Session 04: Containerize a CRUD microservice by creating a Docker container for it, ensuring that it can run independently with all necessary dependencies.

Pre-Lab

1. What is a CRUD microservice?

A **CRUD microservice** is a small, independent application that provides functionality to **Create, Read, Update, and Delete (CRUD)** data related to a specific resource. It exposes APIs (usually RESTful APIs) to perform these operations and works independently from other services in a microservices architecture.

Example:
A *Student Service* that allows:

- Create a student record
- Read student details
- Update student information
- Delete a student record

2. Why do we containerize a microservice? Give one reason.

We containerize a microservice to ensure **consistent execution across different environments**.

Reason:
Containerization packages the application along with its dependencies, libraries, and runtime, allowing it to run the same way on any system without compatibility issues.

3. What is the purpose of a Dockerfile when containerizing a microservice?

A **Dockerfile** is used to define the **instructions required to build a Docker image** for the microservice.

Purpose of a Dockerfile:

- Specifies the base image
- Installs required dependencies
- Copies application code
- Defines the command to run the microservice

It ensures that the microservice can be built and executed **independently and reproducibly** inside a container.

In-Lab Tasks

1. Containerize a simple RESTful CRUD app (Node.js/Express or Python/FastAPI) using a multi-stage Dockerfile. Build an image under 300 MB, run it with a mounted volume for development.

A **RESTful API** is a web service that uses standard **HTTP methods** to perform operations on resources identified by **URLs**, following REST architectural principles.

A RESTful API is a web service that uses HTTP methods like GET, POST, PUT, and DELETE to access and manipulate resources identified by URLs.

Project Structure:

```
project/
├── Dockerfile
├── package.json
└── index.js
```

Step 1: Create RESTful CRUD Application

File: index.js

```
const express = require("express");
const app = express();
const port = 3000;

app.use(express.json());

let items = {};

// Default Route
app.get("/", (req, res) => {
  res.send("Welcome to the Item API!");
});

// POST - Create new item
app.post("/items/:id", (req, res) => {
  items[req.params.id] = req.body.item;
  res.json({ message: "Item created", item: items[req.params.id] });
});

// GET - Retrieve an item by ID
app.get("/items/:id", (req, res) => {
  const item = items[req.params.id];
  if (item) {
    res.json({ item });
  } else {
    res.status(404).json({ message: "Item not found" });
  }
});

// PUT - Update an existing item
app.put("/items/:id", (req, res) => {
  items[req.params.id] = req.body.item;
  res.json({ message: "Item updated", item: items[req.params.id] });
});

// DELETE - Delete an item by ID
app.delete("/items/:id", (req, res) => {
  const deletedItem = items[req.params.id];
  delete items[req.params.id];
  if (deletedItem) {
    res.json({ message: "Item deleted", item: deletedItem });
  } else {
    res.status(404).json({ message: "Item not found" });
  }
});

// Start the server
app.listen(port, "0.0.0.0", () => {
  console.log(`Server running on port ${port}`);
});
```

Step 2: Define Dependencies

File: package.json

```json
{
  "name": "crud-app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "dev": "nodemon index.js"
  },
  "dependencies": {
    "express": "^4.19.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.3"
  }
}
```

Step 3: Create Multi-Stage Dockerfile

File: Dockerfile

```dockerfile
# ---------- Builder stage ----------
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install

# ---------- Runtime stage ----------
FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/node_modules ./node_modules
COPY . .
EXPOSE 3000
CMD ["node", "index.js"]
```

Explanation

- Builder stage installs dependencies
- Runtime stage copies only required files
- Multi-stage build reduces image size below **300 MB**

Step 4: Build Docker Image

docker build -t crudapp_image .

Step 5: Run Container with Mounted Volume

docker run -d --name crudapp -p 3000:3000 -v app_volume:/app crudapp_image

Step 6: Verify Container

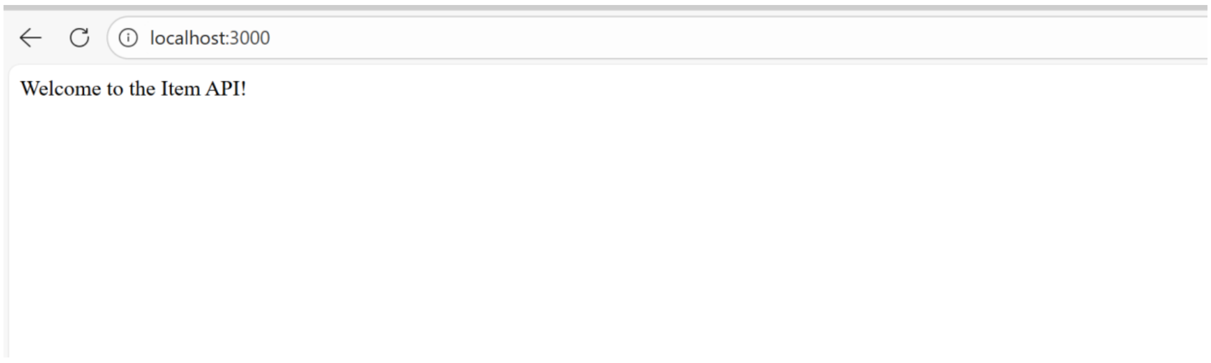
docker ps

Step 7: Verify Application Output

Open browser and access:

http://localhost:3000/

Output:



```
Welcome to the Item API!
```

Result

A simple RESTful CRUD microservice was successfully containerized using a **multi-stage Dockerfile**. The Docker image was built under **300 MB**, and the container was executed with a **mounted volume**, enabling independent and efficient development.

Post Lab Program

1. Run the RESTful CRUD app container, and verify the functionality of the application by testing all CRUD endpoints.

Step 1: Run the CRUD Application Container

```
docker run -d --name crudapp -p 3000:3000 crudapp_image
```

Explanation:

- -d → Runs container in detached mode
- -p 3000:3000 → Maps host port to container port
- crudapp_image → Docker image name

Step 2: Verify Container Status

```
docker ps
```

Expected Output:

- Container crudapp should be in **Up** state

Step 3: Verify Application is Running

Open browser and enter:

http://localhost:3000/

Expected Output:

Welcome to the Item API!

✓ Confirms the containerized application is running successfully.

Step 4: Test CRUD Endpoints (PowerShell Compatible)

In PowerShell, use **Invoke-WebRequest**

1. CREATE – Create a New Item (POST)

```
Invoke-WebRequest `
  -Uri http://localhost:3000/items/1 `
  -Method POST `
  -Headers @{ "Content-Type" = "application/json" } `
  -Body '{"item":"Laptop"}'
```

Expected Response:

```
{
  "message": "Item created",
  "item": "Laptop"
}
```

2. READ – Retrieve an Item (GET)

```
Invoke-WebRequest http://localhost:3000/items/1
```

Expected Response:

```
{
  "item": "Laptop"
}
```

3. UPDATE – Update an Item (PUT)

```
Invoke-WebRequest `
  -Uri http://localhost:3000/items/1 `
  -Method PUT `
  -Headers @{ "Content-Type" = "application/json" } `
  -Body '{"item":"Gaming Laptop"}'
```

Expected Response:

```
{
  "message": "Item updated",
  "item": "Gaming Laptop"
}
```

4. DELETE – Delete an Item (DELETE)

```
Invoke-WebRequest `
  -Uri http://localhost:3000/items/1 `
  -Method DELETE
```

Expected Response:

```
{
  "message": "Item deleted",
  "item": "Gaming Laptop"
}
```

5. VERIFY DELETE (Optional)

Invoke-WebRequest http://localhost:3000/items/1

Expected Response:

```
{
  "message": "Item not found"
}
```

CRUD Verification Summary

Operation	HTTP Method	Endpoint	Status
Create	POST	/items/:id	Verified
Read	GET	/items/:id	Verified
Update	PUT	/items/:id	Verified
Delete	DELETE	/items/:id	Verified

Result

The RESTful CRUD application container was successfully executed. All **CRUD operations (Create, Read, Update, Delete)** were tested using PowerShell commands, and the application responded correctly for each request, confirming proper functionality of the containerized microservice.