

# Spring FHIR

---

## Aufgabe

Implementiere die Entitäten [Patient](#) und [Practitioner](#) auf eine Weise dass sie mit dem FHIR-Standard kompatibel sind. Diese FHIR-Ressourcen haben auch eine Vielzahl an Unterressourcen welche ebenfalls implementiert werden müssen.

Die Entitäten Patient und Practitioner müssen nicht komplett umgesetzt werden. Sollte ein Subtyp noch ein JSON-Objekt als Subtyp haben, kann dieses durch einen einfachen Typen (String, Number, Boolean, Null) ersetzt werden. Wird also z.B. auf eine [Organization](#) verwiesen - eine eher komplexe Ressource - kann diese z.B. durch einen String ersetzt werden in dem der Firmenname eingetragen ist.

### Patient

```
{
  "resourceType" : "Patient",
  // from Resource: id, meta, implicitRules, and language
  // from DomainResource: text, contained, extension, and modifierExtension
  "identifier" : [{ Identifier }], // An identifier for this patient
  "active" : <boolean>, // Whether this patient's record is in active use
  "name" : [{ HumanName }], // A name associated with the patient
  "telecom" : [{ ContactPoint }], // A contact detail for the individual
  "gender" : "<code>", // male | female | other | unknown
  "birthDate" : "<date>", // The date of birth for the individual
  // deceased[x]: Indicates if the individual is deceased or not. One of these 2:
  "deceasedBoolean" : <boolean>,
  // NICHT NOTWENDIG"deceasedDateTime" : "<dateTime>",
  "address" : [{ Address }], // An address for the individual
  // NICHT NOTWENDIG"maritalStatus" : { CodeableConcept }, // Marital (civil)
status of a patient
  // multipleBirth[x]: Whether patient is part of a multiple birth. One of these
2:
  "multipleBirthBoolean" : <boolean>,
  // NICHT NOTWENDIG"multipleBirthInteger" : <integer>,
  // NICHT NOTWENDIG"photo" : [{ Attachment }], // Image of the patient
  "contact" : [{ // A contact party (e.g. guardian, partner, friend) for the
patient
    "relationship" : [{ CodeableConcept }], // The kind of relationship
    "name" : { HumanName }, // A name associated with the contact person
    "telecom" : [{ ContactPoint }], // A contact detail for the person
    "address" : { Address }, // Address for the contact person
    // NICHT NOTWENDIG"gender" : "<code>", // male | female | other | unknown
    // NICHT NOTWENDIG"organization" : { Reference(Organization) }, // C?
Organization that is associated with the contact
    // NICHT NOTWENDIG"period" : { Period } // The period during which this
contact person or organization is valid to be contacted relating to this patient
  }],
  // NICHT NOTWENDIG"communication" : [{ // A language which may be used to
```

```

communicate with the patient about his or her health
  // NICHT NOTWENDIG "language" : { CodeableConcept }, // R! The language which
can be used to communicate with the patient about his or her health
  // NICHT NOTWENDIG "preferred" : <boolean> // Language preference indicator
}],
  // NICHT NOTWENDIG"generalPractitioner" : [{
Reference(Organization|Practitioner| PractitionerRole) }], // Patient's nominated
primary care provider
  // NICHT NOTWENDIG"managingOrganization" : { Reference(Organization) }, //
Organization that is the custodian of the patient record
  // NICHT NOTWENDIG"link" : [{ // Link to another patient resource that concerns
the same actual person
  // NICHT NOTWENDIG "other" : { Reference(Patient|RelatedPerson) }, // R! The
other patient or related person resource that the link refers to
  // NICHT NOTWENDIG "type" : "<code>" // R! replaced-by | replaces | refer |
seealso
  }]
}

```

## Practitioner

```

{
  "resourceType" : "Practitioner",
  // from Resource: id, meta, implicitRules, and language
  // from DomainResource: text, contained, extension, and modifierExtension
  "identifier" : [{ Identifier }], // An identifier for the person as this agent
  "active" : <boolean>, // Whether this practitioner's record is in active use
  "name" : [{ HumanName }], // The name(s) associated with the practitioner
  "telecom" : [{ ContactPoint }], // A contact detail for the practitioner (that
apply to all roles)
  "address" : [{ Address }], // Address(es) of the practitioner that are not role
specific (typically home address)
  "gender" : "<code>", // male | female | other | unknown
  "birthDate" : "<date>", // The date on which the practitioner was born
  // NICHT NOTWENDIG"photo" : [{ Attachment }], // Image of the person
  // NICHT NOTWENDIG"qualification" : [{ // Certification, licenses, or training
pertaining to the provision of care
  // NICHT NOTWENDIG"identifier" : [{ Identifier }], // An identifier for this
qualification for the practitioner
  // NICHT NOTWENDIG"code" : { CodeableConcept }, // R! Coded representation of
the qualification
  // NICHT NOTWENDIG"period" : { Period }, // Period during which the
qualification is valid
  // NICHT NOTWENDIG"issuer" : { Reference(Organization) } // Organization that
regulates and issues the qualification
  // NICHT NOTWENDIG}],
  // NICHT NOTWENDIG"communication" : [{ CodeableConcept }] // A language the
practitioner can use in patient communication
}

```

# Lombok

Tipp: Um nicht immer Getter, Setter etc. in einer Klasse implementieren zu müssen, kann man mit `lombok` (bereits in dem Projekt eingebunden) sich diese im Java-Bytecode automatisch generieren lassen.

## Beispiel-Klasse

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@ToString
class Something {
    int someProperty = 0;
}
```

## Tipps für die Implementierung

Da sowohl `Patient` als auch `Pracitioner` Personen sind wäre es möglich eine abstrakte Basisiklasse `Person` zu erstellen in der allgemeine Properties bereits enthalten sind.

Da beide Controller ein sehr ähnliches Verhalten haben kann man dieses unabhängig vom Typ mittels *generics* implementieren. Für je Patient als auch Pracitioner kann dann eine Ableitung davon anderen Mappings zugeordnet sein.

```
public class PersonController<T, ...>{

    public List<T> getAllEntities(){
        ...
    }

    ...
}

@RequestMapping("/patient")
public class PatientController {

    PersonController<Patient> baseController;

    @GetMapping("/")
    public List<Patient> getAllEntities() {
        return baseController.getAllEntities();
    }

    ...
}
```

# FHIR

FHIR steht für **Fast Healthcare Interoperability Resources**. Und ist ein von **HL7** veröffentlichter Standard um Daten im Gesundheitsbereich Programmübergreifend austauschen zu können. **HL7** ist eine Organisation welche für die ANSI in den USA Standards im Gesundheitswesen entwickelt. Durch diesen Standard können alle möglichen Gesundheitsprogrammen Daten untereinander austauschen.

## Tests

Implementiere deinen eigenen Test der die Daten auf FHIR-Kompatibilität prüft.

Du kannst prüfen, ob die von dir generierten JSON-Daten für einen Patienten passen, indem du sie mit dem Beispiel von [hier](#) vergleichst.

Erstelle dazu JSON-Daten anhand der Dokumentation und vergleiche mittels Soll- und Ist-Wert (Expected and Actual Value).

### Beispiel-Test

```
@Test
public void testCompareReturnedPatientJSONtoFHIRCompliantJSON() throws Exception {
    mockMvc
        .perform(get("/patient/1")) // get patient with id 1
        .andExpect(status().isOk()) // expect 200 HTTP status code
        .andExpect(content().json("{\"your_patient\": \"test_data\"}")); // returned
data should be of type json and
        // contain the same parameters as the test data
}
```

Die initialen Testdaten sollen in der Ressourcendatei **import.sql** direkt als SQL-Statements eingefügt werden. Beim Start deines Programmes werden alle SQL-Befehle in dieser Datei ausgeführt.

## Generierung von SQL-Daten

Um zu testen ob deine SQL-Statements funktionieren kannst du sie bevor du sie in **import.sql** einfügst auch manuell ausführen. Starte dazu den Webserver und gehe dann auf die Weboberfläche der H2-Datenbank. Diese erreichst du unter: <http://localhost:8080/h2-console>

Wenn dein Befehl ohne Fehler ausgeführt wird, kannst du ihn bedenkenlos einfügen. Beachte aber, dass im Ausgangszustand dieses Projekts noch keine Entitäten angelegt wurden. Es ist also noch kein **CREATE TABLE**-Statement ausgeführt worden.

## API-Designrichtlinien

Bitte beachte bei den HTTP-Requestmethoden den dazugehörigen [RFC 7231](#) [Sektion 4.3](#)

Kurz zusammengefasst, wann man was nimmt:

GET	Wenn man eine Ressource holt
-----	------------------------------

POST	Wenn man eine nicht identifizierte Ressource hochlädt oder ändert
PUT	Wenn man eine identifizierte Ressource hochlädt
PATCH	Wenn man eine identifizierte Ressource teilweise ändert. Hat einen extra RFC, siehe: <a href="#">RFC 5789</a>
DELETE	Wenn man eine Ressource löscht

Auch gibt es einen schönen Blogartikel von Stackoverflow in dem gutes API-Design erklärt wird. [Blogartikel: Best practises for REST API Design](#)

## Native Queries

Um komplexere Abfragen jeglicher Art durchführen kann man direkt SQL einsetzen. In Spring gibt es die Möglichkeit mit der [@Query Annotation](#) Abfragen direkt zu definieren. Prüfe vor der Query-Verwendung in der H2-Console ob die Abfragen auch richtig sind.

Gib jeder Funktion in einem Repository diese Annotation mit der entsprechenden Abfrage. Eine Anleitung wie man diese aufbaut findest du [hier](#).