Software Design and Engineering

Lab Document

High Level Purpose Statement:	The goal of today's lab was to containerize the Smart Schedule Tool Express application using Docker and Docker Compose. By packaging the app into containers, we ensure consistent runtime environments, simplify dependency management, and streamline deployment.
Experimental Design:	1. **Verify Docker is Running** o Executed docker info in the IntelliJ terminal to confirm the Docker daemon (Desktop) was active on Windows. o Resolved any connection issues by restarting Docker Desktop. 2. **Create Dockerfile** o In the project root, added a file named Dockerfile with: o FROM node:20-alpine o WORKDIR /usr/src/app o COPY package*.json ./ o RUN npm ci --only=production o COPY . . o EXPOSE 3000 CMD ["npm","start"] o Used npm start in CMD to automatically pick up the app's entry script defined in package.json. 3. **Build Docker Image** o Ran docker build -t smart-schedule-app . to build and tag the image. o Confirmed successful tagging of smart-schedule-app:latest. 4. **Run Container** o **Interactive Mode**: docker run --rm -p 3000:3000 smart-schedule-app ``` - Verified the Express server output and accessed the app at http://localhost:3000. • **Detached Mode**: docker run -d --name schedule-tool -p 3000:3000 smart-schedule-app docker logs -f schedule-tool docker stop schedule-tool ``` - Demonstrated background execution, log streaming, and clean shutdown. 5. **Container Inspection** o Executed: docker run --rm -it smart-schedule-app sh ls -l /usr/src/app exit ``` • Confirmed application files were correctly copied into the container at /usr/src/app.

	6. **Docker Compose Workflow** o Created docker-compose.yml in the project root: version: "3.9" services: web: build: . image: smart-schedule-app:latest ports: - "3000:3000" restart: unless-stopped ``` • Brought up the service with: docker-compose up --build -d docker-compose logs -f web docker-compose down ``` • Verified multi-step build and teardown workflows. 7. **IDE Integration** o Enabled IntelliJ's Docker plugin and opened the Docker tool window. o Configured a Docker Compose run configuration to build and launch the web service directly from the IDE.
Resources Available:	Docker Documentation (https://docs.docker.com) for Docker Engine and Docker Compose reference. Node.js & npm Docs (https://nodejs.org) for package management and scripts. IntelliJ IDEA Docker Plugin Guide for IDE integration. Your project's package.json and Dockerfile as primary configuration references.
Time Estimate:	• 5 min verifying Docker daemon with docker info. • 5 min creating and validating the Dockerfile. • 2 min building the Docker image. • 5 min running containers (interactive & detached) and exploring logs. • 3 min inspecting container filesystem via shell. • 5 min authoring docker-compose.yml and testing Compose workflow. • 5 min configuring IntelliJ Docker plugin and run configurations. Total: ~25 minutes
Experiment Notes:	Missing Dockerfile initially caused a build error; resolved by adding it exactly as Dockerfile in project root. Using CMD ["npm","start"] simplified entrypoint management by deferring to package.json's start script. Learned that Docker Desktop on Windows uses a Linux VM under WSL2 for container execution. Confirmed container filesystem layout (/usr/src/app) via an interactive shell session. Docker Compose streamlined multi-step workflows into single commands, improving efficiency.
Results:	**Environment Validation**: docker info confirmed a running Docker daemon.

	Image Build: smart-schedule-app:latest built without errors. **Container Deployment**: The Express app served correctly at http://localhost:3000 in both interactive and detached modes. **Inspection**: Verified container file structure via interactive shell. **Compose Orchestration**: Multi-service commands executed successfully, demonstrating rapid deployment & teardown. **IDE Workflow**: Docker plugin configurations enabled GUI-based builds and container management.
Consequences for the Future:	**Consistency & Portability**: Docker ensures the app runs identically across machines and CI/CD pipelines. **Next Steps**: Extend docker-compose.yml to include additional services (e.g., databases, caches). Integrate health checks and volume mounts for persistent data. Automate builds and deployments via a CI/CD platform (e.g., GitHub Actions). o Implement multi-stage Docker builds to optimize image size and security.