Handout: Docker Compose

1. Orchestrating Multi-Container Apps with `compose.yaml`

Docker Compose lets you define and manage multi-container applications using a single declarative file — typically named `compose.yaml`.
It serves as a blueprint for how your services (apps, databases, load balancers, etc.) work together.
To start your app stack:

```
docker compose up
```

This reads the `compose.yaml` in the current directory and starts all defined services.
In addition to creating and starting containers, Docker Compose also:

- Creates a dedicated **bridge network**, so services can communicate using their service names as hostnames (via built-in DNS)
  - This means services like `backend` can reach `db` simply by connecting to `db:5432`, without any extra network setup.
- Automatically creates and manages any **named volumes** defined under `volumes:`

2. Starting, Rebuilding, Stopping

2.1. Build and Start (first time or after changes)

```
docker compose up --build
```

- Builds any services with a `build:` section (e.g. `frontend`)
- Then starts the entire stack

Use the `--build` option when you've made code or Dockerfile changes. Docker may otherwise reuse cached images, even if your code has changed.

2.2. Stop and Remove Containers

```
docker compose down
```

- Stops all running containers
- Removes the associated containers (but preserves volumes by default)
- Removes the network created by Docker Compose

To remove named volumes as well:

```
docker compose down -v
```

Use `-v` when you want to start from scratch — it cleanly resets your setup, but keep in mind that it may delete data from databases.

2.3. Check Service Status

```
docker compose ps
```

Lists all services defined in the `compose.yaml`, along with their status, ports, and whether they are running, exited, or unhealthy.

2.4. View Logs for All Services

```
docker compose logs <service-name>
```

Displays logs for a specific service — useful for troubleshooting startup errors, failed healthchecks, or unexpected behavior.

If you're unsure which services are defined, use `docker compose ps` to list them.

3. Key Elements in `compose.yaml`

3.1. `services:` — The Building Blocks

Each service defines a container. They can be built from Dockerfiles or pulled from existing images. `build:` points to a folder with a Dockerfile → Compose will build the image locally.

```
build: ./frontend
```

`image:` uses an existing image from Docker Hub or a registry:

```
image: postgres:17
```

You can also define build-time arguments:

```
build:
  context: ./frontend
  args:
    VITE_API_BASE_URL: "/api"
```

Build arguments defined under `build.args` are passed to the Dockerfile's `ARG` instructions. They are available **only during the image build** and do **not** persist in the final image.

3.2. `restart:` — Auto-Restart Policy

The `restart:` setting controls how Docker handles container restarts — especially after crashes, daemon restarts, or manual interruptions. Example:

```
restart: unless-stopped
```

This ensures the container automatically restarts if it crashes or if the Docker daemon restarts — but it won't restart if you manually stopped it.

Policy	Behavior
`no` (default)	Never restarts the container, even if it crashes
`always`	Always restarts the container, regardless of exit reason
`unless-stopped`	Restarts the container unless it was manually stopped
`on-failure`	Restarts the container only if it exits with a non-zero status code

`unless-stopped` is a safe and common default — it ensures containers come back after system reboots but gives you manual control when needed.

3.3. `ports:` — Publishing to the Host

The `ports` section maps ports from inside the container to your local machine, making services accessible from your browser or other tools:

```
ports:
  - "8080:80" # host:container
  - "9090:443" # host:container
```

This exposes:

- Port `80` inside the container on port `8080` of the host
- Port `443` inside the container on port `9090` of the host

It's equivalent to using `-p 8080:80 -p 9090:443` with `docker run`.

The `ports` field must always be a **list of port mappings**, even if you're only defining one.

3.4. `environment:` — Passing Runtime Variables

Use the `environment:` section to define environment variables that will be available inside the container at runtime:

```
environment:
  - APP_ENV=production
```

Avoid hardcoding sensitive credentials directly in `compose.yaml`. Instead, store them in a `.env` file located in the same directory. Docker Compose will automatically load this file.
To reference values from the `.env` file (or from your shell environment), use the `$` syntax:

```
environment:
  - POSTGRES_PASSWORD=$POSTGRES_PASSWORD
```

This approach keeps secrets out of version control and makes configuration more portable across environments.

3.5. `volumes:` — Persisting or Mounting Data

Use volumes to:

- Persist data across container restarts (e.g. databases)
- Mount files or folders from your host into the container (e.g. config or initialization scripts)

Example:

```
volumes:
  - db-vol:/var/lib/postgresql/data:rw
  - ./db:/docker-entrypoint-initdb.d:ro
```

- `db-vol` is a **named volume** used to persist PostgreSQL data. It's mounted where PostgreSQL stores its database files.
- `./db` is a **bind mount** that injects files from your local `./db` directory into a special path inside the container. The official PostgreSQL image automatically executes any scripts placed there during container initialization.
- The `:ro` and `:rw` flags define read-only or read-write access

Docker distinguishes between **named volumes** and **bind mounts** based on the format of the source path: if it starts with `./` or `/`, it's treated as a **bind mount**; otherwise, it's treated as a **named volume**.
Named volumes must be declared again at the bottom of your `compose.yaml`:

```
volumes:
  db-vol:
```

The `volumes:` section handles **both named volumes and bind mounts**. Be especially careful when using relative paths like `./db` — if you accidentally omit the `./`, Docker will treat it as a **named volume** instead of a bind mount, which can lead to subtle and hard-to-debug issues.

3.6. `command:` — Overriding the Container's Default Behavior

The `command:` field in `compose.yaml` overrides the default `CMD` defined in the image's Dockerfile. It tells the container what to run when it starts.
Example:

```
command:
  - "--api.insecure=true"
  - "--entrypoints.web.address=:80"
```

This replaces the default command with custom flags — in this case, configuring Traefik's behavior.
It's equivalent to overriding the command in a `docker run` call like:

```
docker run <additional-flags> traefik:v3 --api.insecure=true
↪  --entrypoints.web.address=:80
```

4. Using Traefik as a Load Balancer and Reverse Proxy

In our setup, Traefik acts as a **reverse proxy** and **load balancer**: it receives all incoming HTTP requests on port 80 and forwards them to the correct container — based solely on labels defined in our `compose.yaml`.
A **reverse proxy** sits in front of our services and handles external requests. It can route traffic based on path, hostname, or other rules. A **load balancer**, in turn, distributes that traffic across multiple backend containers — helping to improve:

- **Scalability** (handling more users)
- **Availability** (fallbacks if one container fails)
- **Performance** (avoids overloading a single instance)

Traefik combines both roles and integrates tightly with Docker to update its routing rules automatically as containers come and go.

4.1. Integration with Docker

To make Traefik aware of other services, it is given access to the Docker socket:

```
volumes:
  - /var/run/docker.sock:/var/run/docker.sock
```

This allows it to watch running containers and read their labels — which are used to define routing behavior.

4.2. Traefik Configuration (via `command:`)

Traefik is configured entirely through CLI flags in the `command:` section of the service definition. Key flags include:

4.3. Key Traefik Flags

- `--api.insecure=true`
  Enables the Traefik dashboard (on port 8080). Don't use this in production without access control.
- `--providers.docker=true`
  Enables Docker-based service discovery.
- `--providers.docker.exposedByDefault=false`
  Only expose containers that explicitly opt in via `traefik.enable=true`.
- `--providers.docker.allowEmptyServices=true`
  Allows routes to be defined even if no container is running (e.g. for zero-downtime updates).
- `--ping=true` / `--ping.entryPoint=web`
  Enables the `/ping` endpoint, which can be used for health checks on Traefik itself.
- `--entrypoints.web.address=:80`
  Defines an HTTP entrypoint on port 80.

4.4. Routing via Labels

To route requests to a service, Traefik needs three key pieces of information:

- **Should this container be exposed at all?** → via `traefik.enable`
- **When should a request go to this service?** → via a routing `rule`
- **On which entrypoint (port) should Traefik listen for those requests?** → via `entrypoints`

All of this is defined through labels in the `compose.yaml` — no separate configuration files are required. These labels must be attached to the **service definition** of the container you want Traefik to route to. Traefik label keys follow this pattern:

```
traefik.http.routers.<router-name>.<key>
```

- `<router-name>` is user-defined (e.g. `frontend`, `api`, `admin`)
- `<key>` is the configuration setting (e.g. `rule`, `entrypoints`, `service`, `middlewares`)

To expose a new service via Traefik, attach the following labels:

```
labels:
  - "traefik.enable=true"
  - "traefik.http.routers.<router-name>.rule=<routing-rule>"
  - "traefik.http.routers.<router-name>.entrypoints=<entrypoint-name>"
```

Traefik interprets these as:

- `traefik.enable=true`: Opt this container into Traefik-based routing

- `traefik.http.routers.<router-name>.rule`: Define the matching condition for routing (e.g. path, hostname, headers)
- `traefik.http.routers.<router-name>.entrypoints`: Specify the entrypoint name to listen on (e.g. `web` → port 80, `websecure`, custom names)

Example: `backend` service

```
labels:
  - "traefik.enable=true"
  - "traefik.http.routers.backend.rule=PathPrefix(`/api`)"
  - "traefik.http.routers.backend.entrypoints=web"
```

- `traefik.enable=true`: Opts the container into Traefik's routing system
- `rule=PathPrefix(/api)`: Routes requests starting with `/api` to this container
- `entrypoints=web`: Handles those requests on the `web` entrypoint (typically mapped to port 80)

Since only Traefik publishes a port to the host, all other services can share port 80, with Traefik routing requests internally.

No container other than Traefik needs to publish a port — routing and exposure are managed completely through labels.

5. Startup Order and Health Checks

Managing container startup order and runtime readiness is essential in multi-service setups. Docker Compose offers two key mechanisms for this:

- `depends_on` to control startup order
- `healthcheck` to monitor service readiness

5.1. `depends_on`: Startup Order Only

The depends_on field ensures that one service starts after another:

```
services:
  backend:
    depends_on:
      - db
```

This guarantees that the `db` container starts before backend. However, it does not wait until the database is actually ready to accept connections.
So if your app starts quickly and tries to connect to the database too soon, it may still fail — even with `depends_on` in place.

the `depends_on` mechanism only ensures that one container starts after another — it does not wait for the dependency to be ready.

5.2. Health Checks: Monitoring Readiness

To detect whether a service is truly ready, use a `healthcheck:`. It instructs Docker to periodically execute a command and mark the container as unhealthy if the command fails.

A container with no `healthcheck:` is always considered **healthy** — even if it's failing internally.

A health check consists of a test command and optional timing parameters:

```
healthcheck:
  test: ["CMD", "<command>", "<args>"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 5s
```

Field	Description
`test`	The command Docker runs to check health
`interval`	Time between checks (default: `30s`)
`timeout`	Max time before the check is considered failed
`retries`	How many times a check must fail before marking `unhealthy`
`start_period`	Optional delay before the first check

This example checks whether the service returns a successful HTTP status from its health endpoint:

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost/health"]
  interval: 30s
  timeout: 5s
  retries: 3
```

This check sends a request to `/health` and expects a `2xx` response. If it fails 3 times in a row, the container is marked `unhealthy`.

A well-designed health check should reflect **real application readiness** — not just whether the process is running.

5.3. CMD vs. CMD-SHELL

Docker supports two formats for `test:`:

Format	Behavior
`CMD`	Runs the command directly (array syntax). More predictable.
`CMD-SHELL`	Runs the command via shell (`/bin/sh -c`). Allows shell features.

Use `CMD-SHELL` when you need:

- Pipes (`|`), OR (`||`), redirection
- Variable expansion (`$VAR`)
- More complex shell logic

This runs a PostgreSQL-specific readiness check inside the container:

```
test:["CMD-SHELL", "pg_isready -U $POSTGRES_USER -d $POSTGRES_DB -h 127.0.0.1
↪  || exit 1"]
```

- `CMD-SHELL`: Enables variable substitution (`$POSTGRES_USER`, `$POSTGRES_DB`) and shell logic.
- `pg_isready`: A built-in PostgreSQL tool that checks if the database server is ready to accept connections.
- `|| exit 1`: Ensures a failed check returns a non-zero status so Docker marks the container as `unhealthy`.

6. Autoheal: Restart Unhealthy Containers

Docker supports health checks, but does not act on them automatically. If a container becomes `unhealthy`, Docker records the status — but **won't restart it**, even if a restart could fix the issue.
This can be limiting in production setups where long-running containers may temporarily fail due to connectivity issues, crashed dependencies, or memory spikes.
To fill this gap, you can use a **third-party helper** container such as `willfarrell/autoheal`. It listens to Docker's internal events via the Docker socket and automatically restarts any container labeled for autohealing if it becomes `unhealthy`.

```
services:
  autoheal:
    image: willfarrell/autoheal
    restart: always
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

To enable auto-healing for a service, add this label:

```
labels:
  - "autoheal=true"
```

Autoheal only works with containers that define a working `healthcheck:`. Without a `healthcheck`, Docker will consider the container healthy — even if it's broken.