Docker socket, and then scans your `go-app:prod` image.

- **Why it's useful:** A tiny `scratch`-based image is inherently secure because it contains no operating system, shell, or package manager. This means Trivy will find almost zero vulnerabilities, as there are no packages with known CVEs to report. Scanning the naive image, however, would likely find hundreds of vulnerabilities from the `golang:1.20` base image, demonstrating the massive security advantage of your optimized build. The difference in scan results is a powerful testament to the value of a multi-stage, static binary build.

Comparison: naive vs optimized — what to expect (how to interpret results)

Run the measurement commands above and compare. Below is an **example** comparison (sample numbers — your EC2 / network environment may differ). Use these as guidance for expected improvements.

Aspect	Baseline (<code>Dockerfile.naive</code>) — example	Optimized (<code>Dockerfile.prod</code>) — example	Why optimized is better
Image size (human)	850 MB (golang base)	6 MB (scratch with static binary)	The final image contains only the stripped, statically linked binary, completely removing the heavy Go compiler and operating system toolchain.
Build time — 1st build	28 s	32 s	The optimized build may be slightly slower on the first run due to the extra build stage, but overall is similar.
Build time — 2nd build	25 s	2–5 s	The BuildKit cache mounts make subsequent builds dramatically faster by reusing downloaded modules and compiled build artifacts.

Aspect	Baseline (<code>Dockerfile.naive</code>) – example	Optimized (<code>Dockerfile.prod</code>) – example	Why optimized is better
Layer count	10+ layers	4–6 layers	The multi-stage build discards all intermediate layers from the heavy <code>builder</code> stage, resulting in a significantly slimmer final image with fewer layers.
Startup time	~0.2 s	~0.05–0.2 s	A smaller binary with no OS overhead can lead to a slightly faster startup time, reducing latency.
Attack surface	High (toolchain + shell)	Minimal (only binary)	A smaller image means a drastically reduced number of installed packages, which translates to fewer potential CVE sources and a more secure image.
Debuggability	Easy (shell available)	Harder (scratch no shell)	The <code>scratch</code> base image has no shell or debugging tools. An <code>alpine</code> variant provides a trade-off, offering a minimal shell for debugging.
Trivy findings (where present)	Many base-image CVEs	Very few in runtime image	The <code>golang</code> base image carries hundreds of vulnerabilities from its operating system. Your final image has almost none, as it contains only your trusted binary.

Important: The first-build time for the optimized Dockerfile can sometimes be slightly longer because the `builder` stage has to download modules and compile the binary. However, the **second (and subsequent) builds are dramatically faster** with BuildKit cache mounts. The image size reductions are typically enormous when moving from a full `golang` final image to a `scratch/distroless` final image.

Cleanup (after comparing)

```
Shell
docker rm -f go-prod go-naive go-server 2>/dev/null || true


docker rmi go-app:prod go-app:naive go-app:semifinal go-app:prod-distroless
go-app:prod-scratch go-app:builder go-app:step1 go-app:step2 aquasec/trivy
2>/dev/null || true

docker system prune

rm -f trivy_goapp_prod.txt trivy_goapp_naive.txt || true

find . -maxdepth 1 -type f -name 'Dockerfile*' -not -name 'Dockerfile.naive'
-not -name 'Dockerfile.prod' -delete
```

Task 2 — Student exercise : Optimize the Node app Dockerfile

 **Goal:** Starting from the provided `node-exercise/Dockerfile.naive`, write an optimized `Dockerfile.prod` that applies the same classes of optimizations we used for the Go worked example.

Node app — what is provided

Work in the EC2 path:

```
None
/home/ubuntu/docker_lab/Activity6/node-exercise
├─ Dockerfile.naive      # provided (unoptimized)
├─ README.md             # provided (how to run naive image)
├─ app.js                # tiny Express app (GET / and GET /health)
└─ package.json
```

About the Node app: A minimal Express application exposing two endpoints:

- `GET /health` → returns `ok` (used for health checks)
- `GET /` → returns a simple greeting

The app has no build step (plain JavaScript) and uses only a few dependencies listed in `package.json`. Your task is to produce a production-ready Dockerfile for this app (named `Dockerfile.prod`) that is smaller, faster to iterate on, and safer to run.

Baseline (already available)

`Dockerfile.naive`:

```
None
# Dockerfile.naive
FROM node:18
```

```
WORKDIR /app
COPY . .
RUN npm install

EXPOSE 8080
CMD ["npm", "start"]
```

This builds and runs, but leaves many optimization opportunities.

* Where you should focus — *conceptual hints*

Below are **hints** pointing to the areas you should optimize.

- **Dependency isolation & cacheability**
 - Think about which files change frequently and which do not. Arrange steps so dependency installation is cached when only source code changes.
- **Reproducible installs**
 - Consider the difference between a simple `install` and a reproducible install. Which artifact ensures the same install every time?
- **Build vs runtime separation**
 - Can you separate the installation/build of dependencies from the minimal runtime image so the final image contains only what it needs to run?
- **Final image minimality**
 - The runtime image should be as small as possible while still running Node. What base image spectrum exists (full → slim → musl-based → distroless), and what tradeoffs do they bring?
- **Cache reuse across builds**
 - How can you make repeated local edits rebuild very fast? (Think caching mechanisms that persist dependency caches across builds.)
- **Least privilege execution**

- How will you ensure the container process does not run as root at runtime?
- **Health probing in a minimal runtime**
 - If you choose a minimal runtime that lacks shell tools, how will the health probe be implemented?
- **Layer ordering & file copies**
 - Which files should be copied early to maximize cache reuse, and which should be copied later to avoid invalidating dependency layers?
- **Production vs development dependencies**
 - How do you keep dev-only packages out of the final runtime image?
- **Package manager cache location**
 - Where does the package manager cache live (inside the build), and how could reusing it speed builds?

Use these hints to design your `Dockerfile.prod`. The point of this exercise is that you decide the tradeoffs and implement them.

⚙ Minimal `Dockerfile.prod` scaffold (create this file and fill it in)

Create the file

`/home/ubuntu/docker_lab/Activity6/node-exercise/Dockerfile.prod` on EC2. Keep the file mostly empty as a scaffold — include only an initial header and comments describing where to add stages. Example scaffold (paste into `Dockerfile.prod` and implement below it):

```
None
# syntax=docker/dockerfile:1.4
# Dockerfile.prod (student to implement)
# --- Add a builder stage here (if needed) ---
# --- Add a minimal runtime stage here ---
# --- Ensure HEALTHCHECK, non-root USER, and pinned runtime base are handled ---
# End of scaffold
```

Important: keep your final file named exactly `Dockerfile.prod` in the `node-exercise` folder.

✓ Verify the same checks you ran for the Go production image

After building your optimized image on EC2, perform **all** the verification steps we used for the Go image. This is the checklist you must complete **before** submission:

1. Build succeeds

- `DOCKER_BUILDKIT=1 docker build -t node-app:prod -f Dockerfile.prod .` must exit `0`.

2. Runtime health

- Start the naive and optimized containers on different host ports (e.g., 8080 and 8081) and confirm:
 - `curl -fsS http://localhost:<port>/health` returns `ok` for both naive and optimized images.

3. Non-root

- Verify the optimized image does not run as root:
 - Inspect `Config.User` in the image metadata or check container UID at runtime (`id -u` inside container) — it should not be `0`.

4. Image size & layering

- Compare `docker images` and `docker history` for `node-app:naive` vs `node-app:prod`. Optimized image should show fewer unnecessary layers and a reduced size.

5. HEALTHCHECK present and passing

- The optimized image should include a `HEALTHCHECK` instruction that passes when the container is started.

6. Security scan (recommended)

- Optionally run a vulnerability scan (e.g., `trivy`) on both images to observe differences; save outputs locally for your records.

7. Both images present on EC2

- Ensure both the naive and optimized images are present in the EC2 Docker image list (`docker images`), because the submission/test will be performed while both images exist on the EC2 host.
-



Submission instructions (what to do when ready)

1. Ensure both images are present on EC2

- Build the naive image (if not present): `docker build -t node-app:naive -f Dockerfile.naive .`
- Build your optimized image: `DOCKER_BUILDKIT=1 docker build -t node-app:prod -f Dockerfile.prod .`
- Confirm with `docker images | grep node-app`.

2. Copy your optimized Dockerfile to your local clab workspace (for submission):

- Open your local clab workspace editor and paste the contents of your optimized `Dockerfile.prod` into the file:

None

```
/home/labDirectory/Activity6/node-exercise/Dockerfile.prod
```

- Save the file locally.

3. Submit for testing



Cleanup After the Activity

1. Stop and remove all containers/images:

Shell

```
docker ps -aq | xargs docker rm -f
docker images -aq | xargs docker rmi -f
```

2. Remove/Prune all Caches (optional):

Shell

```
docker buildx prune -f
docker image prune -a -f
docker system prune -a --volumes -f
```

3. Remove all files from EC2 instance if needed:

Shell

```
rm -rf ~/docker_lab/*
```

Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:

Shell

```
exit
```

2. Stop or terminate the EC2 instance from the AWS Management Console if no longer required.



Congratulations — You’ve completed Activity 6!

===== END OF DOCUMENT =====