

Spring Boot in Praxis

REST-API

Representational State Transfer

„**Representational State Transfer** (abgekürzt **REST**) ist ein Paradigma für die Softwarearchitektur von verteilten Systemen, insbesondere **Webservices**.“ - Wikipedia

Representational State Transfer

- Client-Server Architektur
 - Zustandslosigkeit (REST-Anfragen)
 - Caching
 - Einheitliche Schnittstelle (z.B. URLs)
 - Mehrschichtiges System
-
- Das REST-Paradigma definiert kein Protokoll
 - Wird i.d.R. mit dem HTTP-Protokoll umgesetzt

Spring Framework

- Das Spring Framework bietet ein zugängliches Programmier- und Konfigurationsmodell für die Entwicklung von Enterprise-Anwendungen mit Java.
 - Dependency Injection
 - Testing
 - Data Access
 - Web Frameworks
 - Spring MVC
 - ...

Spring Boot

- Spring Boot ist ein Werkzeug, um die Entwicklung von Web-Anwendungen und Microservices mit dem Spring Framework schneller und einfacher zu gestalten.
 - *Autoconfiguration*: Anwendungen werden mit vordefinierten *dependencies* initialisiert, welche nicht manuell konfiguriert werden müssen.
 - An *opinionated* approach to configuration: Spring Boot entscheidet wie sogenannte starter dependencies (Spring Starters) konfiguriert werden sollen.
 - *Standalone*-Anwendungen
- Spring Boot enthält über 50 Spring Starters

Maven


- Apache Maven ist eine Projektmanagement Software geschrieben in Java vorrangig für die Entwicklung mit Java.
 - Management von Abhängigkeiten (*3rd Party Libraries*)
 - Beschreibung von Abhängigkeiten in Konfigurationsdatei *pom.xml*
 - Erstellen von ausführbaren Dateien
 - ...
 - Maven ist eine Kommandozeilenanwendung.
 - Z.B: `mvn --version`
 - Ausgabe: Apache Maven 3.8.1 ...
- Maven Download: <https://maven.apache.org/download.cgi>
- Maven ist im Installationspaket von diversen Entwicklungsumgebungen (z.B. Apache NetBean, Eclipse) bereits enthalten

Spring Boot Projekt mit Maven

- Die pom.xml kann sowohl von Hand als auch mithilfe anderer Werkzeuge erzeugt und modifiziert werden.
- Der *Spring Initializer* bietet eine einfache Möglichkeit um ein neues Spring Boot Projekt, unter Verwendung von Maven, von Grund auf zu konfigurieren.

➤ Spring Initializer: <https://start.spring.io/>

Spring Boot Projekt mit Maven

 **spring** initializr

Project
☒ Maven Project ☐ Gradle Project

Language
☒ Java ☐ Kotlin ☐ Groovy

Spring Boot
☐ 3.0.0 (SNAPSHOT) ☐ 3.0.0 (M3) ☐ 2.7.1 (SNAPSHOT) ☒ 2.7.0
☐ 2.6.9 (SNAPSHOT) ☐ 2.6.8

Project Metadata

Group

Artifact

Name



Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 18 ☐ 17 ☒ 11 ☐ 8

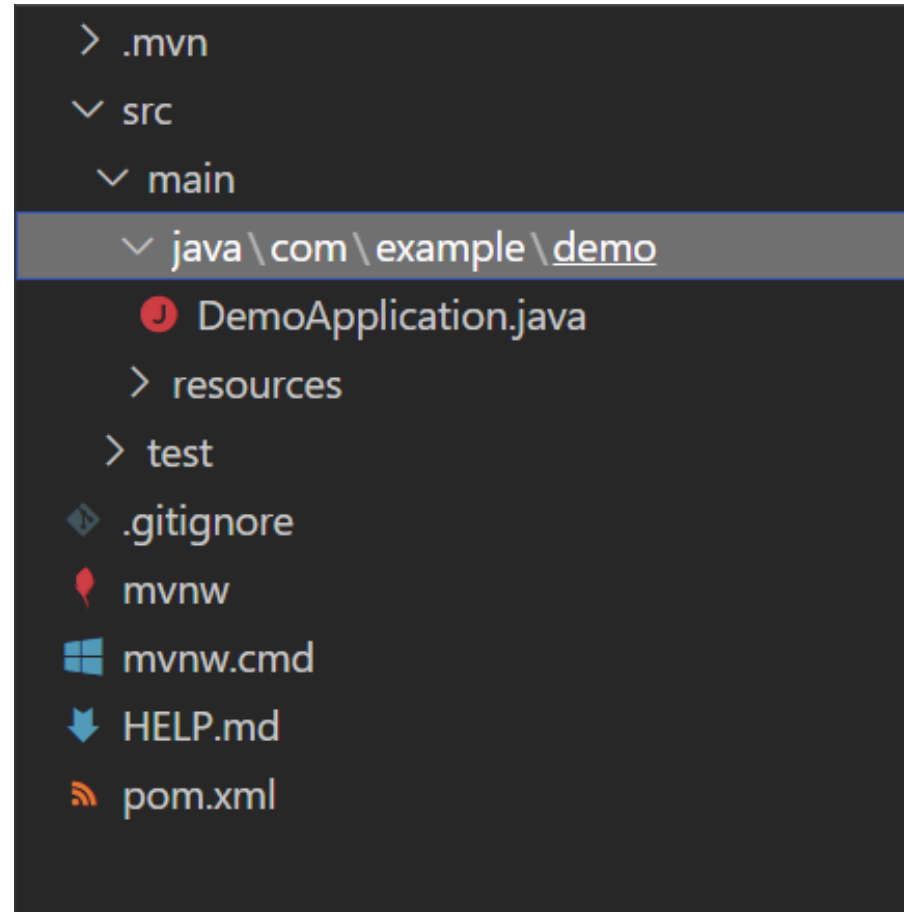
Dependencies [ADD DEPENDENCIES... CTRL + B](#)
Spring Web **WEB**
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.



[GENERATE CTRL + G](#) [EXPLORE CTRL + SPACE](#) [SHARE...](#)

Spring Boot Projekt mit Maven

- Beispielhafter Inhalt eines neu generierten Spring Boot Projekts mit der *Spring Web* dependency.



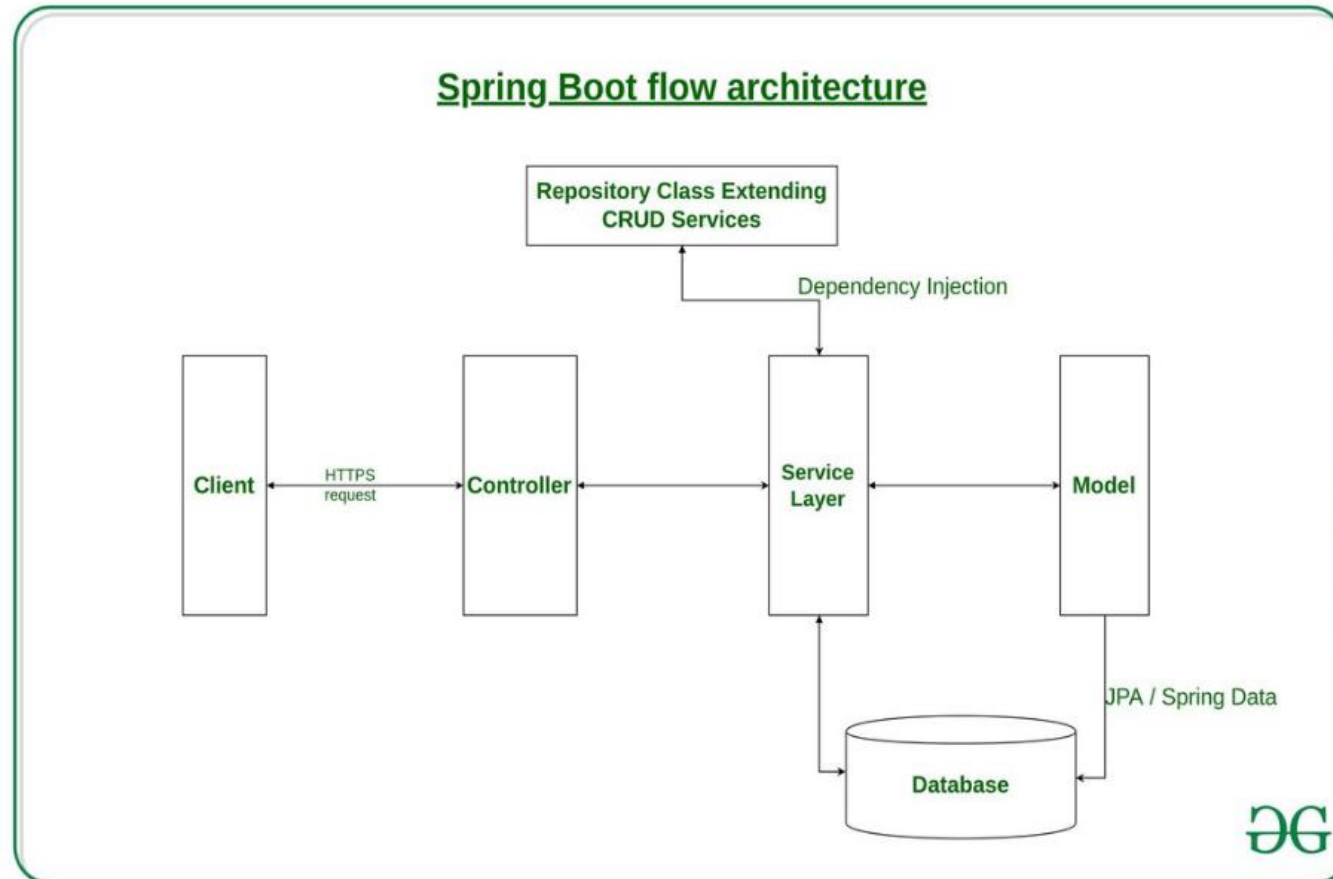
Spring Boot Projekt mit Maven

- Ausführen der Anwendung mit: `mvnw spring-boot:run`
 - Lädt und konfiguriert dependencies vor der Ausführung
- Erzeugen einer ausführbaren jar-Datei: `mvnw clean package`
 - Die ausführbare jar-Datei wird im *target*-Verzeichnis erzeugt
 - Name und Version können in der pom.xml festgelegt werden
- Ausführen der jar-Datei (z.B. demo.jar): `java -jar demo.jar`
 - BZW. `java -jar demo.jar --server.port=<port>`
 - Sofern kein Port festgelegt wird erwartet der Server Anfragen auf Port 8080
- Durchführen von Tests: `mvnw test`

Spring Boot Architektur

- Spring besitzt eine mehrschichtige Architektur
 1. *Presentation Layer*
 - Ansichten, HTML-Seiten, Templates, etc.
 2. *Business Layer*
 - Fachlogik der Applikation (*Service Layer*)
 3. *Persistence Layer*
 - Logik für das Speichern und Laden (sowie ggf. Konvertieren) von Daten in und aus Datenbanken
 4. *Database Layer*
 - Enthält alle Datenbanken

Spring Boot Architektur



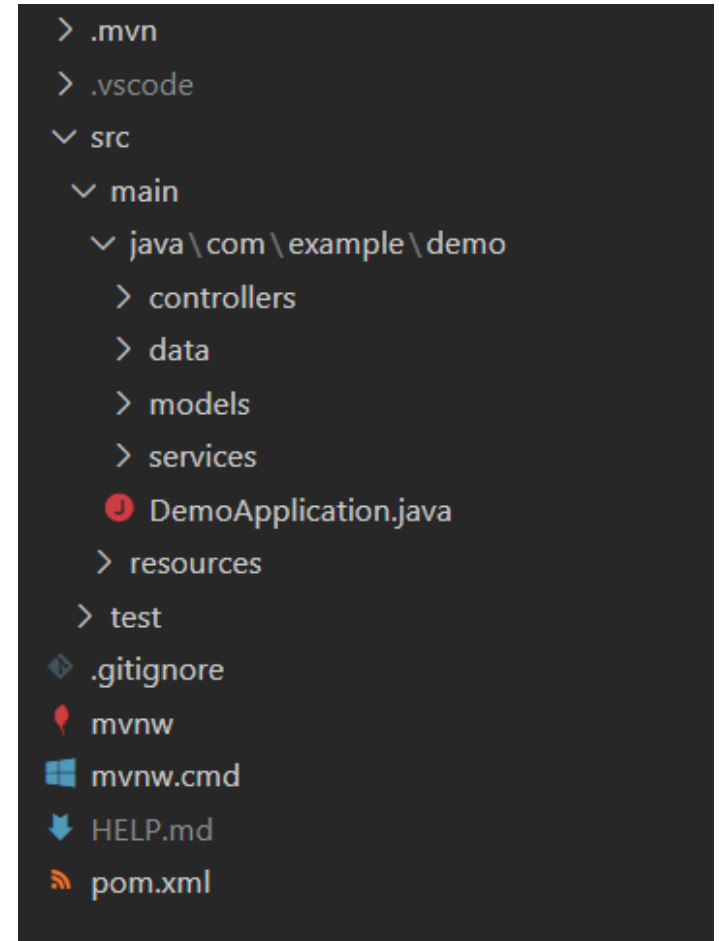
Quelle: Geeks for Geeks, <https://media.geeksforgeeks.org/wp-content/uploads/20220306170607/Fig48.jpg>

Spring Boot Architektur

- Ein Client macht eine HTTP-Anfrage an einen Server (GET, POST, ...).
- Diese Anfrage wird an einen Controller weitergeleitet.
- Der Controller bearbeitet die Anfrage und ruft die Fachlogik im Service Layer auf.
- Die Verwendung von persistenten Objekten muss nicht manuell verwaltet werden. Spring Boot ermöglicht den direkten Zugang zu Datenbankobjekten mittels Java Persistence API (JPA).
- Der Controller liefert eine *Antwort* (z.B. eine HTML-Seite oder ein JSON-Objekt) zurück an den Client.

Spring Boot Architektur

- Eine konkrete Umsetzung der Architektur erfordert eine klare Trennung zwischen: Controller, Datenmodellen, Services (Fachlogik) und Datenbankzugriffslogik.
 - Eine entsprechende *package*-Struktur ist sinnvoll.
- Weitere Fachlogik kann ggf. in andere Pakete ausgelagert werden, sollte jedoch immer über Services ausführbar sein.



Spring Boot Controller-Klassen

- Eine Controller-Klasse kann mittels der `@Controller` (oder `@RestController`) Annotation als solcher definiert werden.
- Mit Hilfe der `@RequestMapping(String)` Annotation kann festgelegt werden, über welchen Pfad ein Controller ausgeführt werden soll.
- Methoden eines Controllers können mit `@Get-`, `@Post-`, `@Put-`, `@Delete-`, `@Patch-` und ebenfalls mit `@RequestMapping(String)` annotiert werden.
- Der Rückgabewert eines Controllers kann bei Bedarf automatisch geparsed werden, jedoch ist es empfehlenswert ein `ResponseEntity<T>` zurückzuliefern.
 - Dieses ermöglicht neben der Definition eines response-bodies auch eine feingranulare Konfiguration der HTTP-Antwort (HTTP-Header, HTTP-Status, etc.)

Spring Boot Controller-Klassen

- Ein einfacher Controller für die Hauptseite:

```
@Controller
@RequestMapping("")
public class HomeController {
    @GetMapping("")
    public ResponseEntity<String> getHome() {
        return ResponseEntity.ok().body("Hello Spring");
    }
}
```

- Die ResponseEntity-Klasse bietet statische Methoden für die einfache Konstruktion eines ResponseEntity-Objekts

Spring Boot Controller-Methoden

- Controller-Methoden können Parameter aus einer URL entgegennehmen.
 - Zum einen *Query-Parameter*, z.B. `foo.de?param1=value1¶m2=value2`
 - Sowie *Pfad-Variablen*, z.B: `foo.de/42` (beliebige Zahl)
- Beispiel:

```
@GetMapping("/{id}")
public ResponseEntity<String> getHomeName(
    @PathVariable Integer id,
    @RequestParam(name = "name", defaultValue = "Anonymous") String name)
{
    return ResponseEntity.ok().body("Hello " + name + id);
}
```

Spring Boot Controller-Ausnahmen

- Im vorherigen Beispiel würde eine Ausnahme ausgelöst werden, sofern in der URL für die *id* keine Zahl sondern ein String übergeben wird (`NumberFormatException`).
- Dies würde ohne weitere Modifikationen dazu führen, dass eine Standardfehlerantwort an den Client zurückgeliefert wird.

Spring Boot Controller-Ausnahmen

- Ausnahmen können auf verschiedene Weise in Spring behandelt werden. Eine empfohlene Herangehensweise ist die Definition von Ausnahme-Handlern in einem Controller.
- Dies erfolgt indem Methoden mit `@ExceptionHandler` annotiert werden.
- Ausnahme-Handler können für bestimmte Ausnahmen definiert werden und werden automatisch aufgerufen sofern eine dieser Ausnahmen im Controller auftritt.

Spring Boot Controller-Ausnahmen

- Ein einfacher Ausnahme-Handler für `NumberFormatException`:

```
@ExceptionHandler({NumberFormatException.class})  
public ResponseEntity<String> badRequest(NumberFormatException e) {  
    return ResponseEntity.badRequest().body(e.getMessage());  
}
```

- Sofern mehr als eine Ausnahme behandelt werden soll, kann eine Liste von Exception-Klassen angegeben werden: `@ExceptionHandler({Exc1.class, Exc2.class})`

Spring Boot Services

- Das Service-Layer dient der Umsetzung des *Service-Locator Patterns*.
 - Ein Service ist eine beliebige Klasse von der in einem festgelegtem Kontext (häufig das gesamte Programm) genau 1 Objekt existiert.
 - Services können mittels eines *Service-Containers* angefordert werden.
 - Spring ermöglicht das automatische einbinden von Services mit der `@autowire` Annotation. Dies erspart das direkte Anfordern von Services und ermöglicht eine indirekte Einbindung.
- Entsprechend der Architektur werden Services für gewöhnlich von Controllern eingebunden. Ein Einbinden von Services an anderen Stellen (z.B. in anderen Services) ist jedoch möglich.

Spring Boot Services

- Ein einfacher *CounterService*, welcher die Anzahl der Besuche von verschiedenen Kombinationen aus Name + Id verwaltet.

```
@Service
public class CounterService {
    private HashSet<String> visitors = new HashSet<>();

    public int getCount() {
        return visitors.size();
    }

    public void visit(String name, int id) {
        visitors.add(name + id);
    }
}
```

Spring Boot Services

- Einbinden eines Services mit `@Autowired` als Attribut einer Klasse:

```
@Controller
@RequestMapping("")
public class HomeController {
    @Autowired
    private CounterService counter;

    @GetMapping("")
    public ResponseEntity<String> getHome(
        @RequestParam(name = "name", defaultValue = "Anonymous") String name
    ) {
        counter.visit(name, 0);
        return ResponseEntity.ok().body("Hello " + name);
    }
}
```

- Das CounterService Objekt (counter) wird von Spring erzeugt und bereitgestellt.

Spring Boot Services

- Wird ein Service an mehreren Stellen eingebunden, so wird stets dasselbe (einmalig erzeugte) Objekt zurück geliefert.

```
@Controller
@RequestMapping("/counter")
public class CounterController {
    @Autowired
    private CounterService counter;

    @GetMapping("")
    public ResponseEntity<String> getCounter() {
        return ResponseEntity.ok().body(String.format(
            "<h1>Counter</h1>" +
            "<p>There were a total of %d different visitors.</p>" +
            "<p><a href='/'>Return Home</a>.</p>", counter.getCount()));
    }
}
```


Spring Boot Services

- Beispiel: Simple HTML-*Template-Engine*

```
@Service
public class PagesService {
    private static final String rootPath = "/templates/";
    private HashMap<String, String> templates = new HashMap<>();

    public PagesService() {
        for (String path : getTemplatePaths()) {
            String name = path.substring(path.lastIndexOf('/') + 1, path.lastIndexOf('.'));
            templates.put(name, readTemplate(path));
        }
    }

    // ...
}
```

- HTML-Templates sind hier definiert als HTML-Seiten mit Formatspezifizierern (z.B. %s, %d, ...).
- Die Initialisierung des Services (und somit das Laden der Ressourcen) erfolgt ein mal beim Start des Programms.

Spring Boot Services

- Beispiel: Simple HTML-*Template-Engine*

```
@Service
public class PagesService {
    // ...

    public String getTemplate(String name) {
        return templates.get(name);
    }

    // ...
}
```

- Liefert das (geladene) Template mit dem übergebenem Namen zurück.

Spring Boot Services

- Beispiel: Simple HTML-*Template-Engine*

```
@Service
public class PagesService {

    // ...

    private List<String> getTemplatePaths() {
        List<String> paths = new ArrayList<>();

        try(Scanner sc = new Scanner(DemoApplication.class.getResourceAsStream(rootPath))) {
            while(sc.hasNext()) {
                paths.add(rootPath + sc.nextLine());
            }
        }

        return paths;
    }

    // ...
}
```

- Liefert eine Liste aller Pfade zu Ressourcen im rootPath („/template/“) zurück.

Spring Boot Services

- Beispiel: Simple HTML-*Template-Engine*

```
@Service
public class PagesService {

    // ...

    private String readTemplate(String path) {
        StringBuilder sb = new StringBuilder();

        try(Scanner sc = new Scanner(DemoApplication.class.getResourceAsStream(path))) {
            while(sc.hasNext()) {
                sb.append(sc.nextLine() + '\n');
            }
        }

        return sb.toString();
    }
}
```

- Lädt Template-Ressource und liefert den Inhalt dieser zurück.

Spring Boot Services

- Beispiel: HTML-*Template*

```
<!DOCTYPE html>
<head>
  <title>Demo Application</title>
</head>

<body>
  <h1>Home</h1>
  <p>Welcome %s#%d.</p>
</body>
```

- Formatspezifizierer können mittels der `String.format()`-Methode in einem geladenem Template durch Inhalte ersetzt werden.
- Hinweis: Die hier aufgeführte Template-Engine dient lediglich als Beispiel für Services.

Spring Boot Persistence – Dependencies

- Das Java Persistence API (JPA) bietet ein Framework für die Verwaltung von persistenten Objekten.
- H2 ist eine quelloffene und leichtgewichtige SQL-Datenbank für Java.
- Mit folgenden Einträgen in der `pom.xml` können H2 und JPA einem Maven-Projekt als Abhängigkeit hinzugefügt werden:

```
<dependencies>
  <!-- ... -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
  <!-- ... -->
</dependencies>
```

Spring Boot Persistence – Datenmodell

- JPA bietet unter anderem die `@Entity` Annotation um Bean-Klassen als Entität zu Kennzeichnen.
- Beispiel User-Modell:

```
@Entity
@Table(name = "Users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String username;
    private String email;

    // ...
}
```

- Zzgl. Öffentlicher Getter- und Setter-Methoden für alle Attribute, Standard-Konstruktor und Konstruktor mit Argumenten für alle Attribute (ohne id) sowie Überladungen für `hashCode()` und `equals()` (ohne id).
- Die `@Table` Annotation ist optional, jedoch empfehlenswert um Namenskonflikte in einer Datenbank zu vermeiden.

Spring Boot Persistence – Repository

- Für den Zugriff auf persistente Objekte kann mittels der `@Repository` Annotation ein Repository-Interface definiert werden.
- Zzgl. kann von `CrudRepository<TEntity, TID>` geerbt werden.
- In diesem Fall wird eine Implementierung des Interfaces automatisch von Spring generiert.
 - Dies erfordert die Einhaltung gewisser Regeln bei der Definition von Methoden-Signaturen.
- Von Repository-Klassen wird jeweils, wie von Service-Klassen, ein Objekt beim Start der Anwendung erzeugt.
- Entsprechend können Repositories ebenfalls mit `@autowire` eingebunden werden.

Spring Boot Persistence – Repository

- Die Typargumente von `CrudRepository<TEntity, TID>` legen jeweils den Typ der in der Repository zu speichernden Entität sowie den Typ ihrer ID fest.
- Das `CrudRepository`-Interface deklariert folgende Methoden:
 - `count`
 - `existsById`
 - `delete`, `deleteAll`, `deleteById`, `deleteAllById`
 - `findAll`, `findById`, `findAllById`
 - `save`, `saveAll`

Spring Boot Persistence – Repository

- Die Signatur der Methoden von `CrudRepository` folgen einem gewissen Schema:
 - `<operation>`
 - `<operation>All`
 - `<operation>By<Attribut>`
 - `<operation>AllBy<Attribut>`
- Es ist möglich weitere Methoden für die `find`-Operation, nach dem drittem Schema, für andere Attribute einer Entität zu deklarieren.

Spring Boot Persistence – Repository

- Es kann nach mehrere Kriterien gefiltert werden:
 - `findBy<Attribut1>And<Attribut2>...`
 - `findBy<Attribut1>Or<Attribut2>...`
- Die Rückgabe der find-Operation kann geordnet werden:
 - `findBy<Attribut1>OrderBy<Attribut2>Asc` (aufsteigend)
 - `findBy<Attribut1>OrderBy<Attribut2>Desc` (absteigend)
- Und mehr, weitere Infos hier (<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>).

Spring Boot Persistence – Repository

- Beispiel: User-Repository

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {
    Iterable<User> findByEmailOrderByUsername(String email, String username);
    Iterable<User> findByUsernameAndEmail(String username, String email);
    Iterable<User> findByUsername(String username);
}
```

Spring Boot Persistence – Service

- Jeglicher Zugriff auf eine Repository sollte über ein zugehöriges Service gelöst werden.
- Dies stellt sicher, dass lediglich die im Service angebotenen Operationen auf persistente Objekte möglich sind.

Spring Boot Persistence – Service

- Beispiel: User-Service

```
@Service
public class UserService {
    @Autowired
    private UserRepository repo;

    public User add(User user) {
        return repo.save(user);
    }

    public User get(Long id) {
        return repo.findById(id).get();
    }

    public void delete(User user) {
        repo.delete(user);
    }
}
```

Spring Boot Persistence – Controller

- Über eine Controller-Klasse kann eine HTTP-Schnittstelle definiert werden welche den Zugang zu Operationen eines Services ermöglicht.

```
@Controller
@RequestMapping("/users")
public class UserController {
    @Autowired
    private UserService users;

    @GetMapping("/{id}")
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        return ResponseEntity.ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(users.get(id));
    }

    // ...
}
```

Spring Boot Persistence – Controller

- Über eine Controller-Klasse kann eine HTTP-Schnittstelle definiert werden welche den Zugang zu Operationen eines Services ermöglicht.

```
@Controller
@RequestMapping("/users")
public class UserController {
    // ...

    @PostMapping("")
    public ResponseEntity<User> postUser(@RequestBody @Validated User user) {
        user = users.add(user);

        return ResponseEntity
            .status(HttpStatus.CREATED)
            .contentType(MediaType.APPLICATION_JSON)
            .body(user);
    }
}
```

- Parameter die aus einem (POST-)Request-Body entgegengenommen werden sollen, können mit `@RequestBody` annotiert werden. Mit `@Validated` wird sichergestellt, dass lediglich gültige JSON-Objekte akzeptiert werden.

Spring Boot Persistence – Konfiguration

- Die Konfiguration für JPA und H2 erfolgt in resources/application.properties. Folgend ein Beispiel:

```
spring.datasource.url=jdbc:h2:file:./mydata
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=admin
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Controls automatic schema creation (create, update, create-drop, validate, none)
spring.jpa.hibernate.ddl-auto=update

# This will ensure, that after Hibernate schema creation is performed
# then additionally schema.sql is read for any additional schema changes
# and data.sql is executed to populate the database.
spring.jpa.defer-datasource-initialization=true

# Also, script-based initialization is performed by default only for
# embedded databases, to always initialize a database using scripts,
# we'll have to use (e.g. main/resources/schema.sql):
# spring.sql.init.mode=always

# Enables h2 gui at http://localhost:8080/h2-console
spring.h2.console.enabled=true
```

Spring Boot Beispiel POST-Request (Script)

- Skript um einen Beispiel Nutzer mittels *curl* per POST-Request an einen Server zu senden:

```
#!/bin/bash

HOST_ADDR=${1:-"localhost:8080"}

TEST_USER='{
    "username": "TestUser",
    "email": "test@mail.com"
}'

curl -si "http://$HOST_ADDR/users" \
    -H "accept: application/json" \
    -H "Content-Type: application/json" \
    -d "$TEST_USER"
```

Spring Boot Beispiel – POST-Request (Form)

- HTML-Seite mit einfachem Formular zum erstellen eines Users:

```
<!DOCTYPE html>
<head>
  <title>Demo Application</title>
</head>

<body>
  <h1>New User</h1>
  <form action="/users/new" method="post">
    <label for="uname">Username:</label><br>
    <input type="text" id="uname" name="username"><br>
    <label for="email">Email:</label><br>
    <input type="text" id="email" name="email"><br><br>
    <input type="submit" value="Submit">
  </form>
</body>
```

Spring Boot Beispiel – POST-Request (Form)

- Anpassungen im UserController zum Anzeigen der Formular-Seite (mithilfe des PagesService) sowie zum bearbeiten des POST-Request vom Formular:

```
@Autowired
private PagesService pages;

@GetMapping
public ResponseEntity<String> getHome() {
    return ResponseEntity.ok()
        .body(pages.getTemplate("form"));
}
```

Spring Boot Beispiel – POST-Request (Form)

- Anpassungen im UserController zum Anzeigen der Formular-Seite (mithilfe des PagesService) sowie zum bearbeiten des POST-Request vom Formular:

```
@PostMapping("/new")
public ResponseEntity<User> postUserForm(
    @RequestParam(name = "username") String username,
    @RequestParam(name = "email") String email)
{
    var user = users.add(new User(username, email));

    return ResponseEntity
        .status(HttpStatus.CREATED)
        .contentType(MediaType.APPLICATION_JSON)
        .body(user);
}
```

- Formularparameter werden in Controllern als Query-Parameter entgegengenommen.