👩🏽‍💻

# Building Your 1st Microservices

We will create a simple **Spring Boot** microservice that mimics a Netflix-like feature—let's say an API that manages a list of movies. It will expose RESTful endpoints for operations like retrieving a list of movies and adding a new movie.

## ▼ Spring Boot Setup

- Create a new **Spring Boot** project using **Spring Initializr**.
- Add the following dependencies:
    - **Spring Web** for building RESTful services.
    - **Spring Data JPA** for database interaction (if needed).
    - **H2 Database** for a simple in-memory database (or any database you prefer).

### Example `pom.xml` Dependencies:

```
<dependencies>
    <!-- Spring Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- H2 Database -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
</dependencies>
```

- **Add Swagger Dependencies:**
  - You can add the following dependencies for **springdoc-openapi** (which is the library for integrating OpenAPI 3 with Spring Boot):
  - In your `pom.xml` file (if you're using Maven):

```
<!-- Swagger-OpenAPI Integration for Spring Boot 3.x -->
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
            <version>2.1.0</version>
        </dependency>

        <!-- Optional: Springdoc OpenAPI Security (for OAuth2 or Security integrations) -->
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-starter-common</artifactId>
            <version>2.1.0</version>
        </dependency>
```

- **Configure Swagger (OpenAPI)**
    - Once you've added the dependencies, you can configure Swagger automatically. The **springdoc-openapi** library automatically generates API documentation for your REST controllers and provides a Swagger UI.
    - If you want to customize your Swagger UI or OpenAPI configuration, create a `SpringDocConfig` class:

    ```
    import org.springdoc.core.models.GroupedOpenApi;
    import org.springframework.context.annotation.Bean;
    import org.springframework.context.annotation.Configuration;

    @Configuration
    public class SpringDocConfig {

        @Bean
        public GroupedOpenApi publicApi() {
            return GroupedOpenApi.builder()
                .group("public-api")
                .pathsToMatch("/api/**")  // Adjust based on your API path
                .build();
        }
    }
    ```

## Create a Simple REST Controller

In this step, create a controller that exposes the movie catalog. This will allow you to **GET** a list of movies and **POST** a new movie.

### Example `MovieController.java`:

```
@RestController
@RequestMapping("/api/movies")
public class MovieController {
```

```
    private final List<Movie> movies = new ArrayList<>
();

    // Get all movies
    @GetMapping
    public List<Movie> getMovies() {
        return movies;
    }

    // Add a new movie
    @PostMapping
    public Movie addMovie(@RequestBody Movie movie) {
        movies.add(movie);
        return movie;
    }
}
```

### Example `Movie.java` Entity:

```
public class Movie {
    private String title;
    private String genre;
    private int releaseYear;

    // Constructors, getters, and setters ...
}
```

## ▼ Configure Swagger in Spring Boot

- Spring Boot automatically configures Swagger when you include the `springdoc-openapi-ui` dependency.
- To customise the documentation, you can add the following configuration in `application.properties`:

```
springdoc.api-docs.enabled=true
springdoc.swagger-ui.path=/swagger-ui.html
```

- When you start the Spring Boot application, you can access the API documentation at `http://localhost:8080/swagger-ui.html`.
- This provides a visual interface where developers can see the API endpoints, test them interactively, and view response details.

## ▼ Create a `Dockerfile`

Now that we have a functional microservice, we will containerize it using Docker.

In the root of your project, create a `Dockerfile` to define how the application will be containerized.

### Example `Dockerfile`:

```
# Use an official OpenJDK runtime as a parent image
FROM openjdk:17-jdk

# Set the working directory inside the container
WORKDIR /app

# Copy the packaged Spring Boot app to the container
COPY target/movie-service-0.0.1-SNAPSHOT.jar /app/movie-service.jar

# Run the application
ENTRYPOINT ["java", "-jar", "movie-service.jar"]

# Expose the port on which the app will run
EXPOSE 8080
```

## ▼ Build and Run the Docker Image

1. **Build the Docker image**:

   ```
   docker build -t movie-service .
   ```

2. **Run the Docker container**:

```
docker run -p 8080:8080 movie-service
```

At this point, your microservice is running inside a Docker container, and you can access it at `http://localhost:8080/api/movies`.

## Deploy the Service and Test the API

- Now that the service is running in Docker, you can test it using the Swagger UI that we set up earlier. Open your browser and go to `http://localhost:8080/swagger-ui.html`.
- You'll be able to interact with the API, test the endpoints, and view the generated documentation.

## Set Up CI/CD with GitHub Actions

- The goal here is to set up continuous integration and deployment so that every code change is automatically built, tested, and deployed.
- This automation allows teams to ship changes more quickly and reduces the risk of errors or delays caused by manual deployments.
- GitHub Actions provides a native way to automate tasks in a CI/CD pipeline, such as running tests, building Docker images, and pushing them to Docker Hub.

### Example `ci.yml` Workflow:

Create a GitHub Actions workflow file in the `.github/workflows` directory of your project. This file defines the steps needed to build, test, and deploy the microservice.

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      # Step 1: Check out the code (v4)
      - name: Checkout code
        uses: actions/checkout@v4

      # Step 2: Set up JDK (Java Development Kit) - v4 and set to Java 17
      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: '17'

      # Step 3: Build the Spring Boot application using Maven
      - name: Build with Maven
        run: mvn clean package

      # Step 4: Build the Docker image
      - name: Build Docker image
        run: docker build -t your-dockerhub-username/movie-service .

      # Step 5: Log in to Docker Hub
      - name: Log in to Docker Hub
        run: echo "${{ secrets.DOCKERHUB_PASSWORD }}" | docker login -u "${{ secrets.DOCKERHUB_USERNAME }}" --password-stdin

      # Step 6: Push the Docker image to Docker Hub
      - name: Push Docker image to Docker Hub
        run: docker push your-dockerhub-username/movie-service
```

This workflow automatically triggers on every push to the `main` branch and pushes the Docker image to Docker Hub.

## Automate Deployment with Docker Hub and Kubernetes (Optional)

- After building the Docker image and pushing it to Docker Hub, we can deploy the microservice to a container orchestration platform like **Kubernetes** for scalability.
- If you're using **Kubernetes**, here's how you can deploy your Dockerized microservice.

1. **Create a Kubernetes Deployment YAML**:
   Create a
   `deployment.yaml` file that defines how Kubernetes will run the microservice.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: movie-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: movie-service
  template:
    metadata:
      labels:
        app: movie-service
    spec:
      containers:
      - name: movie-service
        image: your-dockerhub-username/movie-service:latest
        ports:
        - containerPort: 8080
        resources:
          limits:
            memory: "512Mi"
            cpu: "500m"
          requests:
            memory: "256Mi"
            cpu: "250m"
---
apiVersion: v1
kind: Service
metadata:
  name: movie-service
spec:
  selector:
    app: movie-service
  ports:
    - protocol: TCP
      port: 80         # The port to expose externally
      targetPort: 8080 # The port your container is listening on
  type: LoadBalancer   # Exposes the service externally (can use NodePort for internal testing)
```

1. **Deploy to Kubernetes**:
   - If you have a Kubernetes cluster running (e.g., on Google Kubernetes Engine or a local Minikube cluster), deploy the microservice using `kubectl`.

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

## Using Docker Compose:

Alternatively, if Kubernetes is too complex for your use case, **Docker Compose** can help orchestrate multiple containers, such as running the microservice alongside a database.

1. **Create a `docker-compose.yml` File**:
   Example
   `docker-compose.yml` to run the movie-service and an H2 database together.

```
version: '3'
services:
  movie-service:
    image: your-dockerhub-username/movie-service:latest
```

```
    ports:
      - "8080:8080"   # Expose movie-service on port 8080
    environment:
      SPRING_DATASOURCE_URL: jdbc:h2:tcp://h2db:9092/~/test  # Connect movie-service to H2 DB
      SPRING_DATASOURCE_USERNAME: sa
      SPRING_DATASOURCE_PASSWORD:
      SPRING_DATASOURCE_DRIVER_CLASS_NAME: org.h2.Driver
    depends_on:
      - h2db

  h2db:
    image: oscarfonts/h2
    ports:
      - "8081:8081"   # H2 console exposed on port 8081
    environment:
      H2_OPTIONS: '-tcp -tcpAllowOthers -web -webAllowOthers'  # Allow remote connections
    command: java -jar /opt/h2/bin/h2.jar  # Run H2 database as a service
```

1. **Run the Docker Compose setup**:
   Start the services using Docker Compose.

```
docker-compose up
```

This will run both the microservice and the H2 database in containers and expose the microservice on `http://localhost:8080`.