Docker Masterclass Notes

GitHub Reference Repo:

https://github.com/snkshukla/masterclass-sample

These notes complement our live session, providing deeper insight and helpful commands. Refer back to them whenever you need a refresher on the concepts taught in class.

---

1. The Software Development Life Cycle (SDLC)

1.1 What is SDLC?

A **Software Development Life Cycle (SDLC)** is the process that guides software projects from **planning** through **maintenance**. Typical stages include:

- **Planning**: Determining what to build and why.
- **Development**: Writing the code, implementing features.
- **Testing**: Ensuring the software works as expected.
- **Deployment**: Making the software available to users.
- **Monitoring / Maintenance**: Keeping the software running smoothly, adding updates.

1.2 Focus on Development & Deployment

- **Development**: Where coding, debugging, and feature building happen.
- **Deployment**: Getting your application from the developer’s machine out into the world—whether on a server, cloud, or container platform.

---

2. Challenges in a "Dockerless" World

Without containers, many **pain points** often emerge:

1. **Inconsistent Environments**: Developers use different OS versions and libraries, leading to “it works on my machine” issues.

2. **Dependency Hell**: Conflicting versions of frameworks or packages across different environments cause unexpected errors.
3. **Testing Gaps**: Test environments may not mirror production, causing integration issues late in the pipeline.
4. **Manual Configuration**: Setting up servers, installing dependencies, and configuring them can be time-consuming and error-prone.
5. **Scalability & Downtime**: Scaling monolithic apps or performing updates often causes service disruption.

---

3. Introducing Docker

3.1 Before Docker: The JVM Example

- **Java Virtual Machine (JVM)**:
  - Lets Java code run on any OS that has a JVM installed.
  - James Gosling's concept: **"Write Once, Run Anywhere."**
- **Limitations**:
  - JVM only helps with Java-based dependencies.
  - Non-Java stacks still need consistent environments.

3.2 Docker's Core Idea

- **What Docker Does**: Packages your application and all its dependencies into standardized units called **containers**.
- **Analogy**: Like a **shipping container—**it carries everything your app needs (libraries, runtime, configs), so it runs the same way everywhere.
- **Key Benefits**:
  1. **Consistency**: Same environment in dev, test, and production.
  2. **Efficiency**: Containers share the host OS kernel, making them lightweight compared to virtual machines.
  3. **Isolation**: Each container is isolated, preventing conflicts between different apps.

---

4. Hands-on with Docker

4.1 Basic Commands

1. `docker run hello-world`
    - Tests if Docker is working. It pulls a small image and prints a welcome message.
2. `docker ps`
    - Lists currently running containers.
3. `docker images`
    - Shows images available locally.
4. `docker stop <container_id>`
    - Stops a running container by ID.

4.2 Dockerfiles

A **Dockerfile** is a **recipe** that tells Docker how to build an image:

- **Why Use Dockerfiles?**
    - Automates installs, configuration, and environment setup.
    - Repeatable, version-controlled instructions.

Common Dockerfile Instructions

1. **FROM**: Specifies the base image (e.g., `ubuntu:latest`).
2. **COPY**: Copies files from your local machine into the container.
3. **RUN**: Runs commands at build time, e.g. `RUN apt-get update`.
4. **CMD**: Specifies the **default command** when the container starts, e.g. `CMD ["python", "app.py"]`.

4.3 Example: A Simple Python App

1. **app.py**:

```
print("Hello from Docker!")
```

2. **Dockerfile**:

```
FROM python:3.9-slim
COPY app.py /app/app.py
WORKDIR /app
CMD ["python", "app.py"]
```

3. **Build & Run**:

```
docker build -t my-python-app .
docker run my-python-app
```

   - `t my-python-app` tags the image with a name (`my-python-app`).

4.4 Real-World Dockerfile (Open Source Example)

- **Link**: Python Official Dockerfile
- Notice `FROM` is `alpine3.21` or `debian:buster-slim`, showing how they start with a minimal OS and install Python on top.

---

5. Local Orchestrator - Docker Compose

5.1 Why Docker Compose?

- Orchestrates multiple containers on **a single host** (like your laptop).
- Ideal for local development with multi-container apps (e.g., a web app + database).

5.2 Key Concepts

1. **docker-compose.yml**: Defines services (containers), networks, volumes.
2. **services**: Each represents one container or set of containers.
3. **depends_on**: Ensures containers start in the correct order.
4. **ports**: Maps ports from container to host.
5. **environment**: Sets environment variables for containers.

5.3 Simple Example: Web + Database

```yaml
version: '3'
services:
  web:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - db
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: secret
      POSTGRES_USER: user
      POSTGRES_DB: mydb
```

- **Command**: `docker-compose up`
- This automatically starts both the `web` and `db` containers, networking them together.

---

6. Microservices

6.1 What are Microservices?

An architectural pattern where an application is **broken down** into small, independent services. Each service focuses on a specific function (e.g. payments, authentication, product catalog).

6.2 Challenges with Monolithic Apps

1. **Massive Codebase**: Hard to maintain when many developers are involved.
2. **Scaling**: Must scale the entire app even if only one feature needs more resources.
3. **Deployment Risks**: Small changes can affect the entire system.
4. **Technology Lock-in**: Hard to adopt new tech if everything is tied together.

6.3 Benefits of Microservices

1. **Independent Deployment**: Update or roll back one service without touching others.
2. **Scalability**: Scale services individually.
3. **Resilience**: If one microservice fails, others can remain operational.
4. **Technology Freedom**: Each service can use different languages or frameworks.

6.4 Real Example: Netflix

- Operates **hundreds** of microservices, each serving a unique function (user profiles, recommendations, streaming).
- Deploys **hundreds** of times per day—enabled by small, independent services.

---

7. The Need for Orchestration

7.1 Challenges Without an Orchestrator

- Managing multiple microservices across many servers is **complex**.
- Need to handle:
  - **Replicas** for high availability
  - **Autoscaling** based on traffic
  - **Service discovery** (how services find each other)
  - **Networking** rules, load balancing, etc.

7.2 The Orchestra Analogy

- Think of each microservice as a musician.
- **Orchestration**: The conductor (orchestration tool) ensures everyone plays in **sync** and adjusts as needed.

7.3 Tools

- **Docker Compose**: Good for single-host dev setups.
- **Kubernetes**: The most popular choice for production environments and multi-node clusters.

---

8. Production Orchestrator - Kubernetes (Intro)

Kubernetes manages containers at scale, across multiple machines (nodes). It automates **deployment**, **scaling**, and **load balancing** for containerized applications.

8.1 Namespaces

- **Definition**: A **namespace** in Kubernetes is a way to divide cluster resources among multiple **virtual sub-clusters**.
- **Use Cases**:
  - **Environment separation** (e.g., `dev`, `staging`, `production`).
  - **Team isolation** (each team in its own namespace).
  - **Resource Quotas**: Enforce resource limits per namespace.

Command:

- List namespaces: `kubectl get namespaces`
- Create a namespace: `kubectl create namespace my-namespace`
- Use a namespace: `kubectl -n my-namespace get pods`

8.2 Simple Deployment + Service Example

Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
  namespace: my-namespace
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
```

```yaml
    labels:
      app: myapp
  spec:
    containers:
    - name: myapp-container
      image: myusername/myapp:1.0
      ports:
      - containerPort: 8080
```

Explanation

- **apiVersion / kind**: Tells Kubernetes we're creating a `Deployment` from `apps/v1`.
- **metadata**: Specifies name (`myapp-deployment`) and **labels** (key-value metadata).
- **namespace**: Indicates this resource belongs to the `my-namespace` namespace.
- **spec**:
  - **replicas: 2** means we want two replicas (pods) running.
  - **selector.matchLabels**: Must match the labels in `template.metadata.labels`.
  - **template**: Defines the **pod** specification (metadata + containers).
  - **containerPort: 8080**: The container listens on port 8080.

Service YAML

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
  labels:
    app: myapp
  namespace: my-namespace
spec:
  selector:
    app: myapp
  ports:
    - port: 80
```

```
    targetPort: 8080
  type: ClusterIP
```

Explanation

- **kind: Service**: We're creating a logical grouping of pods under one stable network endpoint.
- **selector.app: myapp**: This Service targets **Pods** with the label `app=myapp`.
- **ports**:
  - **port: 80** means the Service is exposed internally on port 80.
  - **targetPort: 8080** means the Pods themselves listen on port 8080.
- **type: ClusterIP**: Default service type for internal communication within the cluster.

Hands-on Commands

1. **Create Namespace** (optional, if you haven't already):

   ```
   kubectl create namespace my-namespace
   ```

2. **Deploy**:

   ```
   kubectl apply -f deployment.yaml
   kubectl apply -f service.yaml
   ```

   - or combine in one file and apply both.
   - Add `-n my-namespace` if your YAML doesn't already specify a namespace.

3. **Check Status**:

   ```
   kubectl get deployments -n my-namespace
   kubectl get pods -n my-namespace
   kubectl get services -n my-namespace
   ```

4. **Port Forward** (access your app locally):

```
kubectl port-forward service/myapp-service 8080:80 -n my-namespac
e
```

- Now visit `http://localhost:8080` to reach the Pods.

Conclusion

1. **Recap**:
    - **Docker** simplifies building, sharing, and running applications in containers.
    - **Docker Compose** orchestrates multi-container setups on a single host—ideal for local development.
    - **Microservices** break large apps into smaller, independent services that can be deployed and scaled independently.
    - **Kubernetes** orchestrates containers at scale for production, ensuring automated deployments, scaling, and resilience.
2. **Further Learning**:
    - Explore official Docker images on Docker Hub.
    - Check out the **Kubernetes documentation** for advanced use cases.
    - Practice with the **sample GitHub repo**:

      https://github.com/snkshukla/masterclass-sample
3. **Stay Connected**:
    - Continue experimenting, ask questions in your community, and share your progress with peers.
    - The best way to master containers and microservices is **hands-on practice**!

Thank you for joining the Masterclass!

For any feedback or additional questions, please reach out. Happy Containerizing!