#CS #sql

Warehouse Management System – MongoDB & Spring Boot

Verfasser: **Oliwier Przewlocki**
Datum: **25.2.2025**

1. Einführung

In modernen Lagerverwaltungssystemen spielen NoSQL-Datenbanken eine entscheidende Rolle, um flexible und skalierbare Datenspeicherung zu ermöglichen. Dieses Projekt setzt MongoDB als dokumentenorientierte Datenbank und Spring Boot für die Backend-Implementierung ein. Ziel ist die Verwaltung von Lagerstandorten und deren Produkten über eine REST-API.

2. Projektbeschreibung

Das Warehouse Management System erlaubt das Hinzufügen, Abrufen, Aktualisieren und Löschen von Lagerstandorten und Produkten über eine REST-Schnittstelle. Die Daten werden in **MongoDB** gespeichert, und mit **Spring Boot** wird eine Middleware entwickelt, die diese Daten verwaltet. Swagger UI dient zur Dokumentation und erleichtert das Testen der API.

3. Theorie

- **MongoDB**: Eine NoSQL-Datenbank, die Dokumente im **JSON-ähnlichen BSON-Format** speichert. Sie eignet sich für Anwendungen, die flexible Datenmodelle erfordern.
- **Spring Boot**: Ein Java-Framework zur schnellen Entwicklung von Microservices und Webanwendungen. Es integriert sich nahtlos mit **Spring Data MongoDB**.
- **REST API**: Ein Architekturstil für Webservices, der auf HTTP-Methoden wie **GET, POST, PUT, DELETE** basiert.
- **Swagger UI**: Eine Schnittstelle zur Dokumentation und zum Testen von REST-APIs.

Warehouses kann man mittels `db.warehouses.find().pretty()` finden

4. Arbeitsschritt

Zuerst wurde ein MongoDB Docker Image gepullt und ein Container gestartet:

```
docker pull mongo
```

```
docker run -d -p 27017:27017 --name mongo mongo
```

Danach kann man in die Shell reingehen:

```
docker exec -it mongo bash
```

Hier kann man eine neue datenbank mit `use warehouse_db`. Man kann auch alle Datenbanken mit `show dbs` anzeigen. Jetzt kann die Spring-Implementation begonnen werden.

Zuerst wird die `application.properties`-Datei angepasst:

```
spring.application.name=DezsysMongoDB2

# MongoDB Konfiguration
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=warehouse_db

# Server-Port
server.port=8080
```

Es wurden auch die entsprechenden dependencies hinzugefügt:

```
implementation 'org.springframework.boot:spring-boot-starter-data-mongodb'
implementation 'org.springframework.boot:spring-boot-starter-validation'
implementation 'org.springframework.boot:spring-boot-starter-web'
compileOnly 'org.projectlombok:lombok'
developmentOnly 'org.springframework.boot:spring-boot-devtools'
annotationProcessor 'org.projectlombok:lombok'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.2.0'
```

Optional kann man auch eine Config-Datei für Swagger-UI erstellen:

```
@Configuration
public class SwaggerConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
                .info(new Info()
                        .title("Warehouse Management API")
                        .version("1.0")
                        .description("API zur Verwaltung von Lagerstandorten und Produkten"));
    }
}
```

Zuerst wird die Produkt “Entity” erstellt (eigentlich eine Helper-Klasse)

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Product {
    private String productId;
    private String name;
    private String category;
    private int quantity;
}
```

Danach wird das eigentliche “Document-Object” erstellt:

```
@Document(collection = "warehouses")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Warehouse {
    @Id
    private String id;
    private String name;
    private String location;
    private List<Product> products;
}
```

Dafür wird entsprechend auch ein Repository erstellt:

```
public interface WarehouseRepository extends MongoRepository<Warehouse, String> {
    List<Warehouse> findByProductsProductId(String productId);
}
```

Danach kommt die Service-Implementierung:

```
@Service
public class WarehouseService {
    private final WarehouseRepository warehouseRepository;

    public WarehouseService(WarehouseRepository warehouseRepository) {
        this.warehouseRepository = warehouseRepository;
    }
    // Lagerstandort hinzufügen
    public Warehouse addWarehouse(Warehouse warehouse) {
        return warehouseRepository.save(warehouse);
    }
    // Alle Lagerstandorte abrufen
    public List<Warehouse> getAllWarehouses() {
        return warehouseRepository.findAll();
    }
    // Lagerstandort nach ID abrufen
    public Optional<Warehouse> getWarehouseById(String id) {
        return warehouseRepository.findById(id);
    }
    // Lagerstandort löschen
    public void deleteWarehouse(String id) {
        warehouseRepository.deleteById(id);
    }
    // Produkt zu Lagerstandort hinzufügen
    public Warehouse addProductToWarehouse(String warehouseId, Product product) {
        Warehouse warehouse = warehouseRepository.findById(warehouseId)
                .orElseThrow(() -> new RuntimeException("Warehouse not found"));
        warehouse.getProducts().add(product);
        return warehouseRepository.save(warehouse);
    }
    // Alle Produkte aus allen Lagerstandorten abrufen
    public List<Product> getAllProducts() {
        return warehouseRepository.findAll().stream()
                .flatMap(w -> w.getProducts().stream())
                .toList();
    }
    // Produkt nach ID abrufen
    public List<Warehouse> getProductById(String productId) {
        return warehouseRepository.findByProductsProductId(productId);
    }
    // Produkt aus Lagerstandort löschen
```

```
    public void deleteProductFromWarehouse(String warehouseId, String productId) {
        Warehouse warehouse = warehouseRepository.findById(warehouseId)
                .orElseThrow(() -> new RuntimeException("Warehouse not found"));
        warehouse.getProducts().removeIf(p -> p.getProductId().equals(productId));
        warehouseRepository.save(warehouse);
    }}
```

Weil es keine explizite Produkte gibt die unabhängig von den Warehouses existieren können, braucht man keine extra Entity für die Produkte. Alle Methoden für Produkte entkapseln die Liste an Produkten, modifizieren die und speichern das neue Warehouse in der Datenbank.

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

    private final WarehouseService warehouseService;

    public WarehouseController(WarehouseService warehouseService) {
        this.warehouseService = warehouseService;
    }
    // POST /warehouse: Lagerstandort hinzufügen
    @PostMapping("/warehouse")
    public ResponseEntity<Warehouse> addWarehouse(@RequestBody Warehouse warehouse) {
        return ResponseEntity.ok(warehouseService.addWarehouse(warehouse));
    }
    // GET /warehouse: Alle Lagerstandorte abrufen
    @GetMapping("/warehouse")
    public ResponseEntity<List<Warehouse>> getAllWarehouses() {
        return ResponseEntity.ok(warehouseService.getAllWarehouses());
    }
    // GET /warehouse/{id}: Lagerstandort nach ID abrufen
    @GetMapping("/warehouse/{id}")
    public ResponseEntity<Warehouse> getWarehouseById(@PathVariable String id) {
        return warehouseService.getWarehouseById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }
    // DELETE /warehouse/{id}: Lagerstandort löschen
    @DeleteMapping("/warehouse/{id}")
    public ResponseEntity<Void> deleteWarehouse(@PathVariable String id) {
        warehouseService.deleteWarehouse(id);
        return ResponseEntity.noContent().build();
    }
    // POST /product: Produkt zu Lagerstandort hinzufügen
    @PostMapping("/product/{warehouseId}")
    public ResponseEntity<Warehouse> addProductToWarehouse(
            @PathVariable String warehouseId,
            @RequestBody Product product) {
        return ResponseEntity.ok(warehouseService.addProductToWarehouse(warehouseId, product));
    }
    // GET /product: Alle Produkte abrufen
    @GetMapping("/product")
    public ResponseEntity<List<Product>> getAllProducts() {
        return ResponseEntity.ok(warehouseService.getAllProducts());
    }
    // GET /product/{id}: Produkt nach ID und Lagerstandorten abrufen
    @GetMapping("/product/{id}")
    public ResponseEntity<List<Warehouse>> getProductById(@PathVariable String id) {
        return ResponseEntity.ok(warehouseService.getProductById(id));
    }
    // DELETE /product/{warehouseId}/{productId}: Produkt aus Lagerstandort löschen
    @DeleteMapping("/product/{warehouseId}/{productId}")
    public ResponseEntity<Void> deleteProductFromWarehouse(
            @PathVariable String warehouseId,
            @PathVariable String productId) {
        warehouseService.deleteProductFromWarehouse(warehouseId, productId);
```

```
        return ResponseEntity.noContent().build();
    }}
```

Der Controller ist nur für die Endpoints zuständig und beinhaltet keine Business-Logic, sprich es nutzt nur den Service.

```
@Component
public class DataSeeder implements CommandLineRunner {

    private final WarehouseRepository warehouseRepository;

    public DataSeeder(WarehouseRepository warehouseRepository) {
        this.warehouseRepository = warehouseRepository;
    }
    @Override
    public void run(String... args) {
        warehouseRepository.deleteAll();

        // 10 Products in 3 Categories
        List<Product> products = Arrays.asList(
                new Product("p1", "Schraube", "Baumaterial", 100),
                new Product("p2", "Nagel", "Baumaterial", 200),
                new Product("p3", "Hammer", "Werkzeug", 50),
                new Product("p4", "Säge", "Werkzeug", 30),
                new Product("p5", "Bohrer", "Werkzeug", 40),
                new Product("p6", "Holzplatte", "Baumaterial", 80),
                new Product("p7", "Ziegel", "Baumaterial", 500),
                new Product("p8", "Farbeimer", "Malerei", 60),
                new Product("p9", "Pinsel", "Malerei", 120),
                new Product("p10", "Rolle", "Malerei", 90)
        );
        Warehouse warehouse1 = new Warehouse("warehouse1", "Lager Nord", "Berlin",
products.subList(0, 5));
        Warehouse warehouse2 = new Warehouse("warehouse2", "Lager Süd", "München",
products.subList(5, 10));

        warehouseRepository.saveAll(Arrays.asList(warehouse1, warehouse2));

        System.out.println("Seeded 10 products across 2 warehouses.");
    }}
```

Dieser Seeder erstellt beim Starten 2 Warehouses mit jeweils 10 Produkten die 3 Kategorien haben.

5 Mongo-CRUD Operationen

Create: Ein neues Produkt zum Warehouse hinzufügen

```
db.warehouses.updateOne(
  { _id: "warehouse1" },
  { $push: { products: { productId: "p10", name: "Schlüssel", category: "Werkzeug", quantity:
50 } } }
)
```

Output:

```
{
  acknowledged: true,
  insertedId: null,
```

```
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Read: Alle Produkte von einem Warehouse auslesen

```
db.warehouses.find(
  { _id: "warehouse1" },
  { products: 1, _id: 0 }
).pretty()
```

`Update`: Die quantity ändern:

```
db.warehouses.updateOne(
  { "products.productId": "p10" },
  { $set: { "products.$.quantity": 120 } }
)
```

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Delete: Product aus nem Warehouse löschen

```
db.warehouses.updateOne(
  { _id: "warehouse1" },
  { $pull: { products: { productId: "p10" } } }
)
```

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

3 Fragen

1. In welchen Lagerstandorten ist das Produkt mit der ID `p3` vorhanden?

```
db.warehouses.find(
  { "products.productId": "p3" },
```

```
  { name: 1, location: 1, "products.$": 1, _id: 0 }
).pretty()
```

2. Welche Produktkategorie hat den höchsten Gesamtbestand über alle Lagerstandorte hinweg?

```
db.warehouses.aggregate([
  { $unwind: "$products" },
  { $group: { _id: "$products.category", totalQuantity: { $sum: "$products.quantity" } }
},
  { $sort: { totalQuantity: -1 } },
  { $limit: 1 }
])
```

3. Welche Produkte werden in mehr als einem Lagerstandort geführt?

```
db.warehouses.aggregate([
  { $unwind: "$products" },
  { $group: { _id: "$products.productId", warehouses: { $addToSet: "$name" }, count: {
$sum: 1 } } },
  { $match: { count: { $gt: 1 } } },
  { $project: { productId: "$_id", warehouses: 1, _id: 0 } }
])
```

5. Zusammenfassung

Dieses Projekt zeigt die Entwicklung einer dokumentenorientierten Middleware mit **MongoDB und Spring Boot**. Es ermöglicht die Verwaltung von Lagerstandorten und deren Produkten über eine REST-API. Die Implementierung umfasst CRUD-Operationen, die Speicherung mehrerer Standorte sowie Swagger UI zur Visualisierung.

6. Quellen

- MongoDB Dokumentation (https://www.mongodb.com/docs/)
- Spring Boot Dokumentation (https://spring.io/projects/spring-boot)
- Spring Data MongoDB (https://spring.io/projects/spring-data-mongodb)
- Java SE Dokumentation (https://docs.oracle.com/en/java/javase/)