📘 Phase-1: Enterprise Application Development

1. Introduction to Enterprise Application Development

🔹 Definition

Enterprise applications are **large-scale, business-critical software** that manage data, processes, and services.

They must be:

- **Scalable** (handle millions of users),
- **Reliable** (minimal downtime),
- **Secure** (protect data & APIs),
- **Modular** (easy to maintain).

💡 **Docker** is widely used to package these applications into containers, so they run consistently across environments (dev, test, prod).

2. Monolithic vs Microservice Architecture (with Docker)

🔹 Monolithic Approach (MERN Example + Docker)

Structure: All in one codebase.

📂 Example Project Tree:

```
monolithic-app/
 ├── backend/ (Express + Node)
 ├── frontend/ (React)
 ├── database/ (MongoDB)
 ├── Dockerfile
 └── docker-compose.yml
```

docker-compose.yml (Monolithic)

```
version: "3"
services:
  mongo:
    image: mongo
    container_name: mongo_container
    ports:
      - "27017:27017"

  backend:
    build: ./backend
    ports:
      - "5000:5000"
    depends_on:
      - mongo

  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend
```

👉 Here, **frontend** + **backend** + **database** run as containers.

But backend has **all routes (users, products, orders)** inside one app → difficult to scale separately.

🔹 Microservice Approach (MERN + Docker)

Structure: Break into multiple services (users, products, orders).

📂 Example Project Tree:

```
microservice-app/
 ├── user-service/ (Express + Mongo)
 ├── product-service/ (Express + Mongo)
 ├── order-service/ (Express + Mongo)
 ├── api-gateway/ (Reverse proxy)
 ├── frontend/ (React)
 ├── docker-compose.yml
```

docker-compose.yml (Microservices)

```
version: "3"
services:
  mongo-users:
    image: mongo
    container_name: mongo_users
    ports:
      - "27018:27017"

  mongo-products:
    image: mongo
    container_name: mongo_products
    ports:
      - "27019:27017"

  user-service:
    build: ./user-service
    ports:
      - "5001:5000"
    depends_on:
      - mongo-users

  product-service:
    build: ./product-service
    ports:
      - "5002:5000"
    depends_on:
      - mongo-products
  order-service:
    build: ./order-service
    ports:
      - "5003:5000"

  api-gateway:
    build: ./api-gateway
    ports:
      - "5000:5000"
    depends_on:
      - user-service
      - product-service
      - order-service

  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - api-gateway
```

👉 Each **service has its own database**.

👉 We can **scale one service** (`docker-compose up --scale product-service=3`) without scaling others.

3. Scalability & Performance of Applications (Docker Perspective)

- **Horizontal Scaling**: Spin up multiple containers for high-load services.
- **Load Balancing**: Use **NGINX or API Gateway** to distribute requests across containers.
- **Caching**: Add a **Redis container** to reduce DB queries.
- **Monitoring**: Use **Prometheus** + **Grafana** to track performance.

Example (Scale Product Service)

```
docker-compose up --scale product-service=3 -d
```

👉 Now 3 containers of `product-service` handle requests → better performance during traffic spikes.

---

4. MERN Stack with Docker

📂 Typical Setup in `docker-compose.yml`

```
version: "3"
services:
  mongo:
    image: mongo
    container_name: mongo_db
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db

  backend:
    build: ./backend
    container_name: express_backend
    ports:
      - "5000:5000"
    depends_on:
      - mongo

  frontend:
    build: ./frontend
    container_name: react_frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend
```

```
volumes:
  mongo_data:
```

👉 With this, a full **MERN app runs inside containers**, ensuring consistent environments across dev/prod.

---

5. Microservice Architecture (Detailed in MERN + Docker Project)

Let’s design a **Dockerized E-commerce MERN App**:

- **User Service (port 5001)** → handles login, register
- **Product Service (port 5002)** → manages catalog
- **Order Service (port 5003)** → handles cart, orders
- **Payment Service (port 5004)** → Stripe/PayPal integration
- **API Gateway (port 5000)** → routes requests to correct service
- **React Frontend (port 3000)** → UI for users

📂 `docker-compose.yml` will spin up all services together.

👉 Benefit: You can scale **Order Service** separately during festive sales.

---

6. Breaking Down Application (Project Approach)

Example: Dockerized MERN E-commerce

1. **Frontend (React, containerized)**
    - Login, Products, Cart, Checkout pages.
    - Dockerfile:

    ```
    FROM node:18
    WORKDIR /app
    COPY package*.json ./
    RUN npm install
    ```

```
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

2. **Backend (Express, containerized)**
    - Routes → `/api/users`, `/api/products`, `/api/orders`
    - Dockerfile:

      ```
      FROM node:18
      WORKDIR /app
      COPY package*.json ./
      RUN npm install
      COPY . .
      EXPOSE 5000
      CMD ["node", "server.js"]
      ```

3. **Database (Mongo, containerized)**
    - Persist with Docker volume.
4. **Microservices (Optional scaling)**
    - Split backend into **user-service, product-service, order-service**.

✅ Summary of Phase-1 (with Project + Docker)

1. **Enterprise Apps** → large, scalable, secure software for organizations.
2. **Monolithic vs Microservices** → monolithic is simple but rigid, microservices are modular and scalable.
3. **Scalability & Performance** → use horizontal scaling (Docker), caching, load balancing.
4. **MERN + Docker** → MongoDB, Express, React, Node all containerized.
5. **Microservices in MERN** → break into independent services for better scaling.
6. **Break Down Applications** → React frontend, Express backend, Mongo DB, Dockerized services.
