Handout: Dockerfile Instructions Overview

1. Building a Docker Image with a Dockerfile

A **Dockerfile** is a plain-text file that contains a set of instructions to build a Docker image — like a recipe that describes exactly what should go into your container.
To build an image from a Dockerfile, use:

```
docker build -t <name>:<tag> <path-to-build-context>
```

This command reads the Dockerfile in the given directory, executes its instructions, and creates a new image. The `-t` flag assigns a human-readable name and optional tag to the image, such as `myapp:latest` or `backend:v1`.

Always tag your images. If omitted, Docker assigns an untagged image ID, which makes the image harder to reference and manage.

1.1. Specifying the Build Context

The **build context** is the directory you pass as the last argument to `docker build`. Docker sends the entire contents of this directory to the Docker daemon, including all subfolders and files — unless excluded via `.dockerignore`.

Typically, you will use `docker build` from your projects root directory:

```
docker build -t myapp:latest .
```

In this case, the current directory (`.`) is used as the build context. Docker looks for a Dockerfile inside and uses any referenced files during the image build.

Docker can only access files that are part of the build context. Files outside the context cannot be copied into the image.

1.2. Optimizing the Build Context with `.dockerignore`

Limit the size of your build context to keep builds fast and clean. Use a `.dockerignore` file to exclude unnecessary files and folders such as:

- `node_modules/`, `venv/`, or other local dependencies
- build artifacts like `.out/`, `dist/`, or `*.log`
- editor or OS metadata (`.DS_Store`, `.idea/`, etc.)

Large build contexts slow down builds and may leak sensitive or unnecessary files into the image. Always use a `.dockerignore` file.

2. Defining the Base Image with `FROM`

Every Dockerfile must start with a `FROM` instruction. It sets the **base image** that your image will build upon — like choosing the starting point for your application environment.

```
FROM <image-name>:<image-tag>
```

For example, the following uses the official `python` image, version `3.12`, with the `slim` variant as the base:

```
FROM python:3.12-slim
```

The choice of base image affects the size, performance, and security of your final image. Avoid unnecessarily large images if a smaller one will do.

2.1. Selecting a Suitable Base Image

Choosing the right base image depends on your application's requirements — including runtime, size, and system-level dependencies.
In general, you should aim to **minimize image size** whenever possible. Smaller images build faster, consume less storage, and reduce potential attack surface.

Image	Category	Typical Use Case
`debian`, `ubuntu`	General-purpose	Flexible base with broad package support, but larger
`python`, `node`, etc.	Runtime-specific	Quick setup for apps using a specific language runtime
`alpine`	Minimal	Smallest size, ideal for simple or security-focused deployments

Lighter images like `alpine` reduce build time and attack surface — but may require extra effort if standard libraries or tools are missing.

A common compromise is to use a `-slim` variant based on Debian, which offers a smaller footprint than the default image while retaining compatibility.

3. The `COPY` Instruction

The `COPY` instruction adds files and directories like application code, configuration files, or static assets from your **build context** into the image's filesystem.

```
COPY <source> <destination>
```

For example, this copies everything from the build context root (excluding files ignored via `.dockerignore`) into the `/app` folder inside the image.

```
COPY . /app
```

Paths in `COPY` are relative to the build context, not the Dockerfile location. If you reference a file that's outside the build context, Docker will raise an error.

4. The `WORKDIR` Instruction

The `WORKDIR` instruction sets the working directory inside the image for any subsequent instructions like `COPY`.
Think of it like running cd in a shell — once set, all relative paths are resolved from this directory.

```
WORKDIR <path>
```

For example, this sets /app as the working directory and then copies files into it using relative paths.

```
WORKDIR /app
COPY . .
```

Each `WORKDIR` layer creates a directory if it doesn't already exist in the image. Be consistent to avoid path confusion.

5. The `RUN` Instruction

The `RUN` instruction executes commands inside the image during the build process — typically for installing packages, setting permissions, or building your application.

```
RUN <command>
```

For example, to install Python dependencies using pip:

```
RUN pip install -r requirements.txt
```

Each `RUN` creates a new intermediate layer in the image. To keep image size small and build efficient, you should combine related commands where appropriate:

```
RUN apt update && apt install-y curl
```

5.1. Optimizing Build Layers for Caching

Docker builds images in layers, and each instruction (like `COPY` or `RUN`) creates a new one.
To speed up rebuilds, Docker caches unchanged layers — so reusing common steps can drastically reduce build time.
A common optimization is to copy and install dependencies before copying the full application code. This way, Docker can cache the dependencies layer as long as your requirements haven't changed.
For example, instead of this:

```
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
```

Use this pattern:

```
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
```

This way:

- If `requirements.txt` hasn't changed, the `RUN pip install` step is cached.
- Only the final `COPY . .` step is re-executed when app code changes.

Always copy dependency files first, then install, then copy the rest of your code. This improves cache reuse and keeps builds fast and efficient.

6. The CMD Instruction

The `CMD` instruction defines the default command to run when a container starts — unless it's overridden via `docker run`.
Unlike `RUN`, which executes at build time, `CMD` runs when the image is launched as a container.

```
CMD ["executable", "param1", "param2"]
```

For example:

```
CMD ["python", "app.py"]
```

This tells Docker to start the container by running `python app.py`.
You can override the default CMD with docker run:

```
docker run myapp echo "Hello from inside the container"
```

This replaces the `CMD` in the image with the given arguments.
You can override the default `CMD` with `docker run`:

```
docker run myapp echo "Hello from inside the container"
```

This replaces the `CMD` in the image with the given arguments.

6.1. Shell vs Exec Form

`CMD` supports two forms:
Shell form (less recommended)

```
CMD python app.py
```

Docker interprets this as:

```
/bin/sh -c "python app.py"
```

This can be convenient — but it has drawbacks:

- The container process is now `sh`, **not** `python`.
- Signals (like `SIGINT` / `SIGTERM`) may not reach your app properly.
- You can't easily pass arguments to the executable later.

Exec form (preferred)

```
CMD ["python", "app.py"]
```

This launches `python` directly as PID 1 — no shell involved. It ensures:

- Clean signal handling
- Direct argument passing
- More predictable container behavior

7. The `EXPOSE` Instruction

The `EXPOSE` instruction documents which port(s) your containerized application will listen on at runtime. It does not publish the port — it only acts as metadata for humans and tooling.

```
EXPOSE <port>[/<protocol>]
```

By default, the protocol is `tcp`. For example, a typical web app might include:

```
EXPOSE 8000
```

`EXPOSE` **does not publish the port** — it only documents it. To make the port accessible, you must use `-p` with `docker run`, e.g.: `docker run -p 8000:8000 myapp` This maps port `8000` inside the container to port `8000` on your host. Think of `EXPOSE` as a **hint** — not a requirement.

8. The ARG Instruction

`ARG` defines a build-time variable, used only during `docker build`.

```
ARG VERSION=1.0
RUN echo "Building version $VERSION"
```

Pass it at build time:

```
docker build --build-arg VERSION=2.0 .
```

`ARG` values are **not included** in the final image. Use `ENV` for runtime values instead.

9. The ENV Instruction

`ENV` sets an environment variable inside the image — available to all future `RUN`, `CMD`, and processes inside the container.

```
ENV APP_ENV=production
```

You can override it at runtime:

```
docker run -e APP_ENV=development myapp
```

10. The `LABEL` Instruction

`LABEL` lets you attach metadata to your image, such as author or version.

```
LABEL maintainer="you@example.com" \
      version="1.2.3"
```

Labels are key-value pairs stored in the image — useful for automation, auditing, or organizing images.

11. Multi-Stage Builds

Multi-stage builds let you use one image for building and another for running your application — helping you create small, secure, production-ready images without including build tools or unnecessary files.
They work by chaining multiple `FROM` instructions in the same Dockerfile. You can name stages and selectively `COPY` artifacts from one stage to another.

11.1. Example: Compiling and Shipping Separately

```
# Stage 1: Build
FROM node:23 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Runtime
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

- The first stage installs dependencies and builds the app.
- The second stage is based on a lightweight runtime image.
- Only the final build artifacts are copied — not `node_modules` (the local folder containing all installed JavaScript dependencies), not the source code (your full `.js` or `.ts` files), and not even the `npm` CLI tool (Node.js's package manager used to install and run scripts).

You can use as many stages as you want. Only the **final stage** becomes your image.

Multi-stage builds keep production images minimal and secure by excluding compilers, test data, and source code — resulting in faster startups and a smaller attack surface. Always copy only what's needed from build stages to avoid leaking secrets or temporary files.