Middleware Engineering "REST and Data Formats"

Mit Java Springboot Webendpunkte

Verfasser: Ramis Ekici, 4CHIT

Datum: 25.09.2025

1. Aufgabenstellung

Ziel dieser Übung ist die Entwicklung einer Java-Spring-Boot-Anwendung, die ein Lagerverwaltungssystem (Warehouse) simuliert und Daten über REST-Endpunkte bereitstellt. Die Daten sollen in unterschiedlichen Formaten (JSON, XML) abrufbar sein. Zusätzlich soll eine einfache Weboberfläche zur Anzeige und Filterung der Lagerbestände mit Thymeleaf implementiert werden.

Funktionen:

- Bereitstellung eines REST-Endpunkts für Lagerdaten (/warehouse/{id}/json und /warehouse/{id}/xml).
- Simulation von Lager- und Produktdaten.
- Darstellung der Lagerdaten in einer Tabelle im Browser.
- Filterung nach Produktname und Produktkategorie.

2. Implementierung

2.1 Architektur

SpringBootApplication

WarehouseApplication

Die `WarehouseApplication` ist der Einstiegspunkt und startet die Spring Boot-Anwendung, stellt Beans wie `RestTemplate` bereit und führt beim Start optional eine kleine Testausgabe der Lagerdaten auf der Konsole aus. Initialisiert wichtige Komponenten und stellt zentrale Konfigurationen bereit

Model

WarehouseData

Datenklassen für Lager (`WarehouseData`) und Produkte (`ProductData`). Diese Klasse enthält die entprechende Attributen für ein Warehous und logischerweise als ein Attribut als Set Collection die Produkte.

```
private String warehouseID;
private String warehouseName;
private String timestamp;
private String warehouseAddress;
private String warehousePostalCode;
private String warehouseCity;
private String warehouseCountry;
private Set<ProductData> products;
```

Getter und Methoden und Konstruktor wurde mit Intellij generate sehr schnell und praktisch erstellt. Am Ende ebenso für Testfallen und alles andere eine String dartstellung von Objekten.

```
public WarehouseData() {
        this.timestamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new
Date());
}

public Set<ProductData> getProducts() {
    return products;
}

public void setProducts(Set<ProductData> products) {
    this.products = products;
}

public String getWarehouseAddress() {
    return warehouseAddress;
}

@Override
    public String toString() {
        return "WarehouseData{" +
                "warehouseID='" + warehouseID + '\'' +
                ", warehouseName='" + warehouseName + '\'' +
                ", timestamp='" + timestamp + '\'' +
                ", warehouseAddress='" + warehouseAddress + '\'' +
                ", warehousePostalCode='" + warehousePostalCode + '\'' +
                ", warehouseCity='" + warehouseCity + '\'' +
                ", warehouseCountry='" + warehouseCountry + '\'' +
                '}';
    }
```

ProductData

Natürlich wird auch später eine Klasse für Produkten gebraucht, die wir dann später in Warehouse (Lager) sammeln.

Eine sehr simple Klassen mit entsprechenden Attributen und vom Intellij generierte Konstruktor und toString Methode.

```
private String productID;
    private String productName;
    private String productCategory;
    private String productUnit;
    private int productQuantity;
    public ProductData(String productID, String productName, String productCategory,
String productUnit, int productQuantity) {
        this.productID = productID;
        this.productName = productName;
        this.productCategory = productCategory;
        this.productUnit = productUnit;
        this.productQuantity = productQuantity;
    }
    @Override
    public String toString() {
        return "ProductData{" +
                "productID='" + productID + '\'' +
                ", productName='" + productName + '\'' +
                ", productCategory=" + productCategory +
                ", productUnit='" + productUnit + '\'' +
                '}';
    }
```

WarehouseSimulation

Die Klasse `WarehouseSimulation` erzeugt **Beispiel-Lagerdaten** mit festen Produkten (und optional zufälligen Werten) für **Test- und Demonstrationszwecke**.

Ihre Rolle ist es, im **Model-Bereich** als **Datenquelle/Mock** zu dienen, damit Controller und Service ohne echte Datenbank arbeiten können.

Ist also hauptsächlich für das Zurückgeben von Beispiel-Lagerdaten zuständig.

```
public class WarehouseSimulation {
    private double getRandomDouble( int inMinimum, int inMaximum ) {

        double number = ( Math.random() * ( (inMaximum-inMinimum) + 1 )) + inMinimum;
        double rounded = Math.round(number * 100.0) / 100.0;
        return rounded;
    }
    private int getRandomInt( int inMinimum, int inMaximum ) {

        double number = ( Math.random() * ( (inMaximum-inMinimum) + 1 )) + inMinimum;
        Long rounded = Math.round(number);
        return rounded.intValue();
    }

    /**
     * Liefert Beispiel-Lagerdaten für ein gegebenes Lager anhand der Lager-ID.
     * Enthält feste Produkte mit Beispielwerten.
     *
     * @param inID ID des Lagers
     * @return Lagerdaten als WarehouseData-Objekt
     */
    public WarehouseData getData( String inID ) {
        WarehouseData data = new WarehouseData();
        data.setWarehouseID( inID );
        data.setWarehouseName( "Linz Bahnhof" );
        data.setWarehouseAddress("Bahnhofsstrasse 27/9");
        data.setWarehousePostalCode("Linz");
        data.setWarehouseCity("Linz");
        data.setWarehouseCountry("Austria");

        HashSet<ProductData> products = new HashSet<>();
        products.add(new ProductData("00-443175", "Bio Orangensaft Sonne", "Getraenk",
"Packung 1L", 2500));
        products.add(new ProductData("00-871895", "Bio Apfelsaft Gold",    "Getraenk",
"Packung 1L", 3420));
        products.add(new ProductData("01-926885", "Ariel Waschmittel
Color","Waschmittel", "Packung 3KG", 478));
        products.add(new ProductData("00-316253", "Persil Discs Color",
"Waschmittel", "Packung 700G", 1430));
        data.setProducts(products);
        return data;
    }
}
```

Service

Enthält die Geschäftslogik und Filtermethoden (`WarehouseService`).

WarehouseService

Der Service kümmert sich um die **Logik und Datenaufbereitung**, damit der Controller nur die Ergebnisse weitergeben muss.

- **@Service (Annotation)**: Markiert die Klasse als *Service-Komponente*. Spring erkennt sie automatisch und verwaltet ihre Lebenszyklen (Dependency Injection möglich).
- Der **Controller** ruft diese Klasse auf, um **Daten und Logik** zu verarbeiten.
- Der Service ruft wiederum die **Datenquelle** auf (hier: `WarehouseSimulation`) und bereitet die Daten so auf, dass der Controller nur fertige Ergebnisse bekommt.
- In Insgesamten gehören diese Verantwortlichkeiten zum Service: **Geschäftslokig, Datenbereitstellung und Abstraktion.**

In Insgesamten schaut die Klasse mit Methoden für Datenkriegen und Filterung, folgendermaßen aus:

```java
@Service
public class WarehouseService {

    /**
     * Liefert einen Begrüßungstext für ein gegebenes Modul.
     *
     * @param inModule Name des Moduls
     * @return Begrüßungstext als String
     */
    public String getGreetings( String inModule ) {
        return "Greetings from " + inModule;
    }

    /**
     * Liefert die Lagerdaten für ein bestimmtes Lager anhand der Lager-ID.
     *
     * @param inID ID des Lagers
     * @return Lagerdaten als WarehouseData-Objekt
     */
    public WarehouseData getWarehouseData( String inID ) {
        WarehouseSimulation simulation = new WarehouseSimulation();
        return simulation.getData( inID );
    }

    /**
     * Liefert gefilterte Lagerdaten anhand von Produktkategorie und Produktname.
     * Es werden nur Produkte zurückgegeben, die den Filterkriterien entsprechen.
     *
     * @param productCategorie Kategorie des Produkts (null oder leer für keine
Filterung)
     * @param productName Name des Produkts (null oder leer für keine Filterung)
     * @return Gefilterte Lagerdaten als WarehouseData-Objekt
     */
    public WarehouseData getWarehouseDataFiltered(String productCategorie, String
productName) {
        WarehouseData data = getWarehouseData("001"); // oder anhand Location
dynamisch abrufen

        if (data.getProducts() != null) {
            Set<ProductData> filtered = data.getProducts().stream()
                    .filter(p -> productCategorie == null ||
productCategorie.isEmpty()
                            ||
p.getProductCategory().toLowerCase().contains(productCategorie.toLowerCase()))
                    .filter(p -> productName == null || productName.isEmpty()
                            ||
p.getProductName().toLowerCase().contains(productName.toLowerCase()))
                    .collect(Collectors.toCollection(HashSet::new));
            data.setProducts(filtered);
        }
```

```
        return data;
    }
}
```

View

welcome.html

Diese HTML-Datei ist eine **statische Startseite** der Anwendung. Sie zeigt eine Begrüßung und enthält einen Link zur Ansicht des Lagers *001*.

Da sie im Ordner `src/main/resources/static` liegt, wird sie von Spring Boot **direkt als statische Ressource** ausgeliefert (ohne Controller). Das ist nützlich für einfache Startseiten oder feste Inhalte, die nicht dynamisch über Thymeleaf/Controller erzeugt werden müssen. Es hat den Endpunk **/warehouse/view/001/**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Warehouse Start</title>
</head>
<body>
<h1>Willkommen zur Warehouse-Anwendung</h1>
<p>
    <a href="/warehouse/view/001">Warehouse 001 ansehen</a>
</p>
</body>
</html>
```

warehouse-view.html

Die Datei wird **nicht direkt** ausgeliefert wie eine statische Ressource, sondern vom **Controller** aufgerufen. Spring Boot rendert sie mit Thymeleaf und ersetzt die **Platzhalter/Variablen** (z. B. `${warehouse}`) durch echte Daten, die im Controller ins `Model` gelegt wurden.

- Dient zur **dynamischen Darstellung** von Lagerdaten im Browser.
- Arbeitet eng mit dem `WarehouseViewController` zusammen, der die Daten liefert.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Warehouse View</title>
</head>
<body>
<h1 th:text="${warehouse.warehouseName}">Warehouse Name</h1>

<form th:action="@{/warehouse/view}" method="get">
  Category: <input type="text" name="location" />
  Produktname: <input type="text" name="productName" />
  <button type="submit">Filtern</button>
</form>

<table border="1">
  <tr>
    <th>ID</th><th>Name</th><th>Category</th><th>Unit</th><th>Quantity</th>
  </tr>
  <!-- Schleife über alle Produkte im Warehouse-Objekt -->
  <tr th:each="p : ${warehouse.products}">
    <td th:text="${p.productID}">ID</td>
    <td th:text="${p.productName}">Name</td>
    <td th:text="${p.productCategory}">Category</td>
    <td th:text="${p.productUnit}">Unit</td>
    <td th:text="${p.productQuantity}">Quantity</td>
  </tr>
</table>
</body>
</html>
```

Controller

Controller: Steuert die Endpunkte und die View-Darstellung (`WarehouseController` , `WarehouseViewController`).

WarehouseController

- Besser gesagt spielt diese Klasse eine Rolle als REST-Controller. Also sie stellt **REST-Endpunkten** für Lagerdaten bereit.
- Er nimmt HTTP-Anfragen entgegen (`/warehouse/{id}/json` , `/warehouse/{id}/xml`) und gibt die Daten im gewünschten Format zurück.

Für diese Klasse werden folgende Annotationen benötigt:

- @ RestControlling
    - Markiert die Klasse als REST-Controller. Rückgabewerte von Methoden werden direkt als HTTP-Response (z. B. JSON oder XML) gesendet.
- @ Autorwired
    - Automatische Abhängigkeitsinjektion. Spring erstellt das Objekt (z. B. `WarehouseService`) und stellt es der Klasse zur Verfügung.
- @ RequestMapping
    - Definiert die URL und das Ausgabeformat (z. B. `/warehouse/{id}/json` mit `produces = MediaType.APPLICATION_JSON_VALUE`).
- @ PathVariable
    - Bindet einen Wert aus der URL (z. B. `{id}`) an eine Methodenvariable.
- WarehouseService
    - Die Service-Klasse liefert dem Controller die fertigen Daten und übernimmt die gesamte Geschäftslogik, sodass der Controller sich nur um die HTTP-Anfrage und -Antwort kümmern muss.

Am Ende ensteht so eine Klasse:

```java
@RestController
public class WarehouseController {

    @Autowired
    private WarehouseService service;

    /**
     * Hauptseite der Anwendung. Gibt eine einfache HTML-Übersicht mit Links zu
Beispiel-Lagerdaten zurück.
     *
     * @return HTML-String mit Hinweisen auf verfügbare Lagerdaten-Endpunkte
     */
    @RequestMapping("/")
    public String warehouseMain() {
        String mainPage = "This is the warehouse application! (DEZSYS_WAREHOUSE_REST)
<br/><br/>" +
                          "<a href='http://localhost:8080/warehouse/001/json'>Link to
warehouse/001/json</a><br/>" +
                          "<a href='http://localhost:8080/warehouse/001/xml'>Link to
warehouse/001/xml</a><br/>"
                          ;
        return mainPage;
    }

    /**
     * Liefert Lagerdaten im JSON-Format für ein gegebenes Lager.
     *
     * @param inID ID des Lagers, dessen Daten abgerufen werden sollen
     * @return Lagerdaten als WarehouseData-Objekt
     */
    @RequestMapping(value="/warehouse/{inID}/json", produces =
MediaType.APPLICATION_JSON_VALUE)
    public WarehouseData warehouseData( @PathVariable String inID ) {
        return service.getWarehouseData( inID );
    }

    /**
     * Liefert Lagerdaten im XML-Format für ein gegebenes Lager.
     *
     * @param inID ID des Lagers, dessen Daten abgerufen werden sollen
     * @return Lagerdaten als WarehouseData-Objekt
     */
    @RequestMapping(value="/warehouse/{inID}/xml", produces =
MediaType.APPLICATION_XML_VALUE)
    public WarehouseData warehouseDataXML( @PathVariable String inID ) {
        return service.getWarehouseData( inID );
    }
}
```

WarehouseViewController

Diese Klasse ist ein **Spring MVC Controller**, der Lagerdaten im Browser über **Thymeleaf-Views** darstellt.

Sie holt die Daten vom **WarehouseService**, übergibt sie ans **Model** und rendert die HTML-Seite `warehouse-view`.

```java
@Controller
public class WarehouseViewController {
    /** Service zur Verwaltung und Bereitstellung von Lagerdaten */
    private final WarehouseService service;

    public WarehouseViewController(WarehouseService service) {
        this.service = service;
    }

    /**
     * Zeigt die Detailansicht eines Lagers anhand der Lager-ID an.
     *
     * @param id Lager-ID aus dem Pfad
     * @param model Model zur Übergabe der Daten an die View
     * @return Name der Thymeleaf-Template-Datei
     */
    @GetMapping("/warehouse/view/{id}")
    public String showWarehouse(@PathVariable String id, Model model) {
        WarehouseData data = service.getWarehouseData(id); // direkt
        model.addAttribute("warehouse", data);
        return "warehouse-view";
    }

    /**
     * Zeigt die Lageransicht mit optionalen Filterkriterien an.
     * Filterbar nach Produktkategorie und Produktname.
     *
     * @param location Optional: Kategorie des Produkts (früher "Standort")
     * @param productName Optional: Name des Produkts
     * @param model Model zur Übergabe der Daten an die View
     * @return Name der Thymeleaf-Template-Datei
     */
    @GetMapping("/warehouse/view")
    public String showWarehouse(
            @RequestParam(required = false) String location,
            @RequestParam(required = false) String productName,
            Model model) {

        WarehouseData data = service.getWarehouseDataFiltered(location, productName);
        model.addAttribute("warehouse", data);
        return "warehouse-view";
    }

}
```

2.2 Endpunkte

- `GET /warehouse/{id}/json` → gibt Lagerdaten im JSON-Format zurück.

- `GET /warehouse/{id}/xml` → gibt Lagerdaten im XML-Format zurück.
- `GET /warehouse/view/{id}` → zeigt Lagerdaten im Browser (Thymeleaf).
- `GET /warehouse/view?productName=xyz&productCategorie=abc` → zeigt gefilterte Tabelle im Browser.

2.3 Filterung

Die Methode `getWarehouseDataFiltered(String productCategorie, String productName)` filtert die Produkte anhand der übergebenen Parameter:

- **productCategorie** → sucht in der Kategorie des Produkts.
- **productName** → sucht nach Teilstrings im Produktnamen.

Beide Filter können kombiniert oder einzeln genutzt werden.

2.4 Technologien

- **Spring Boot** für REST-API und Webserver.
- **Thymeleaf** als Template Engine für HTML-Seiten.
- **Lombok** für vereinfachte Getter/Setter.
- **Gradle** für Projekt- und Build-Management.
- **JavaDoc** für die vom Gradle generierte Java Dokumente

Dependencies sehen genau so aus:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-rest'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    compileOnly("org.projectlombok:lombok:1.18.42")
    annotationProcessor("org.projectlombok:lombok:1.18.42")
    testCompileOnly("org.projectlombok:lombok:1.18.42")
    testAnnotationProcessor("org.projectlombok:lombok:1.18.42")
    implementation 'com.fasterxml.jackson.dataformat:jackson-dataformat-xml'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
}


tasks.javadoc {
    options.encoding = 'UTF-8'
    options.author = true
    options.version = true
    options.links("https://docs.oracle.com/en/java/javase/21/docs/api/") // Optional: externe Java API Links
}
```

3. Quellen

[1] S. Riffaldi, „Answer to ‚spring boot: what dependencies are needed make a simple spring boot app with domain objects, services and web controllers (similar to grails?)‘“, Stack Overflow. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://stackoverflow.com/a/77390421

[2] M. Reiche, „Answer to ‚spring boot: what dependencies are needed make a simple spring boot app with domain objects, services and web controllers (similar to grails?)‘“, Stack Overflow. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://stackoverflow.com/a/77391138

[3] J. Little, „spring boot: what dependencies are needed make a simple spring boot app with domain objects, services and web controllers (similar to grails?)“, Stack Overflow. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://stackoverflow.com/q/77384136

[4] S. D’sa, „Answer to ‚spring boot: what dependencies are needed make a simple spring boot app with domain objects, services and web controllers (similar to grails?)‘“, Stack Overflow. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://stackoverflow.com/a/77384214

[5] „Tutorial: Using Thymeleaf“. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

[6] „Spring Boot“. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://spring.io/projects/spring-boot

[7] „Getting Started | Consuming a RESTful Web Service“, Getting Started | Consuming a RESTful Web Service. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://spring.io/guides/gs/consuming-rest

[8] „Getting Started | Building an Application with Spring Boot“, Getting Started | Building an Application with Spring Boot. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://spring.io/guides/gs/spring-boot

[9] „Getting Started | Building a RESTful Web Service“, Getting Started | Building a RESTful Web Service. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://spring.io/guides/gs/rest-service

[10] „Actuator REST API :: Spring Boot“. Zugegriffen: 25. September 2025. [Online]. Verfügbar unter: https://docs.spring.io/spring-boot/api/rest/actuator/index.html