

Spring Boot, Teil 2

Eine REST-API mit Persistenz, Validierung und Fehlerbehandlung

Informatik – 3. Semester

1 Ziel & Aufbau

3 DTOs & Validierung

5 REST-Controller

7 Konfiguration & Profile

2 Entity & Repository

4 Service & Transaktionen

6 Fehlerbehandlung

8 Testen (Ausblick)

Ein Durchstich von der Datenbank bis zur JSON-Antwort

Kennt ihr schon

- REST-Vorlesung: Ressourcen, Verben, Statuscodes, JSON
- Teil 1: DI, Beans, `@Controller` mit Thymeleaf
- Teil 1: `@Entity`, `JpaRepository`, abgeleitete Queries
- Teil 1: Spring Security im Überblick

Kommt heute dazu

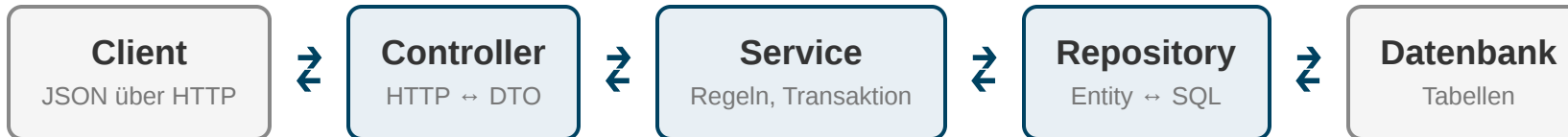
- JSON-API mit `@RestController`
- DTOs statt Entities an der API-Grenze
- Service-Schicht mit Transaktionen
- Fehler als saubere Statuscodes
- Profile für H2 und PostgreSQL
- Wie man das Ganze testet

Heute zählt, was ihr für das Back-End eures MVP braucht: eine Ressource sauber von der Datenbank bis zur Antwort im Browser.

Ein Team-Board mit Aufgaben (Tasks). Aus der REST-Vorlesung übernehmen wir zuerst den **Vertrag**:

Methode	Pfad	Wirkung	Statuscodes
GET	/api/teams/{teamId}/tasks	Tasks eines Teams	200 · 404
GET	/api/tasks/{id}	eine Task	200 · 404
POST	/api/teams/{teamId}/tasks	Task anlegen	201 · 400 · 404 · 409
PUT	/api/tasks/{id}	Task ändern	200 · 400 · 404
DELETE	/api/tasks/{id}	Task löschen	204 · 404

Erst den Vertrag festlegen, dann implementieren: So kann euer Front-End parallel gegen dieselbe Tabelle arbeiten.



- An jeder Grenze wechselt die Darstellung: **JSON** → **DTO** → **Entity** → **Zeile** und zurück
- Jede Schicht spricht nur mit ihrem direkten Nachbarn
- Pakete **nach Fachlichkeit** schneiden, nicht nach Schicht: So arbeitet jede:r an einem Feature, ohne sich ständig in die Quere zu kommen

```
de.th1.taskboard
├── TaskboardApplication.java
├── task/
│   ├── Task.java
│   ├── TaskRepository.java
│   ├── TaskService.java
│   ├── TaskController.java
│   └── dto/
├── team/
└── common/    (Exceptions, Advice)
```

- 1 Ziel & Aufbau
- 3 DTOs & Validierung
- 5 REST-Controller
- 7 Konfiguration & Profile

2 Entity & Repository

- 4 Service & Transaktionen
- 6 Fehlerbehandlung
- 8 Testen (Ausblick)

Das Datenmodell als Java-Klassen

```
@Entity
public class Task {
    @Id @GeneratedValue
    private Long id;
    private String title;
    private LocalDate dueDate;
    @Enumerated(EnumType.STRING)
    private TaskStatus status;
    @ManyToOne(fetch = FetchType.LAZY)
    private Team team;

    protected Task() {} // für JPA

    public Task(String title, LocalDate dueDate,
        Team team) {
        this.title = title;
        this.dueDate = dueDate;
        this.team = team;
        this.status = TaskStatus.OPEN;
    }
    // Getter, update(...)
}
```

- `@Enumerated(String)` speichert den Enum als Text, nicht als Ordinalzahl
- `@ManyToOne` legt die Fremdschlüsselspalte `team_id` an
- `LAZY` : Das Team wird erst geladen, wenn es gebraucht wird (Default wäre `EAGER`)
- JPA braucht einen parameterlosen Konstruktor; `protected` hält ihn aus eurem Code fern

```
public interface TaskRepository
    extends JpaRepository<Task, Long> {

    List<Task> findByTeamIdOrderByDueDateAsc(
        Long teamId);

    boolean existsByTeamIdAndTitle(
        Long teamId, String title);
}
```

- `TeamId` navigiert über die Beziehung zu `team.id`
- `existsBy...` liefert nur ja/nein: ideal für Regelprüfungen

```
@Entity
public class Team {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @OneToMany(mappedBy = "team")
    private List<Task> tasks = new ArrayList<>();
}
```

- Der Fremdschlüssel gehört der `@ManyToOne`-Seite
- `mappedBy` spiegelt die Beziehung nur, ändert nichts in der DB

Bidirektional plus Entity als JSON ergibt eine Endlosschleife: `Team → tasks → team → tasks ...` Das ist Grund Nr. 1 für DTOs.

1 Ziel & Aufbau

3 DTOs & Validierung

5 REST-Controller

7 Konfiguration & Profile

2 Entity & Repository

4 Service & Transaktionen

6 Fehlerbehandlung

8 Testen (Ausblick)

Was über die Leitung geht, ist nicht euer Datenmodell

Over-Posting

Der Client schickt `id` oder `status` mit und überschreibt Felder, die er nie setzen darf.

Datenleck

Interne Felder wie `passwordHash` landen ungefiltert im Browser.

Rekursion & Lazy

Beziehungen erzeugen Endlosschleifen oder Fehler beim Nachladen außerhalb der Transaktion.

Kopplung

Jede Änderung am DB-Schema bricht sofort das Front-End.

DTO (Data Transfer Object)

Ein schlankes Objekt nur für die Übertragung: pro Richtung und Anwendungsfall genau die Felder, die über die Leitung gehen sollen, ohne Logik und ohne JPA-Annotationen.

```
public record CreateTaskRequest(  
    @NotBlank @Size(max = 100) String title,  
    @FutureOrPresent LocalDate dueDate) {}  
  
public record UpdateTaskRequest(  
    @NotBlank @Size(max = 100) String title,  
    LocalDate dueDate,  
    @NotNull TaskStatus status) {}  
  
public record TaskResponse(Long id, String title, LocalDate dueDate,  
                           TaskStatus status, Long teamId) {  
    public static TaskResponse from(Task t) {  
        return new TaskResponse(t.getId(), t.getTitle(), t.getDueDate(),  
                                t.getStatus(), t.getTeam().getId());  
    }  
}
```

- Records sind unveränderlich, `equals` / `toString` gibt es gratis, Jackson liest und schreibt sie direkt
- Request-DTOs ohne `id` und ohne `status` beim Anlegen: Over-Posting ist damit ausgeschlossen

Annotation	prüft
@NotNull	Wert vorhanden
@NotBlank	Text nicht leer
@Size(min, max)	Länge von Text/Liste
@Min / @Max / @Positive	Zahlenbereich
@Email / @Pattern	Format
@Past / @FutureOrPresent	Datum

- Auslöser ist `@Valid` am Body-Parameter der Controller-Methode
- Verstoß → 400, **bevor** eure Methode läuft
- Starter `spring-boot-starter-validation` ist **nicht** in Spring Web enthalten. Ohne ihn wird still gar nichts geprüft

Bean Validation prüft die **Form** einer Eingabe. Regeln, für die man die Datenbank braucht (Titel im Team eindeutig), prüft der **Service**.

- Nehmt die wichtigste Ressource eures MVP (z. B. Buchung, Rezept, Termin)
- Welche Felder schickt der Client beim **Anlegen**? Welche bekommt er **zurück**?
- Welche Felder darf der Client **nie** selbst setzen?
- Welche Constraints gelten (Pflicht, Länge, Bereich, Datum)?

3 Min

- 1 Ziel & Aufbau
- 3 DTOs & Validierung
- 5 REST-Controller
- 7 Konfiguration & Profile

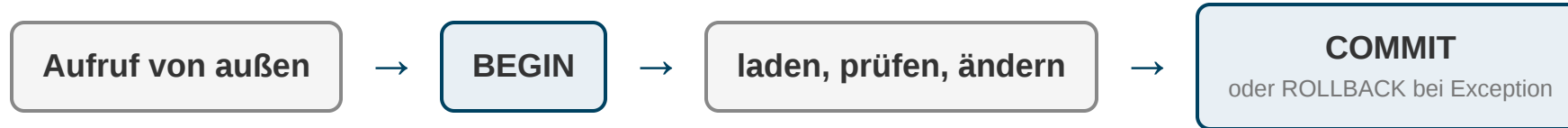
- 2 Entity & Repository
- 4 Service & Transaktionen**
- 6 Fehlerbehandlung
- 8 Testen (Ausblick)

Hier wohnt die Geschäftslogik

```
@Service @Transactional
public class TaskService {
    private final TaskRepository tasks;
    private final TeamRepository teams;
    // + Konstruktor-Injection (Teil 1); findById, findByTeam, delete analog

    public TaskResponse create(Long teamId, CreateTaskRequest req) {
        Team team = teams.findById(teamId)
            .orElseThrow(() -> new NotFoundException("Team", teamId));
        if (tasks.existsByTeamIdAndTitle(teamId, req.title())) {
            throw new ConflictException("Titel im Team schon vergeben");
        }
        Task task = new Task(req.title(), req.dueDate(), team);
        return TaskResponse.from(tasks.save(task));
    }

    public TaskResponse update(Long id, UpdateTaskRequest req) {
        Task task = tasks.findById(id)
            .orElseThrow(() -> new NotFoundException("Task", id));
        task.update(req.title(), req.dueDate(), req.status());
        return TaskResponse.from(task); // kein save() nötig
    }
}
```



- **Alles oder nichts:** Scheitert ein Schritt, wird keine der Änderungen geschrieben
- **Rollback** passiert per Default bei `RuntimeException`, nicht bei checked Exceptions
- **Dirty Checking:** Geladene Entities werden beobachtet; Änderungen schreibt JPA beim Commit selbst. Darum braucht `update` kein `save()`
- `@Transactional(readOnly = true)` für reine Lesemethoden

Die Annotation wirkt über einen Spring-Proxy. Ruft der Service eine eigene Methode per `this.x()` auf, greift sie nicht.

Symptom im Code	Besser
Controller ruft <code>TaskRepository</code> direkt auf	Controller spricht nur mit dem Service
<code>if</code> -Geschäftsregeln im Controller	Regel in den Service, Controller bleibt dünn
Controller gibt <code>Task</code> (Entity) zurück	Service liefert <code>TaskResponse</code>
Service baut <code>ResponseEntity</code> oder kennt <code>HttpStatus</code>	Service wirft fachliche Exception, Advice übersetzt
<code>@Transactional</code> am Controller	Transaktionsgrenze = Service-Methode

Faustregel: Der Service muss sich **ohne HTTP** aufrufen und testen lassen.

- 1 Ziel & Aufbau
- 3 DTOs & Validierung
- 5 REST-Controller**
- 7 Konfiguration & Profile
- 2 Entity & Repository
- 4 Service & Transaktionen
- 6 Fehlerbehandlung
- 8 Testen (Ausblick)

Der API-Vertrag in Java

```
@RestController
@RequestMapping("/api")
public class TaskController {
    private final TaskService service;
    // Konstruktor-Injection wie beim Service

    @GetMapping("/teams/{teamId}/tasks")
    public List<TaskResponse> list(@PathVariable Long teamId) {
        return service.findByTeam(teamId);
    }

    @GetMapping("/tasks/{id}")
    public TaskResponse get(@PathVariable Long id) {
        return service.findById(id);
    }
}
```

- `@RestController` = `@Controller` + `@ResponseBody` : Der Rückgabewert wird per Jackson zu JSON, kein Template
- Ohne weitere Angabe antwortet Spring mit 200; der 404-Fall kommt als Exception aus dem Service

```
@PostMapping("/teams/{teamId}/tasks")
public ResponseEntity<TaskResponse> create(@PathVariable Long teamId,
    @Valid @RequestBody CreateTaskRequest req) {
    TaskResponse created = service.create(teamId, req);
    URI location = URI.create("/api/tasks/" + created.id());
    return ResponseEntity.created(location).body(created);    // 201
}

@PutMapping("/tasks/{id}")
public TaskResponse update(@PathVariable Long id,
    @Valid @RequestBody UpdateTaskRequest req) {
    return service.update(id, req);    // 200
}

@DeleteMapping("/tasks/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)    // 204
public void delete(@PathVariable Long id) {
    service.delete(id);
}
```

- `ResponseEntity` für Status **und** Header, `@ResponseStatus` für einen festen Status ohne Body

Gleicher Ursprung

HTML, CSS und JS unter `src/main/resources/static/` legen. Spring liefert sie mit aus, `fetch("/api/...")` funktioniert ohne Konfiguration.

Eigener Dev-Server

Front-End läuft z. B. auf Port 5173, API auf 8080: anderer Ursprung. Der Browser blockt die Antwort, bis die API **CORS** erlaubt.

Globale CORS-Freigabe für alle API-Pfade:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins("http://localhost:5173")
            .allowedMethods("GET", "POST", "PUT", "DELETE");
    }
}
```

- 1 Ziel & Aufbau
- 3 DTOs & Validierung
- 5 REST-Controller
- 7 Konfiguration & Profile
- 2 Entity & Repository
- 4 Service & Transaktionen
- 6 Fehlerbehandlung**
- 8 Testen (Ausblick)

Aus Exceptions werden Statuscodes

Situation	erkannt in	Exception	Status
Pflichtfeld fehlt, Text zu lang	@Valid	MethodArgumentNotValidException	400
JSON kaputt	Jackson	HttpMessageNotReadableException	400
Team oder Task existiert nicht	Service	NotFoundException (eigene)	404
Titel im Team schon vergeben	Service	ConflictException (eigene)	409
alles Unerwartete	irgendwo	jede andere	500

```
public class NotFoundException extends RuntimeException {  
    public NotFoundException(String resource, Object id) {  
        super(resource + " " + id + " nicht gefunden");  
    }  
}
```

```
@RestControllerAdvice
@Order(Ordered.HIGHEST_PRECEDENCE)
public class GlobalExceptionHandler {

    @ExceptionHandler(NotFoundException.class)
    ProblemDetail notFound(NotFoundException e) {
        return ProblemDetail.forStatusAndDetail(
            HttpStatus.NOT_FOUND, e.getMessage());
    }

    @ExceptionHandler(ConflictException.class)
    ProblemDetail conflict(ConflictException e) {
        return ProblemDetail.forStatusAndDetail(
            HttpStatus.CONFLICT, e.getMessage());
    }
}
```

HTTP/1.1 404 Not Found
Content-Type: application/problem+json

```
{
  "title": "Not Found",
  "status": 404,
  "detail": "Task 99 nicht gefunden",
  "instance": "/api/tasks/99"
}
```

- `ProblemDetail` ist das Standardformat für Fehler nach **RFC 9457**
- `@Order` : Unser Advice greift vor dem eingebauten von Spring Boot

```
@ExceptionHandler(  
    MethodArgumentNotValidException.class)  
ProblemDetail invalid(  
    MethodArgumentNotValidException e) {  
    var pd = ProblemDetail.forStatusAndDetail(  
        HttpStatus.BAD_REQUEST,  
        "Eingabe ungültig");  
    var errors = new HashMap<String, String>();  
    for (FieldError fe : e.getBindingResult()  
        .getFieldErrors()) {  
        errors.put(fe.getField(),  
            fe.getDefaultMessage());  
    }  
    pd.setProperty("errors", errors);  
    return pd;  
}
```

HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

```
{  
  "title": "Bad Request",  
  "status": 400,  
  "detail": "Eingabe ungültig",  
  "instance": "/api/teams/1/tasks",  
  "errors": {  
    "title": "darf nicht leer sein"  
  }  
}
```

- Das Front-End kann jede Meldung direkt neben das passende Formularfeld schreiben

1 Ziel & Aufbau

3 DTOs & Validierung

5 REST-Controller

7 Konfiguration & Profile

2 Entity & Repository

4 Service & Transaktionen

6 Fehlerbehandlung

8 Testen (Ausblick)

Derselbe Code, andere Umgebung

```
# src/main/resources/application.properties
spring.application.name=taskboard
spring.profiles.active=dev
```

```
spring.jpa.open-in-view=false
spring.mvc.problemdetails.enabled=true
```

```
# eigene Einstellung
taskboard.max-tasks-per-team=50
```

```
@ConfigurationProperties("taskboard")
public record TaskboardProperties(
    int maxTasksPerTeam) {}
```

- Gilt für **alle** Umgebungen; Profile ergänzen und überschreiben
- Umgebungsvariablen schlagen die Datei:
 `SPRING_DATASOURCE_URL` ersetzt
 `spring.datasource.url`
- Eigene Werte typsicher per
 `@ConfigurationProperties`, dazu
 `@ConfigurationPropertiesScan` an der
 Application-Klasse
- `open-in-view=false`: kein Nachladen im
 Controller, Fehler fallen sofort auf

```
# application-dev.properties
spring.datasource.url=jdbc:h2:mem:taskboard
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.h2.console.enabled=true
```

```
# application-prod.properties
spring.datasource.url=${DB_URL}
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASSWORD}
spring.jpa.hibernate.ddl-auto=validate
```

```
./gradlew bootRun # nutzt spring.profiles.active=dev
SPRING_PROFILES_ACTIVE=prod java -jar taskboard.jar # Umgebungsvariable gewinnt
```

Passwörter nie ins Repo. `${DB_PASSWORD}` wird beim Start aus der Umgebung gelesen.

- | | |
|---------------------------|----------------------------|
| 1 Ziel & Aufbau | 2 Entity & Repository |
| 3 DTOs & Validierung | 4 Service & Transaktionen |
| 5 REST-Controller | 6 Fehlerbehandlung |
| 7 Konfiguration & Profile | 8 Testen (Ausblick) |

Schichten, die man trennt, kann man einzeln testen

Schicht	Testart	Werkzeug	startet
Service	Unit-Test	JUnit 5 + Mockito	nichts von Spring, Millisekunden
Controller	@WebMvcTest	MockMvcTester + @MockitoBean	nur die Web-Schicht
Repository	@DataJpaTest	eingebettete H2	nur die JPA-Schicht
alles	@SpringBootTest	echter Kontext	die ganze Anwendung, langsam

- Viele schnelle Unit-Tests für die Regeln im Service, wenige breite Tests für das Zusammenspiel
- Eigene Repository-Methoden testen; geerbte wie `save` oder `findById` nicht

Vertiefung SWT-Vorlesung Testen

```
@WebMvcTest(TaskController.class)
class TaskControllerTest {
    @Autowired MockMvcTester mvc;
    @MockitoBean TaskService service;

    @Test
    void unbekannteTaskLiefert404() {
        given(service.findById(99L))
            .willThrow(new NotFoundException("Task", 99L));

        assertThat(mvc.get().uri("/api/tasks/99"))
            .hasStatus(HttpStatus.NOT_FOUND);
    }

    @Test
    void leererTitelLiefert400() {
        assertThat(mvc.post().uri("/api/teams/1/tasks")
            .contentType(MediaType.APPLICATION_JSON)
            .content("{\"title\": \"\"}"))
            .hasStatus(HttpStatus.BAD_REQUEST);
    }
}
```

- **Entity + Repository:** Datenmodell mit Beziehungen, Queries per Methodenname
- **DTOs als Records** an der API-Grenze, mit Bean Validation
- **Service:** Geschäftsregeln, `@Transactional`, fachliche Exceptions
- `@RestController` : HTTP ↔ DTO, passende Statuscodes
- `@RestControllerAdvice` : Exceptions → `ProblemDetail`
- **Profile:** dev mit H2, prod mit PostgreSQL, Secrets aus der Umgebung
- **Tests** pro Schicht

Für euer MVP: Baut diesen Durchstich einmal für eure wichtigste Ressource. Danach wiederholt sich das Muster für jede weitere.