REST-API

Worum geht es?

- Anwendung müssen häufig mit anderer Anwendungen interagieren
  - Verschiedene Technologien
  - Skalierbar
- Hier kommt häufig das HTTP basierte Programmierparadigma REST zum Einsatz

HTTP

Request

```
GET /events HTTP/1.1
Accept: */*
```

Response

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json;charset=UTF-8
Date: Mon, 24 Aug 2015 19:15:40 GMT

[{"name":"Java User Group Treffen","id":1}]
```

HTTP Anfragemethoden

- GET
  - Fragt eine Ressource vom Server ab
  - Modifiziert keine Daten, die Ressource bleibt unverändert
- POST
  - Schickt Daten zur Verarbeitung an den Server
  - Kann genutzt werden um neue Ressourcen auf dem Server anzulegen oder bestehende zu modifizieren
- PUT
  - Schickt Daten zur Verarbeitung an den Server
  - Kann genutzt werden um neue Ressourcen auf dem Server anzulegen oder bestehende zu modifizieren
- DELETE
  - Löscht die angegebene Ressource auf dem Server

HTTP Statuscodes

- 200 OK
  - Der Request war erfolgreich
- 201 Created
  - Der Request war erfolgreich und die Ressource konnte angelegt werden
- 400 Bad Request
  - Der Request war fehlerhaft und konnte deshalb vom Server nicht verarbeitet werden
- 403 Forbidden
  - Der Zugriff auf die Ressource ist nicht gestattet
- 404 Not Found
  - Die angefragte Ressource wurde nicht gefunden

Was ist REST?

- Verwendung von URL und Pfaden zum Zugriff auf Ressourcen
  - `http://example.com/events`
  - `http://example.com/events/1`
- Verwendung der HTTP Methoden POST, GET, PUT, DELETE um Aktionen abzubilden
- Optional: Verwendung von HTTP Statuscodes um dem Client eine Rückmeldung zu geben
- Als Datenformat wird meist JSON verwendet (application/json)

JSON

- JSON ist das gängige Datenformat für REST Services
- Es unterstützt die Datentypen: Nullwert, Boolean, Zahl, Zeichenkette, Array, Objekt
- Datentypen können beliebig tief verschachtelt werden
- Spring Boot verwendet die Bibliothek Jackson zur JSON De-/Serialisierung
- Jackson kann Java Objekt ohne aufwändige Konfiguration von/nach JSON konvertieren

```
{
  "firstName": "Sandy",
  "lastName": "Smith",
  "isCustomer": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ]
}
```

Spring Web MVC

- Webframework von Spring
- Ursprünglich für "klassische" Webanwendungen entwickelt
  - Unterstützung für verschiedene Templatesprachen und Webframeworks
- Verwendet Model-View-Controller Pattern um die Webanwendung zu strukturieren
  - Separierung von Logik und Darstellung
  - Controller empfängt und verarbeitet den Request
- Der Controller muss den HTTP Request nicht selbst parsen, es ist einfach auf Request Parameter oder übermittelte Daten zuzugreifen

REST mit Spring Web MVC

- Verwendung von `@RestController`
- Mappen von HTTP Requests auf Java Methoden
  - Eine Methode pro Pfad, HTTP Methode, Accept Header
  - Einfacher Zugriff auf URL Pfade, Request Parameter, HTTP Header, Benutzerdaten
  - Konvertierung von Request Body in Java Objekt
- Umgehung einer HTML View, stattdessen wird das Objekt (oder ResponseEntity) direkt nach JSON (oder XML) gerendert

Erstellen eines Controllers

```
@RestController
public class GreetingController {

  private static final String template = "Hello, %s!";
  private final AtomicLong counter = new AtomicLong();

  @RequestMapping("/greeting")
  public Greeting greeting(@RequestParam(value="name", defaultValue="World") String name) {
      return new Greeting(counter.incrementAndGet(), String.format(template, name));
  }
}
```

`@RequestMapping`

- Beschreibt welche Controller Methode für einen bestimmten HTTP Request ausgeführt wird
- Optionale Definition auf Klassenebene für allgemeine Konfiguration, z.B. Pfad
- Parameter
  - `value`: Gemappte URL Pfad(e), Platzhalter umd Teilpfade auf Methodenparameter zu mappen
  - `method`: Die HTTP Method(en), z.B. `POST` oder `GET`
  - `produces`: Beschreibt welcher Medientypen erzeugt werden, der `Accept` Request Header muss passen
  - `consumes`: Einschränkung, welche Medientypen verarbeitet werden, der `Content-Type` Request Header muss passen
  - Außerdem `param` und `header`

Methoden Parameter

- Werte im HTTP Request können auf Methoden Parameter gebunden werden.
  - `@PathVariable`: Teil des URL Pfades extrahieren

```
@RequestMapping(value = "/pets/{petId}")
public Pet getPet(@PathVariable String petId) {...}
```

  - `@RequestParam`: Request Parmeter extrahieren, Angabe von Default Werten möglich
  - `@RequestHeader`: HTTP Header extrahieren
  - `@RequestBody`: Den HTTP Request Body extrahieren, Konvertierung von JSON auf ein Java Objekt wird unterstützt

```
@RequestMapping(value = "/pets", method = RequestMethod.POST))
public Pet createPet(@RequestBody Pet pet) {...}
```

Testen von REST Schnittstellen

- Die Maven dependency `spring-boot-starter-test` beinhaltet nützliche Bibliotheken zum Testen
  - JUnit: Standard (Unit) Test Framework
  - Hamcrest: Matcher zum Vergleich von erwarteten und tatsächlichen Werten
  - Spring Test: Framework für Integrationstests
- Prinzipiell besteht mit `MockMVC` von Spring MVC die Möglichkeit Unittests für REST Controller zu schreiben, allerdings verhalten sich die gemockten Schnittstellen nicht immer 100% gleich.

Integrationstests

- Integrationstests starten die Anwendung und kommunizieren via HTTP mit der Anwendung
- Es wird also der komplette Stack getestet
- Der Test wird mit Annotationen konfiguriert, um die Anwendung zu konfigurieren und zu starten:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = Application.class)
@WebIntegrationTest
public class SomeIntegrationTest {

    @Test
    public void test_getAllEvents(){
           ...
    }

}
```

Integrationstests mit REST Assured

- Nun können beliebige HTTP Clients verwendet werden um Tests zu schreiben
- Mit REST Assured https://github.com/jayway/rest-assured können Tests mit einer DSL einfach im BDD Stil geschrieben werden

```
given(...)
.when()
    .get("/orders/523")
.then()
    .statusCode(200)
    .contentType("application/json;charset=UTF-8")
    .body("status", equalTo("shipped"))
;
```

- Tipp: Mit `when().log().all()` und `then().log().all()` werden Request und Response komplett auf der Konsole ausgegeben.

Übung 4

- Ersetze die Annotation `@Controller` in `EventController` durch `@RestController` und entferne die Annotationen `@ResponseBody`
- Erweitere die Klasse `Event` um die Attribute id, startDate und endDate
- Erweitere den `EventController` um die folgenden Endpunkte
  - Einen Endpunkt um ein einzelnes Event über seinen Namen abzufragen
  - Einen Endpunkt um ein Event zu speichern (method POST)
  - Einen Endpunkt um eine Ressource zu löschen
  - Ergänze dafür entsprechende Methoden im EventService. Verwende eine Map um die Eventobjekte zwischenzuspeichern.
  - Um Fehlerhandling und das Mapping auf HTTP Statuscodes kümmern wir uns später
- Erstelle für die Endpunkte Integrationstests mit dem "REST Assured" Framework. Teste dabei den Status Code, den Content-Type und den Inhalt des Bodys der Antwort auf eine HTTP Anfrage. Füge dazu in der pom.xml die auf der folgenden Seite aufgeführten Dependencies hinzu.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>2.5.0</version>
    <scope>test</scope>
</dependency>
```

Konfiguration

Worum geht es

- Konfigurationseinstellungen werden benötigt um die verwendeten Spring Features und umgebungsspezifische Parameter festzulegen
  - Festlegung zur Laufzeit
  - Festlegung zur Compile-Zeit
- Spring Boot
  - unterstützt eine XML-basierte (legacy) Konfiguration und eine Java-Annotation-basierte Konfiguration
  - unterstützt Property-Files im Java Property Format oder in YAML für Parameter

@EnableAutoConfiguration

```
@EnableAutoConfiguration
public class SampleController
```

- Spring Boot ermittelt aus den vorhandenen JARs (Logging Framework, Webcontainer, Datenbanktreiber) die "gewünschte" Konfiguration
  - spring-boot-starter-web: Spring MVC, Tomcat, Logback, Jackson, YAML-basierte Konfiguration, etc.
- Guter Start, kann später überschrieben werden
- 3 Konfigurationsmöglichkeiten
  - Property Files
  - Configuration Beans (@Configuration)
  - Annotationen, die spezifische Features aktivieren/deaktivieren

Property Files

- Für Einstellungen die zur Laufzeit geändert werden sollen
  - Umgebungsabhängige Einstellungen z.B. Datenbankverbindung in Entwicklung, Test, Produktion
- Ursprünglich wurde das Properties Format genutzt
  - `src/main/resources/application.properties`
- Spring Boot unterstützt auch YAML Format
  - `src/main/resources/application.yml`

Beispiel YAML Konfiguration:

```
email:
  customerservice:
    address: customerservice@mycompany.com
    password: supersecretpassword
```

Einbinden in den Code:

```
@Value("${email.customerservice.address}")
String customerServiceEmail;
```

Vordefinierte Spring Boot Properties

- Spring Boot bietet eine vordefinierte Liste an Properties für die enthaltenen Komponenten
  - Logging, Application Server, Start Banner, JSON Serialisierung etc.
- Eine komplette Liste findet sich unter
  - http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties

Übung

- Ergänze eine YAML-basierte Properties Datei und setze das vordefinierte `spring.main.show_banner` auf den Wert `false`.
- Ergänze dort das Property `email.admin`, injecte das Property in den EventController und gib in der Methode `home` den Wert des Properties zurück.

Datenzugriff

Um was geht es?

- Zugriff auf Daten in externen Datenspeichern, beispielsweise Suchserver, SQL- oder NoSQL Datenbanken
- Bei relationalen Datenbanken müssen die Zeilen der Datenbank auf Objekte in Java gemapped werden (Objekt-relationales Mapping)
- Dafür gibt es die Java Persistence API
- Spring Data bietet dafür eine Vereinfachung

Objekt-relationales Mapping

- Technik um Objekte einer objektorientieren Sprache in einer relationalen Datenbank zu speichern
- Prinzipiell wird pro Klasse eine Tabelle verwendet, die Attribute werden in Spalten abgelegt
- Oft werden "künstliche" IDs zur Klasse hinzugefügt
  - Als Primärschlüssel der Datenbanktabelle
  - Zur Abbildung von Referenzen
- Es gibt verschiedene Strategien um Vererbung abzubilden

Java Persistence API (JPA)

- Standardisierte API für Objekt-relationales Mapping
- Definiert Mechanismen wie Java Objekte auf Datenbanktabellen abgebildet werden können (Entities)
- Definiert eine API für Datenbankabfragen (Java Persistence Query Language)
- Beliebte Implementierung ist Hibernate

Entities

- @Entity zeigt an, dass Instanzen einer Klasse in einer Datenbank persistiert werden soll
- POJOs (Plain Old Java Objects)
- Instanzen einer Entity Klasse repräsentieren eine Zeile in einer Datenbanktabelle
- Alle Angaben, die benötigt werden um ein Java Objekt auf eine Tabellenspalte abbilden zu können, werden über Annotationen bereitgestellt
  - @Id kennzeichnet den Primärschlüssel, @GeneratedValue einen automatisch generierten Primärschlüssel
  - Jede nicht-statische, nicht-finale Klassenvariable wird automatisch persistiert
    - Ausnahmen können mit `@Transient` gekennzeichnet werden
  - @Column kann genutzt werden, um weitere Details zur Speicherung festzulegen
    - Beispielsweise Spaltennamen, Null-Werte

Beispiel

```
@Entity
public class Order {
    public enum OrderState  { CREATED, UPLOADED, DELETED }

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    private String caseId;

    @Enumerated(EnumType.STRING)
    private OrderState state = OrderState.CREATED;


    // getter and setter...
}
```

```
CREATE TABLE "order"
(
    id BIGINT PRIMARY KEY NOT NULL,
    case_id VARCHAR(255),
    state VARCHAR(255)
);
```

```
1,    123,    UPLOADED
2,    124,    CREATED
3,    125,    UPLOADED
```

Was ist Spring Data?

- Abstraktion für verschiedene Datenspeicher, relational und nicht-relational
- Hauptkonzept ist ein allgemeines Interface `Repository` bzw. `CrudRepository` für CRUD Operationen
  - Diese stellen häufig benötigte Standardfunktionen bereit
  - Man programmiert nur gegen dieses Interface
  - Die eigentliche Implementierung (z.B. JPA+Hibernate) bleibt verborgen
  - Prinzipiell kann der Datenspeicher gewechselt werden, ohne die Anwendung zu verändern
    - Solange man keine speziellen Features benötigt ;-)

CrudRepository

```
public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {

    <S extends T> S save(S entity);

    T findOne(ID primaryKey);

    Iterable<T> findAll();

    Long count();

    void delete(T entity);

    boolean exists(ID primaryKey);

    // …
}
```

Eigene Repositories

Erstellen des Repositories

```
public interface OrderRepository extends CrudRepository<Order, Long>{

}
```

Injecten des Repositories

```
@Autowired
OrderRepository orderRepo;
```

Verwenden des Repositories

```
Iterable<Order> allOrders = orderRepo.findAll();
```

Hinzufügen von Abfragen

- Spring Data kann aus Methodennamen automatisiert SQL Queries erzeugen
- Diese müssen einem Namenschema folgen
- Beispiele:

```
List<Order> findByState(Order.OrderState state);
```

```
OrderFile findFirstByIdAndOrderId(Long id, Long OrderId);
```

- Übersicht der Schlüsselwörter
  - http://docs.spring.io/spring-data/commons/docs/current/reference/html/

Konfiguration

- Es müssen Dependencies zu `spring-boot-starter-data-jpa` und einem Datenbanktreiber eingebunden werden
- Für In-Memory Datenbanken ist meistens keine weitere Konfiguration notwendig
- Für "echte" Datenbanken werden Verbindungparameter via Properties festgelegt:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Testen

- Es ist sinnvoll, die eigenen Repository Methoden und das OR-Mapping der Entities zu testen
- Hierfür ist ein Intergrationtest notwendig, die Repository Instanz wird in den Test injected
- Es ist wichtig auf ein konsistentes Testdatenset zu achten:
  - Verändert ein Test die Datenbank, sind die Änderungen für andere Tests sichtbar
  - Die Testreihenfolge ist nicht deterministisch
- Um einen Integrationstest zu schreiben, können die folgenden Annotationen verwendet werden

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = EventApp.class)
```

Übung 5

- Ergänze die auf der folgenden Seite aufgeführten Dependencies in der pom.xml
- Erstelle eine Klasse `org.informatica.persistence.EventRepository` und baue sie in die Klasse `org.informatica.service.EventService` ein, so dass Events immer direkt aus der Datenbank gelesen und in diese geschrieben werden.
- Implementiere im EventRepository eine Methode um Events nach Namen zu suchen
- Implementiere Tests für diese Methode

pom.xml

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

Inhalt der h2 Datenbank anschauen

Klasse EventApp ergänzen

```
...
@Configuration
public class EventApp {

    ...

    @Bean
    public ServletRegistrationBean h2WebConsole() {
        return new ServletRegistrationBean(new WebServlet(), "/h2-console/*");
    }
}
```

application.properties ergänzen

Setze in den application.properties den Wert `spring.h2.console.enabled` auf `true`

Anschliessend kann der Inhalt der h2 Datenbank unter http://localhost:8080/h2-console im Browser betrachtet werden. Als JDBC Url muss dabei der folgende Wert angegeben werden:

```
jdbc:h2:mem:testdb
```