

Kursunterlagen

Modul 223 Multi-User-Applikationen objektorientiert realisieren

Dokumentinformationen

Dokumenttitel: Kursunterlagen M223
Thema: Modul 223: Multi-User-Applikationen objektorientiert realisieren
Dateiname: ku-modul223v0.6.docx
Speicherdatum: 22.06.2018
Autor: Joel Holzer

Die vorliegenden Kursunterlagen wurden in ihrer Originalfassung von der Noser Young Professionals AG (NYP) entwickelt. Der Einsatz und die Weiterentwicklung dieser Unterlagen in der Noser Young Professionals AG erfolgt mit der ausdrücklichen Genehmigung des NYP.

Inhaltsverzeichnis

1	Einleitung	4
1.1	Sinn und Zweck	4
1.2	Referenzdokumente und Internetquellen.....	4
2	Administratives.....	5
2.1	Verwendete Software.....	5
2.2	Wichtige Webseiten	5
2.3	Code-Beispiele / Demo-Projekte	6
3	Software-Architekturmuster im Überblick	7
3.1	Gruppenarbeit.....	7
3.2	Ergebnisse der Gruppenarbeit	7
4	Analyse & Design von Applikationen.....	8
4.1	Einleitung	8
4.2	Domänenmodell.....	9
4.3	Use Cases	12
4.4	User Interface Design	20
4.5	Mockups erstellen	24
4.6	Entity Relationship Diagram (ERD)	26
4.7	UML Klassendiagramm.....	28
4.8	UML Sequenzdiagramm	32
5	RESTful-APIs - Basiswissen	36
5.1	Was ist eine RESTful-API?	36
5.2	Architekturprinzipien von REST	38
5.3	http Request Methoden – kurze Übersicht	38
5.4	JSON	39
6	Erstes Projekt – Hello RESTful-API	40
6.1	Projekt erstellen	40
6.2	Dependencies hinzufügen.....	41
6.3	Klassen erstellen.....	41
6.4	Projektstruktur.....	42
6.5	Applikation starten & testen.....	43
7	Datenbank-Anbindung mit JPA & MySQL.....	44
7.1	Was ist OR-Mapping?	44
7.2	MySQL RESTful-API mit grundlegenden DB-Operationen	45
7.3	MySQL RESTful-API mit Relationen	50
7.4	MySQL RESTful-API mit n:m Relationen	57
7.5	Eigene Repository-Abfragen	65
8	RESTful-API - Data Transfer Objects (DTO).....	68
8.1	Variante ohne DTO	68
8.2	Variante mit DTO	68
8.3	Vorteile von DTOs.....	69
8.4	Nachteile von DTOs	69
9	Authentifizierung & Autorisierung mit JWT.....	70
9.1	Authentifizierung vs. Autorisierung	70
9.2	Authentifizierung bei REST-API	70
9.3	JSON Web Token (JWT)	73

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

9.4	Authentifizierung mit JWT & Spring Boot.....	76
10	Sicherheit bei REST-APIs	78
10.1	HTTPS.....	78
10.2	Authentifizierung & Autorisierung	78
10.3	Input Validierung	78
10.4	Content-Type validieren	78
10.5	Endpoint-Aufrufe einschränken.....	78
11	Input Validierung mit Spring Boot.....	79
11.1	Warum eine Input Validierung?	79
11.2	Standard-Validierung von Spring Boot	79
11.3	Weitere Input-Validierungen mit Spring Boot.....	80
12	DB-Transaktionen	83
12.1	Einleitung	83
12.2	Was ist eine DB-Transaktion?	84
12.3	Eigenschaften von Transaktionen - ACID-Prinzip.....	84
12.4	Fehler bei Transaktionsausführung	85
12.5	DB-Transaktionen mit MySQL.....	87
12.6	DB-Transaktionen mit Spring Boot.....	88
13	REST-API Deployment auf Heroku.....	93
13.1	REST-API deployen.....	93
13.2	REST-API testen.....	95
14	Android-App – REST-API anbinden.....	97
14.1	Android-App – REST-Anbindung umsetzen	98
15	Abbildungsverzeichnis	102
16	Tabellenverzeichnis	102

Änderungsgeschichte

Version	Datum	Autor	Details
0.1	12.06.2018	holzer	Erste Version. Inhalt von Tag 1.
0.2	13.06.2018	holzer	Inhalt von Tag 2
0.3	14.06.2018.	holzer	Inhalt von Tag 3
0.4	20.06.2018	holzer	Inhalt von Tag 4
0.5	21.06.2018	holzer	Inhalt von Tag 5
0.6	22.06.2018	holzer	Inhalt von Tag 6

Tabelle 1: Änderungsgeschichte

1 Einleitung

1.1 Sinn und Zweck

Dieses Dokument unterstützt im Unterricht des ÜK-Moduls 223. Es kann einerseits vom Dozenten für den Unterricht in der Klasse beigezogen werden, andererseits dient es den Lernenden als Nachschlagewerk und Lernleitfaden.

Der klare Fokus des Dokuments liegt auf den Inhalten der Modulbeschreibung. Dieses Dokument beinhaltet nicht alle Bereiche der Entwicklung von objektorientierten Multi-User Applikationen, da dies den Zeitrahmen des ÜK-Moduls bei weitem übersteigen würde. Alle im ÜK in der Theorie behandelten Themen sind jedoch Teil dieses Dokuments.

Die Entwicklung von Multi-User-Applikationen im Modul 335 erfolgt im Rahmen einer Service-orientierten Architektur (SOA). Der Service wird als RESTful-API mit Spring Boot umgesetzt. Als Client wird eine Android-App angebunden.

Weiter erläutert dieses Dokument Elemente, welche beim Entwurf von Multi-User-Applikationen relevant sind. Dies sind beispielsweise verschiedene UML-Diagramme, das Transaktionsmanagement und das Testing.

Für die Durcharbeitung dieser Unterlage werden fundierte Kenntnisse in der Programmiersprache Java, gute Kenntnisse und korrekte Anwendung der Prinzipien der Objektorientierten Programmierung und Kenntnisse über die gängigen Design Patterns vorausgesetzt.

Die Abfolge der Kapitel in diesem Dokument entspricht nicht zwingend dem Ablauf der Unterrichtseinheiten.

1.2 Referenzdokumente und Internetquellen

- [1] Modulidentifikation zum Modul 223, Modulbaukasten Release 3 (BIVO R6)
<https://cf.ict-berufsbildung.ch/modules.php?name=Mbk&a=20101&cmodnr=223>

2 Administratives

2.1 Verwendete Software

2.1.1 Spring Tool Suite

Spring Tool Suite ist eine Entwicklungsumgebung, welche auf Eclipse basiert. Sie beinhaltet bereits verschiedene Tools, welche die Entwicklung von Java-Anwendungen mit dem Spring-Framework vereinfacht.

Download Spring Tool Suite: <https://spring.io/tools>

2.1.2 Versionsverwaltung Programmcode (GIT)

GIT ist das im ÜK eingesetzte Tool für die Versionsverwaltung von Programmcode. Einerseits stellt der Dozent gewisse Unterlagen & Beispielprojekte über ein GIT-Repository zur Verfügung, andererseits müssen die verschiedenen Gruppen den Source Code ihrer Projekte über ein GIT-Repository austauschen und dem Dozent abgeben.

Download GIT: <https://git-scm.com/>

Download SourceTree: <https://www.sourcetreeapp.com/>

Download GitKraken: <https://www.gitkraken.com/>

2.2 Wichtige Webseiten

Nachfolgend ein paar Webseiten, welche im ÜK unterstützen können.

2.2.1 EN ISO 9241-110

Folgende Webseite bietet eine gute Übersicht über die Gestaltungsgrundsätze von Dialogen (GUIs) nach ISO 9241-110.

http://www.ergo-online.de/site.aspx?url=html/software/grundlagen_der_software_ergon/grundsaezze_der_dialog_gestalt.htm

2.3 Code-Beispiele / Demo-Projekte

Auf dem GitHub-Repository von Joel Holzer sind nachfolgende Code-Beispiele / Demo-Projekte abgelegt. Diese sind auch jeweils in den entsprechenden Kapiteln dieses Dokuments angegeben.

<https://github.com/joe-nyp/modul-223-examples>

Projekt	Beschreibung	Kapitel
HelloWorldRestAPI	Erstes Spring Boot Projekt – RESTful-API, welche „Hello NAME“ zurückgibt.	6
DbFirstRestAPI	DB Anbindung mit JPA und MySQL, Insert + GET aus 1 Tabelle.	7.2
DatabaseRestAPI	DB Anbindung mit JPA und MySQL. Tabellen mit Fremdschlüssel-Beziehungen. Einsatz von DTOs.	7.3
DbManyToManyRestAPI	DB Anbindung mit JPA und MySQL. m:n Beziehungen. Einsatz von DTOs.	7.4
DbCustomRepoRestAPI	DB Anbindung mit JPA und MySQL. Eigene Queries (Select, Update) absetzen mit Derived Queries und Named Queries.	7.5
ValidationRestAPI	Input-Validierung mit Bean Validation.	11
DBTransactionalRestAPI	DB Transaktion ausführen (Insert von mehreren Datensätzen)	12.6
LoginJWTRestAPI	REST-API mit Login-Funktion unter Einsatz von JWT.	9.4
AndroidRestApp	Android-App mit Login und User-Darstellung. Nutzt die LoginJWTRestAPI hochgeladen auf Heroku.	

Tabelle 2: Code-Beispiele / Demo-Projekte auf GitHub

3 Software-Architekturmuster im Überblick

3.1 Gruppenarbeit

Sie erarbeiten in einer Gruppenarbeit à 2 Personen eine Präsentation zu einer der folgenden SW-Architekturmuster:

- Client-Server Architektur / Peer-to-Peer
- Serviceorientierte Architektur
- Microservices
- Mehrschichten Architektur
- Model View Controller

Die Präsentation soll folgenden Inhalt aufweisen:

- Was charakterisiert die Architektur? Grafiken, Diagramme
- Unter-Arten / verwandte Architekturen
- Kommunikationstechnologien
- Stärken / Schwächen
- Einsatzbereich / Wann verwende ich diese Architektur

3.2 Ergebnisse der Gruppenarbeit

Die Ergebnisse der Gruppenarbeit sind auf dem GitHub-Repository (siehe Kapitel 2.3) abgelegt.

4 Analyse & Design von Applikationen

4.1 Einleitung

Ein Informatik-Projekt beginnt nicht bei der Programmierung, sondern beim ersten Gespräch mit dem Kunden. Eine fundierte Analyse und Design einer Applikation ist zwingend notwendig, damit der Kunde schlussendlich das erhält, was er sich auch wünscht. Sie kennen wahrscheinlich die nachfolgende Abbildung:

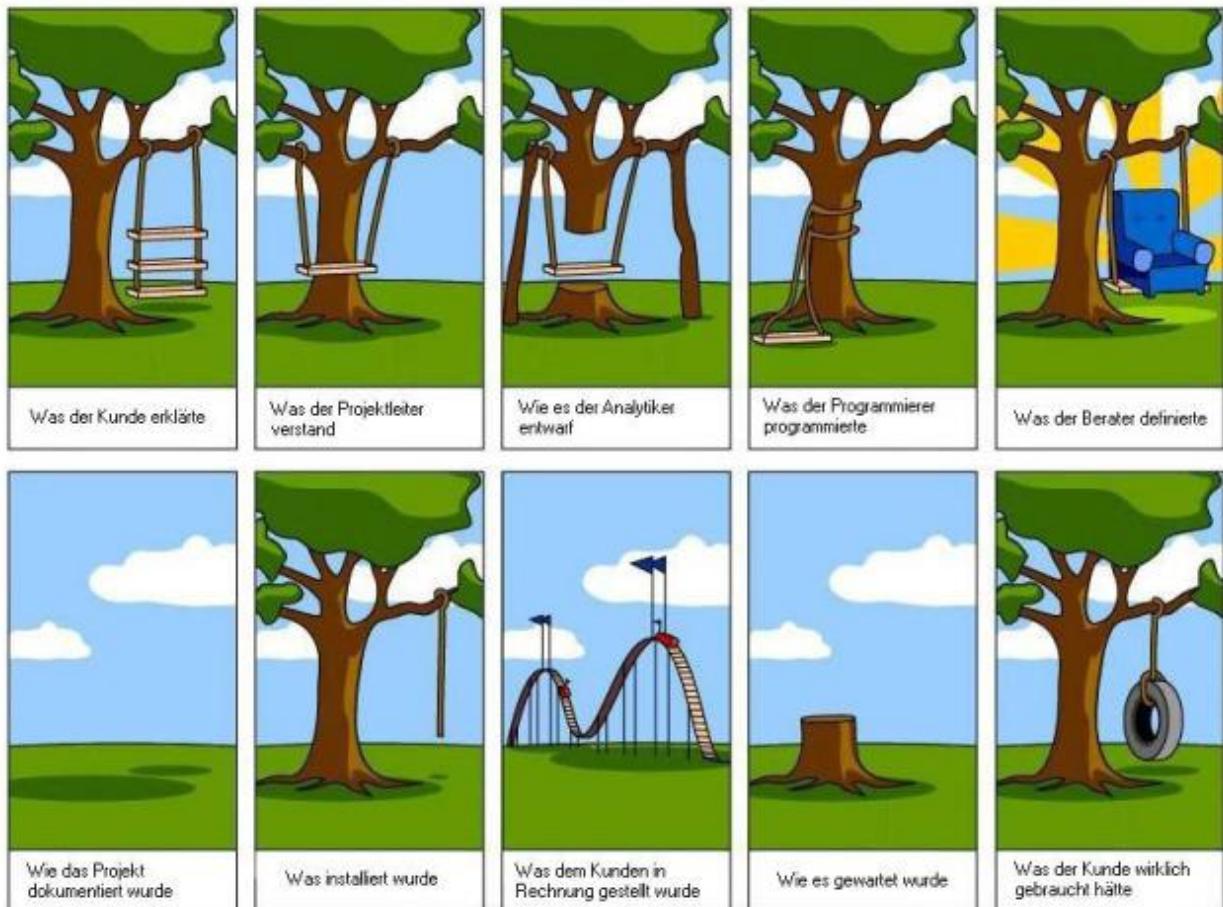


Abbildung 1: Analyse und Design von Applikationen

Dieses Kapitel erläutert einige wichtige Methoden zur Analyse und dem Design von Applikationen.

4.2 Domänenmodell

4.2.1 Was ist ein Domänenmodell?

Gleich zu Beginn eines Projektes sollte ein Domänenmodell erstellt werden. Dies soll sicherstellen, dass alle Beteiligten dieselben Grundlagen über die Zusammenhänge in der jeweiligen Branche verstehen.

Folgende Ratschläge sollten eingehalten werden:

1. Versteh das Business Modell
2. Verwende gute Namen für die Bezeichnung der Domänen-Modell-Klassen (klare Bedeutung, einmalig)
3. Erstelle das Domänen-Modell selbsterklärend
4. Das Domänen-Modell ist unabhängig von unterliegenden Funktionsänderungen

Das Domänenmodell ist noch sehr weit von der umzusetzenden Lösung und den Anforderungen weg. Es geht hier rein um die Business Analyse, d.h. zu verstehen in welchem Business sich der Kunde befindet und ein gemeinsames Verständnis / gemeinsame Sprache der Branche zu entwickeln.

Es empfiehlt sich, ein Domänenmodell zuerst relativ einfach auf Top-Level-Ebene zu erstellen, welches die Hauptentitäten und deren Beziehungen zueinander aufzeigt (Siehe Schritt 1 – Grobes Domänenmodell). Danach kann das Model verfeinert werden (siehe Schritt 2 – UML Klassendiagramm).

4.2.2 Was enthält ein Domänenmodell?

Immer:

- Konzeptuelle Problemklassen (keine Programmklassen). Dies kann beispielsweise bei einer Migros-Kassen-Applikation der Kunde, der Warenkorb oder der Artikel sein.
- Assoziationen zwischen den Klassen

Falls gewünscht (erst im UML Klassendiagramm):

- Attribute
- Klassenoperationen

4.2.3 Warum ein Domänenmodell erstellen?

Für Informatiker mit Programmier-Knowhow ist es häufig schwierig, nicht von Anfang an im ERD oder UML-Klassendiagramm zu denken, sondern auf der abstrakten Business-Ebene.

Dennoch ist es gerade bei grösseren Applikationen sinnvoll, zuerst das Domänenmodell zu erstellen. Das Domänenmodell bietet u.a. folgende Vorteile:

- Gemeinsame Sprache zwischen Kunde & Entwickler
- Entwickler versteht Business und Bedürfnisse des Kunden besser
- Von Anfang an der ganze Umfang und das Umfeld der Problemdomäne erkennen. Verhindert falsche Erwartungen oder eine falsche Zeitplanung.

Modul 223: Multi-User-Applikationen objektorientiert realisieren

4.2.4 Beispiel 1 – Grobes Domänenmodell für eine Bibliothek

Da Sie als Entwickler womöglich bei einer Informatik nahen Lösung sofort wieder im ERD oder Klassendiagramm denken, wird hier bewusst ein Informatik-fernes Thema aufgegriffen. Und zwar soll ein Domänenmodell für eine Bibliothek erstellt werden (Sinnvoll, wenn Sie z.B. eine Bibliothek-Verwaltungssoftware erstellen müssten).

Beispiel mit Beziehungen, aber ohne Multiplizität und Attribute:

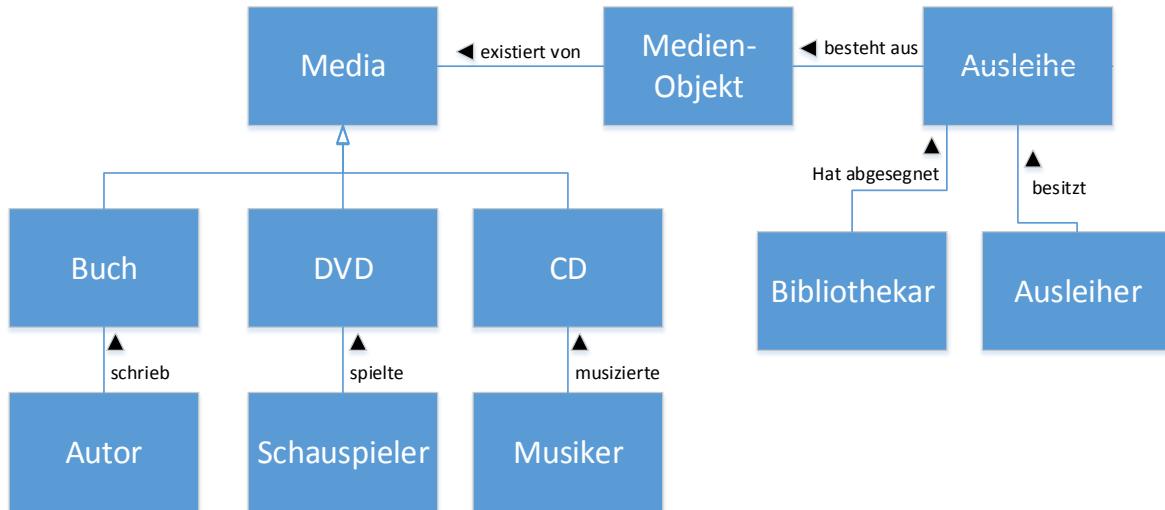


Abbildung 2: Grobes Domänenmodell für eine Bibliothek

4.2.5 Beispiel 2 – Grobes Domänenmodell für XelHa-Beobachtungsmodul

Nachfolgend ein grobes Domänenmodell für das XelHa-Beobachtungsmodul. Der Berufsbildner beobachtet die Lernenden während dem Semester. Stellt er ein Verhalten (Positiv oder Negativ) fest, dass er gerne festhalten würde, erstellt er mit XelHa eine Beobachtung. Damit die Beobachtungen mit dem Bildungsbericht verknüpft werden können, werden diese jeweils einer Kompetenz zugeordnet.

Beispiel mit Beziehungen und Multiplizität, ohne Attribute:

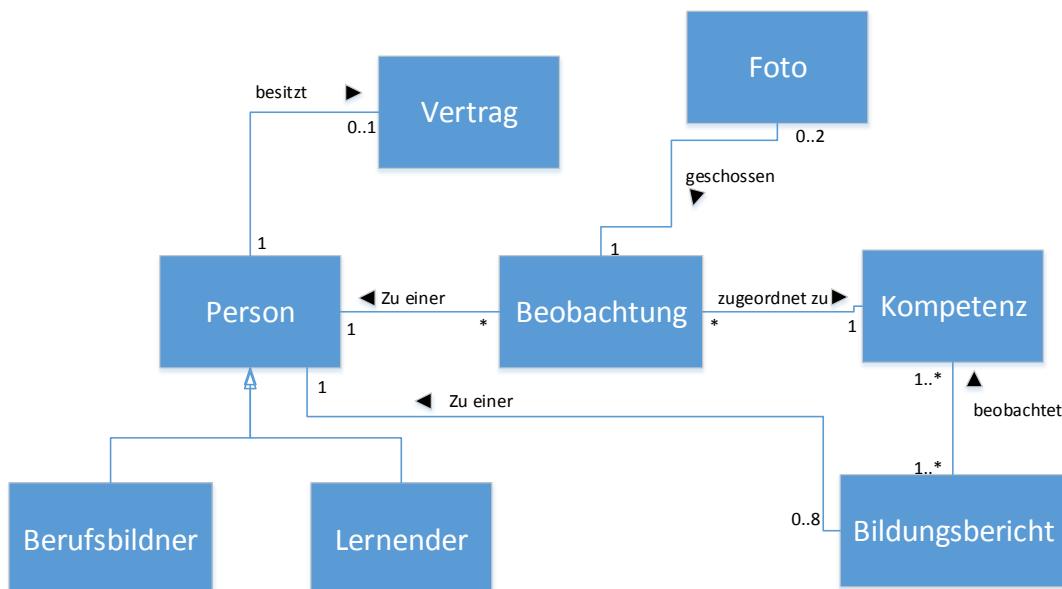


Abbildung 3: Grobes Domänenmodell für XelHa-Beobachtungsmodul

Modul 223: Multi-User-Applikationen objektorientiert realisieren

<https://www.java-forum.org/thema/was-ist-das-domain-model.127987/>

<https://baruzzo.wordpress.com/2010/08/29/data-object-models-difference/>

4.2.6 Beispiel 3 – UML Klassendiagramm

Das grobe Domänenmodell von zu Beginn des Projekts wird in einem späteren Verlauf der Analyse und/oder des Designs mehrfach verfeinert. Dies wird im Rahmen eines Klassendiagramms gemacht. Je nachdem auf welcher Ebene das Klassendiagramm gemacht wird (z.B. Domänenebene, Programmcode) ist dieses näher oder weiter vom Domänenmodell entfernt.

Nachfolgendes Klassendiagramm zeigt eine Verfeinerung des Domänenmodells „Bibliothek“ um Attribute. Umgesetzt als UML-Klassendiagramm. Dieses Diagramm ist jedoch in der Detailstufe noch nicht auf Ebene Programmcode oder ERD. Auf die Datentypen wurde verzichtet.

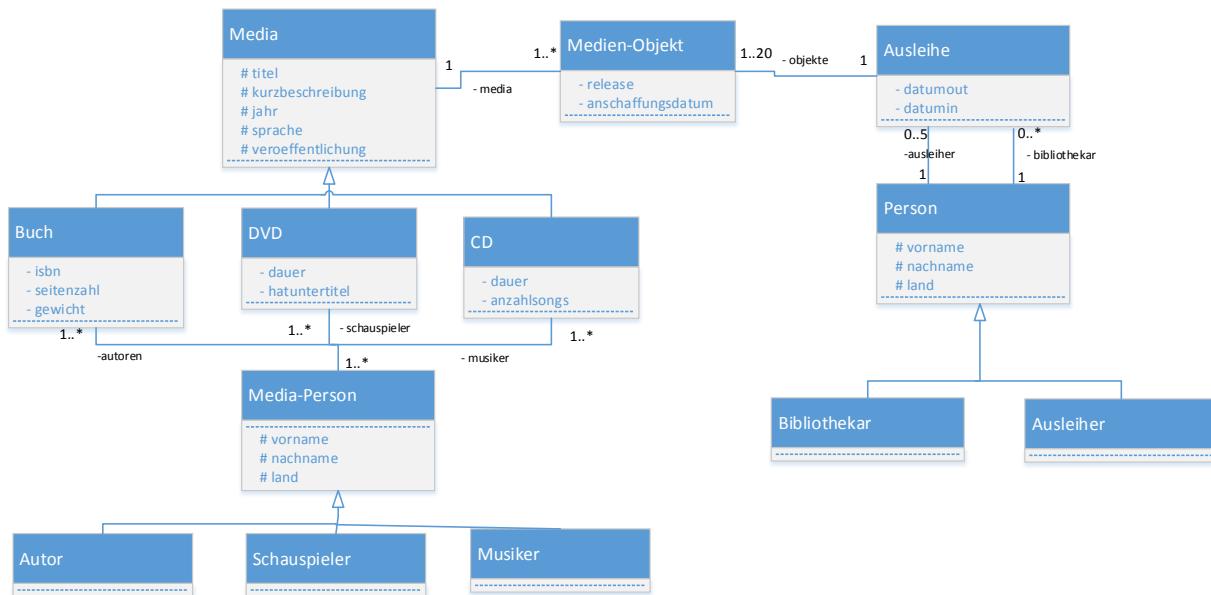


Abbildung 4: UML Klassendiagramm auf Domain-Ebene

4.3 Use Cases

4.3.1 Was sind Use Cases?

Der Hauptzweck eines Use Case Modells liegt darin, dass sich das Business wirklich mit den Anforderungen auseinandersetzen muss. Folgende Fragen stehen dabei im Zentrum:

- Wo sind die Systemgrenzen?
- Wer oder was verwendet das System?
- Welche Funktionalität soll das System den Akteuren zur Verfügung stellen?

Use Case Modelle sind ein Hilfsmittel zur generischen Anforderungsermittlung aus Nutzersicht. Es bietet einen guten ersten Überblick, indem man das System in grobe Hauptfunktionen gliedert und dient somit als hervorragende Diskussionsgrundlage sowie Basis zur weiteren Aufteilung und Detaillierung.

Um **systeminterne** Aktivitäten darzustellen sind sie NICHT geeignet.

4.3.2 UML Use Case Diagramm

4.3.2.1 Grundelemente

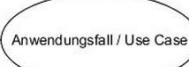
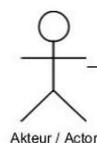
 Der **Systemkontext** wird durch Systemgrenzen in Form von Rechtecken gekennzeichnet.



Akteure werden als „Strichmännchen“ dargestellt, welche sowohl Personen wie Kunden oder Administratoren als auch ein System darstellen können. Sie beantworten die Frage: Wer oder was benutzt das System?



Anwendungsfälle werden in Eclipsen dargestellt und beschreiben „was macht ein Akteur mit dem System“.



Kommunikation von Akteur und Anwendungsfall.

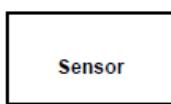
Um die Akteure in einem komplexeren System visuell zu unterscheiden, können allenfalls erweiterte Symbole eingesetzt werden. Die restliche Notation MUSS jedoch 1:1 eingehalten werden.



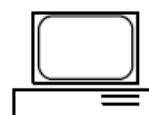
Person



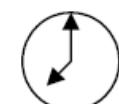
Akustisches Signal



Sensor



Computer



Zeitgesteuertes Ereignis

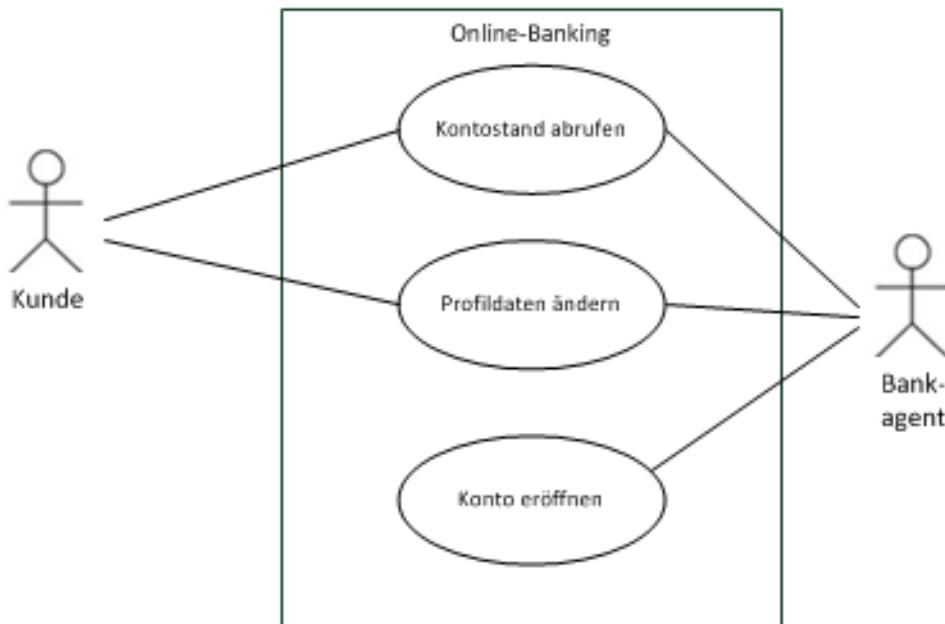
**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**
4.3.2.2 Beispiel Grundmodell


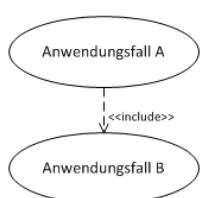
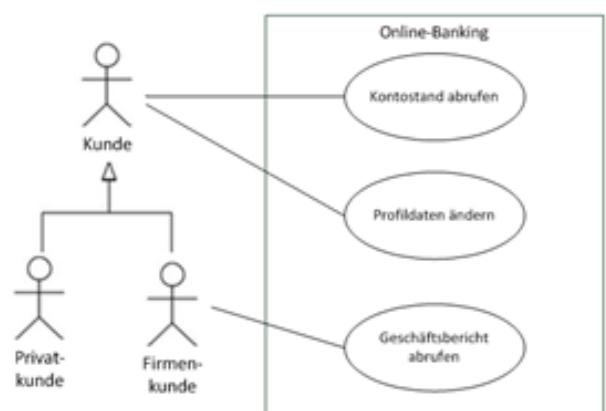
Abbildung 5: Beispiel Use Case Diagramm Grundmodell

4.3.2.3 Erweiterte Elemente

Generalisierung von Akteuren wird für Spezialisierungen benötigt, z.B. kann ein abstrakter Akteur „Kunde“ in „Privat-Kunde“ und „Firmen-Kunde“ spezialisiert werden.

Beispiel Generalisierung von Akteuren:

Dabei gilt, alle Kommunikationen vom abstrakten Akteur „Kunde“ gelten auch für seine Spezialisierungen „Privatkunde“ und „Firmenkunde“. Den Use Case „Geschäfts-bericht abrufen“ wird jedoch nur direkt von einem „Firmenkunden“ benutzt. Der „Privatkunde“ verwendet/ benötigt diese Funktion nicht.

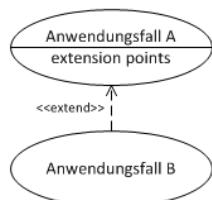
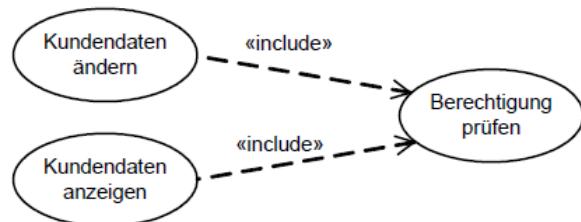

Include-Beziehung

Die <<include>>-Beziehung visualisiert, dass ein Use Case (Anwendungsfall A) das Verhalten eines anderen Use Case (Anwendungsfall B) importiert. Der inkludierte Anwendungsfall wird also IMMER aufgerufen.

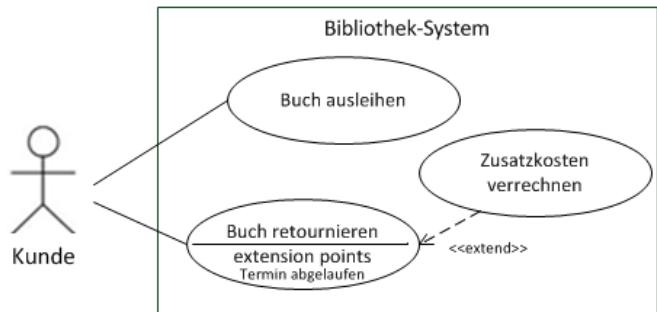
**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

Beispiel Include-Beziehung:

So importiert der Use-Case „Kundendaten ändern“ den Use-Case „Berechtigung prüfen“. Das bedeutet, dass während der Use-Case „Kundendaten ändern“ abläuft, in einem Ablaufschritt der Use-Case „Berechtigung prüfen“ aufgerufen wird, dann abläuft und sein Ergebnis in „Kundendaten ändern“ genutzt wird.


Extend-Beziehung

Die <<extend>>-Beziehung zeigt an, dass das Verhalten eines Use Case (Anwendungsfall A) durch einen anderen Use Case (Anwendungsfall B) erweitert werden kann, aber nicht muss. Ein Anwendungsfall kann mehrere „extension points“ beinhalten.



Beispiel Extend-Beziehung:

Ein Extend Use Case ist ein optionaler Aufruf des angegebenen Use Cases, wenn der Fall des „extension points“ eintritt. Wird ein Buch also zu spät retourniert, wird der Use Case „Zusatzkosten verrechnen“ aufgerufen.

4.3.2.4 Do & Don't's

- Benennung des Akteurs immer in Einzahl und NIE mit konkreten Namen (z.B. Frau Meier) sondern immer als Rolle
- Die Anwendungsfälle sollten mit Substantiv + Verb (Konto buchen, Raum planen, Benutzerdaten prüfen, ...) beschrieben werden
- Use Cases nicht zur detaillierten Beschreibung von Operationen oder Funktionen verwenden
- Keine nicht-funktionalen Anforderungen mittels Use Cases spezifizieren
- Nicht zu viele Use Cases modellieren. Als Richtwert gilt: wenn mehr als 10 Hauptfunktionen enthalten sind, wird das Modell zu komplex
- Sparsamer Umgang mit <<includes>> und <<extends>> da sie die intuitive Lesbarkeit einschränkt
- Verfeinern der Use Cases durch eine Beschreibung pro Use Case
- Für Standardfunktionalitäten wie Erfassen, Lesen, Ändern und Löschen von Daten kann auch ein „CRUD“ Use-Case erstellt werden (CRUD = Create, Read, Update, Delete). Dies vereinfacht die Lesbarkeit des Modells.

4.3.3 Use Case Beschreibung

Für jeden identifizierten Use Case wird nun eine textuelle Beschreibung angelegt. Ziel ist es, den Anwendungsfall vollständig zu beschreiben und einen leichten Zugriff auf bestimmte Aspekte zu ermöglichen. Für die Beschreibung sollte das folgende Template verwendet werden:

Use Case:	Eindeutiger Name für den Use Case. Geeignet ist Substantiv + Verb, da der Anwendungsfall beschreibt dass etwas geschieht.
Use Case ID:	Eindeutige Nummernvergabe als Identifikation.
Kurzbeschreibung:	Eine kurze (!!) Beschreibung, was der Use Case tut. Der Leser sollte beim Lesen eine Idee des Hauptinhaltes bekommen.
Vorbedingung:	Spezifizierte die Dinge, welche erfüllt sein müssen, bevor der Use Case starten kann. Verwende einfache aussagekräftige Sätze. Vermeide den Gebrauch von „und“ – es ist besser zwei separate Bedingungen aufzuführen. z.B. 1. Ein gültiger Benutzer ist am System eingeloggt
Akteur (Primary):	Liste der Akteure, welche einen Use Case anstoßen.
Akteur (Secondary):	Liste der Akteure, welche Informationen an einen Use Case übergeben oder Informationen von einem Use Case erhalten, diesen aber nicht anstoßen.
Hauptablauf:	Listet die einzelnen Schritte im Use Case auf. Der Use Case startet immer mit dem Akteur, der etwas tut, z.B. 1. Der Use Case startet wenn der <Akteur> <Funktion> 2. ... Es beinhaltet Abfolge von kurzen Schritten, numeriert und zeitgerecht. Der Hauptablauf entspricht IMMER dem „happy flow“ oder „perfect world“ Szenario wo alles wie erwartet abläuft OHNE Fehler, Unterbrüche oder Sonderfälle.
Nachbedingung:	Spezifizierte die Dinge, welche nach der Ausführung des Use Cases zutreffen. Verwende einfache Veränderungen am Akteur oder am System als Resultat des Anwendungsfalles. z.B. 1. Die Bestellung ist als bestätigt markiert und im System abgespeichert aussagekräftige Sätze. Nachbedingungen definieren die
Alternative Flows:	Auflistung der Alternativen Abläufe.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.3.3.1 Alternative Abläufe

Die Verwendung von **WHILE** und Alternativer Flow.

Use Case:	Kundenkonto eröffnen
Use Case ID:	5
Kurzbeschreibung:	Das System erstellt ein neues Kundenkonto.
Vorbedingung:	Keine
Akteur (Primary):	Kunde
Akteur (Secondary):	Keiner
Hauptablauf:	
1. Der Use Cases startet, wenn der Kunde „neues Kundenkonto erstellen“ anwählt 2. WHILE (SOLANGE) die Kundenangaben nicht korrekt sind 2.1. Das System fragt den Kunden erneut um Eingabe von Email-Adresse, Passwort und Bestätigung Passwort 2.2. Das System prüft die Kundeneingaben 3. Das System eröffnet ein neues Kundenkonto	
Nachbedingung:	Ein neues Kundenkonto wurde eröffnet.
Alternative Flows:	
UngültigeEmailAdresse	
UngültigesPasswort	

- Die ID des Alternativen Flows wird vom Hauptablauf abgeleitet
- Der Alternative Flow beginnt immer mit dem Einstiegspunkt aus dem Hauptablauf

Alternative Flow:	Kundenkonto eröffnen: UngültigeEmailAdresse
ID:	5.1
Kurzbeschreibung:	Das System informiert den Kunden, dass er eine ungültige Email Adresse eingegeben hat.
Vorbedingung:	Der Kunde hat eine ungültige Email Adress eingegeben
Akteur (Primary):	Kunde
Akteur (Secondary):	Keiner
Alternativer Flow:	
1. Der Alternative Flow beginnt nach dem Schritt 2.2 im Hauptablauf 2. Das System informiert den Kunden, dass die eingegebene Emailadresse ungültig ist	
Nachbedingung:	Keine

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.3.3.2 Iterationen innerhalb eines Ablaufes

 Die Verwendung von **FOR** und **WHILE**.

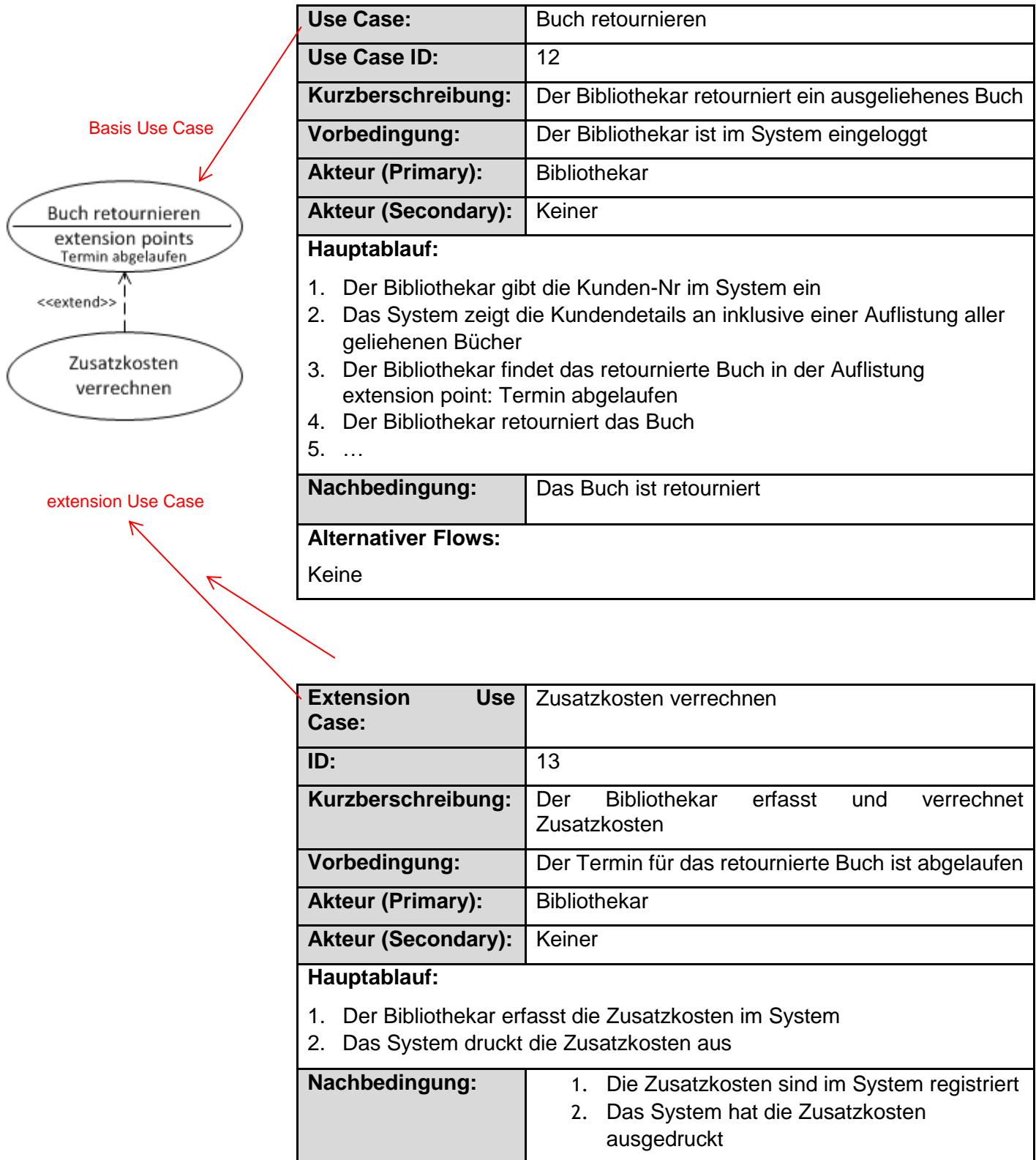
Use Case:	Produkt finden
Use Case ID:	3
Kurzbeschreibung:	Das System findet Produkte basierend auf den Suchkriterien vom Kunden und zeigt sie diesem an.
Vorbedingung:	Keine
Akteur (Primary):	Kunde
Akteur (Secondary):	Keiner
Hauptablauf:	
<ol style="list-style-type: none"> 1. Der Use Case startet wenn der Kunde „Produkt suchen“ anwählt 2. Das System fragt den Kunden nach Suchkriterien 3. Der Kunde gibt angefragte Kriterien an 4. Das System sucht Produkte, welche die Suchkriterien des Kunden erfüllen 5. FOR (FÜR) jedes gefundene Produkt <ol style="list-style-type: none"> 5.1. Das System zeigt ein Summary des Produkts an 5.2. Das System zeigt den Preis an 	
Nachbedingung:	Keine
Alternativer Flows:	
Keine	

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.3.3.3 Includes innerhalb eines Ablaufes

Use Case:	<i>Mitarbeiterdaten ändern</i>
Use Case ID:	7
Kurzbeschreibung:	<i>Der Vorgesetzte ändert die Mitarbeiterdaten.</i>
Vorbedingung:	<i>Der Vorgesetzte ist am System eingeloggt.</i>
Akteur (Primary):	Vorgesetzter
Akteur (Secondary):	Keiner
Hauptablauf:	
<ol style="list-style-type: none"> 1. <i>Include (MitarbeiterdatenFinden)</i> 2. Das System zeigt die Mitarbeiterdaten an 3. Der Vorgesetzte ändert die Mitarbeiterdaten 4. ... 	
Nachbedingung:	Die Mitarbeiterdaten wurden geändert
Alternativer Flows:	
Keine	

Use Case:	Mitarbeiterdaten finden
Use Case ID:	9
Kurzbeschreibung:	Der Vorgesetzte findet die Mitarbeiterdaten
Vorbedingung:	Der Vorgesetzte ist am System eingeloggt
Akteur (Primary):	Vorgesetzter
Akteur (Secondary):	Keiner
Hauptablauf:	
<ol style="list-style-type: none"> 1. Der Vorgesetzte gibt die Mitarbeiter-Nr ein 2. Das System findet die Mitarbeiterdetails 	
Nachbedingung:	Das System hat die Mitarbeiterdaten gefunden
Alternativer Flows:	
Keine	

4.3.3.4 Extends innerhalb eines Ablaufes


4.4 User Interface Design

Eine Benutzeroberfläche mit der Entwicklungsumgebung umsetzen ist die eine Sache, dass diese Benutzeroberfläche jedoch nutzerorientiert und intuitiv daher kommt, eine andere. Als Designer und Entwickler von Benutzeroberflächen gilt es einiges zu beachten. So sollten einerseits bei jeder Gestaltung eines Software-GUIs, egal ob für PC oder mobilen Geräte, gewisse ergonomische Grundsätze eingehalten werden. Zum anderen muss sich jeder Designer/Entwickler bewusst sein, dass die Darstellungsmöglichkeiten auf mobilen Geräten limitiert sind und verschiedene mobile Gerätetypen mit unterschiedlichen Bildschirmgrößen/Auflösungen existieren.

4.4.1 Ergonomische Standards

Die Norm EN ISO 9241 ist ein internationaler Standard, welcher Richtlinien zur Interaktion zwischen Mensch und Computer beschreibt. Diese Norm beinhaltet verschiedene Teile. Interessant für die Gestaltung von Benutzeroberflächen ist der Teil 110 (DIN EN ISO 9241-110) dieser Norm. Diese sagt folgendes:

- **Aufgabenangemessenheit:** Unnötige Interaktionen sollen minimiert werden. Das Fenster und die in diesem Fenster angezeigten Elemente und Texte sollen zur Erledigung der Aufgabe beitragen.
- **Selbstbeschreibungsfähigkeit:** Texte, Meldungen, etc. sollen auf Anhieb verständlich und selbstbeschreibend sein. Der Benutzer soll mit Rückmeldungen über Systemaktionen informiert werden.
- **Lernförderlichkeit:** Die Bedienschritte, Tastenkürzen, der GUI-Aufbau oder Menüeinträge sollen einem einheitlichen, leicht zu verstehenden Prinzip folgen mit dem Ziel: minimale Erlernzeit.
- **Steuerbarkeit:** Schaltflächen, Icons und Menüeinträge sollten den Benutzern mit einfachen und flexiblen Dialogwegen zum Ziel der Aufgabe führen. Bedienungsschritte sollten aufhebbar oder rückgängig gemacht werden können.
- **Erwartungskonformität:** Die Abläufe, Symbole und die Anordnung von GUI-Elementen sollen innerhalb der Anwendung konsistent sein.
- **Individualisierbarkeit:** Fenstereinstellungen, Sortierungen, Symbolleisten, Menüs, Tastenkürzen, Funktionstasten, etc. sollen individuell an die Bedürfnisse und Kenntnisse des Benutzers angepasst werden.
- **Fehlertoleranz:** Das System sollte helfen, Fehler durch Personen möglichst zu vermeiden und Korrekturmöglichkeiten anbieten.

4.4.2 Checkliste zur GUI-Darstellung

- Ist das GUI angemessen farbig, nicht zu viel und nicht zu langweilig?
- Werden die Farben konistent verwendet, erkennt man intuitiv ein Konzept, wofür die Farben stehen?
- Ist die Sprache im GUI einheitlich?
- Sind die Inhalte logisch gruppiert?
- Sind die Inhalte auf dem Fenster logisch positioniert?
- Ist eine inhaltlich logische Reihenfolge intuitiv erkennbar (intuitive Benutzerführung)?
- Lässt sich das Fenster in der Grösse anpassen?
- Können noch alle Felder und Buttons im Fenster angewählt werden, bei einer Anzeige auf einem Bildschirm mit minimaler Auflösung z.B. 1280x720 oder 1378x768?
- Sind die Buttons in einheitlichen Grössen gehalten? Zumindest in der Höhe?
- Sind die Felder gross genug für den erwarteten Inhalt?
- Sind wichtige Felder/Buttons eher grösser als unwichtige?
- Unterscheiden sich Felder zur Anzeige von solchen zur Eingabe?
- Sind Muss-Eingaben speziell gekennzeichnet?
- Sind die Felder und Buttons nach optischen Linien ausgerichtet?
- Haben die Felder und Buttons einheitliche Abstände vom Fensterrand?
- Sind die Eingabeelemente (Felder, Checkboxen etc.) mit sprechenden Bezeichnern (Labels) versehen?
- Sind die Bezeichner (Labels) einheitlich angeordnet, z.B. rechtsbündig, linksbündig, oben unten?
- Haben die Bezeichner am Ende konsequent einen Doppelpunkt oder nicht?
Z.B. "Vorname:" "Nachname:" oder dann "Vorname" "Nachname"
- Haben Felder mit der gleichen semantischen Bedeutung einheitliche Bezeichner?
z.B. nicht einmal Name, Vorname dann Nachname und Vorname
- Haben Felder mit der gleichen semantischen Bedeutung einheitliche Reihenfolgen?
z.B. nicht einmal Nachname, Vorname dann Vorname, Nachname
- Sind die Standardfunktionen (z.B. OK, Schliessen, Abbrechen) einheitlich und intuitiv auf dem Fenster angeordnet?
- Haben alle aktvierten Buttons und Felder die erwartete Funktion?
- Sind jene Buttons deaktiviert oder nicht sichtbar, welche im aktuellen Zustand nicht verwendet werden können oder keine Funktion haben?
- Ist erkenn- oder erahnbar, wodurch deaktivierte Buttons wieder aktiv werden?
- Sind Inhalte in Tabellen oder Auswahlboxen nach der ersten Spalte sortiert?
- Haben Menüeinträge, welche ein weiteres Fenster öffnen, die Endung "..."?

4.4.3 Checkliste GUI Handling

- Ist das Fenster auch rein über die Tastatur bedienbar?
- Sind die Felder über die Tab-Taste in der logischen Reihenfolge anwählbar?
- Gibt es ein Konzept zum Standardverhalten wenn die Enter-Taste gedrückt wird? (Default-Button, z.B. immer auf OK oder Abbrechen)
- Kann der Haupt-Anwendungsfall mit möglichst wenigen Klicks abgewickelt werden?
- Hat der Benutzer innerhalb von 200ms eine Rückmeldung über seine Eingabe? z.B. Sanduhr, aktiverter Button
- Haben Datums- und Zeitfelder einen sinnvollen Voreinstellung?
z.B. bei Datum den aktuellen Tag, den ersten des Monats, eine Zeit mit Minuten auf 00
z.B. Geburtstage: macht dann das aktuelle Datum als Default wirklich Sinn?
- Können aufgrund der vorherigen An-/Eingaben des Benutzers gewisse Voreinstellungen gemacht werden?
Z.B. ableiten der Email-Adresse aus Vor- und Nachname
- Speichern ohne Schliessen des Fensters: Speichern oder Save oder *Apply*
- Schliessen des Fensters mit Speichern ohne Programm beenden: OK
- Schliessen des Fensters ohne Speichern ohne Programm beenden: Schliessen oder Close
- Abbrechen des Dialogs ohne Speichern ohne Programm beenden: Abbrechen oder Cancel
- Schliessen des Fensters und beenden des Programms: Beenden oder Exit oder Quit
- Sind die Menüs logisch und unkompliziert aufgebaut?
- Sind die Menüeinträge eindeutig und nicht mehrfach vorhanden?
- Enthalten Menüpunkte die Funktion, die unter der Bezeichnung vom Benutzer erwartet wird?
- Können die Voreinstellungen angepasst/konfiguriert werden?
- Wird beim Löschen eines Eintrags eine Bestätigung verlangt?
- Bleiben die eingegebenen Daten erhalten, wenn im Assistent/Wizard zurück und vorwärts gesprungen wird?
- Können in Dialogfolgen mit mehreren Dialogen die Änderungen/Angaben rückgängig gemacht werden?
- Folgen Checkboxen positiver Logik folgen, d.h. wenn angewählt, dann wird etwas getan und nicht weggelassen?
z.B. Checkbox "logging" ist besser als "no logging"

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.4.4 Checkliste GUI Fehlerhandlung

- Werden Fehler nach einem einheitlichen Konzept dargestellt?
- Sind Fehlermeldungen für den Benutzer verständlich und hilfreich?
z.B. kann der Benutzer seinen Fehler sinnvoll korrigieren?
- Sind die Fehlermeldungen für den Benutzer freundlich formuliert, z.B. *keine Ausrufezeichen*?
- Ist die Anwendung fehlertolerant und robust?
- Stimmen bei Fehlermeldungen die Bezeichnungen der Buttons mit den Auswahlmöglichkeiten überein?
z.B. die Fehlermeldung sagt OK drücken, als Button steht jedoch Ja und Nein zur Auswahl.

4.4.5 Checklisten im Internet

Nachfolgend eine Sammlung von Links, welche Checklisten im Internet zur Verfügung stellen:

4.4.5.1 Checklisten auf Deutsch:

- <https://apliki.de/2013/11/25/die-mutter-aller-usability-checklisten/>
- http://tobiasiordans.de/archivierte-arbeiten-fachhochschule-aachen/referate/Referat_Jeff-Johnson_GUI-Bloopers.html

4.4.5.2 Checklisten auf Englisch

- <https://www.smashingmagazine.com/2009/06/45-incredibly-useful-web-design-checklists-and-questionnaires/>
- <http://www.usability.gov/what-and-why/user-interface-design.html>
- <http://bloq.teamtreehouse.com/10-user-interface-design-fundamentals>
- <http://www.gui-bloopers.com/checklist.php>

4.5 Mockups erstellen

4.5.1 Was sind Mockups? Warum Mockups?

Bevor ein GUI umgesetzt wird, sollte zumindest ein Mockups erstellt werden. Dies ist eine Zeichnung, wie das GUI aufgebaut sein soll und welche GUI-Elemente verwendet werden.

Anhand des fertigen Mockups, können also die folgenden Fragen geklärt werden:

- Aus welchen Inhalten setzt sich die Seite zusammen?
- Wo werden die Inhalte auf der Seite platziert?
- Welche Interaktionsmöglichkeiten werden dem User bereitgestellt?
- usw.

Weitere Vorteile von Mockups:

- **Geringer Aufwand:**

Mit der Hilfe eines Mockups, erhalten Sie viel schneller einen visuellen Eindruck von der zukünftigen Seite. Damit haben Sie im Handumdrehen ein präsentierbares Design.

- **Grundlage für Diskussionen:**

Haben Sie schon mal versucht ein Bild/Design zu beschreiben? Ich schon und ich kann Ihnen sagen, dass ist nicht so einfach. Jeder stellt sich etwas anderes vor und erst auf einen Prototypen zu warten, kann viel Zeit und Nerven kosten. Deshalb ist es wichtig so schnell wie möglich einen ersten visuellen Entwurf zu haben und genau dafür ist das Mockup gedacht.

4.5.2 Storyboard zur Präsentation von Mockups

Ein Storyboard ist eine gute Möglichkeit, die Mockups und auch die Zusammenhänge der einzelnen Seiten auf einer Grafik darzustellen. Nachfolgend ein Beispiel eines Storyboards.

Ein gutes Storyboard charakterisiert sich durch folgende Eigenschaften:

- Alle Ansichten sind vorhanden (auch für alle User-Rollen).
- Alle Abläufe (Erfolgsfälle, Alternativfälle) sind festgehalten.
- Nicht verständliche Felder und Verhalten sind durch Kommentare erläutert. Z.B. welche Einträge sind in einer ComboBox vorhanden.
- Systemrückmeldungen (Fehlermeldungen, Erfolgsmeldungen, Warte-Meldungen) sind ersichtlich.
- Falls unterschiedliche Darstellung je nach Bildschirmgrösse: Mockups für alle Ausprägungen vorhanden.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

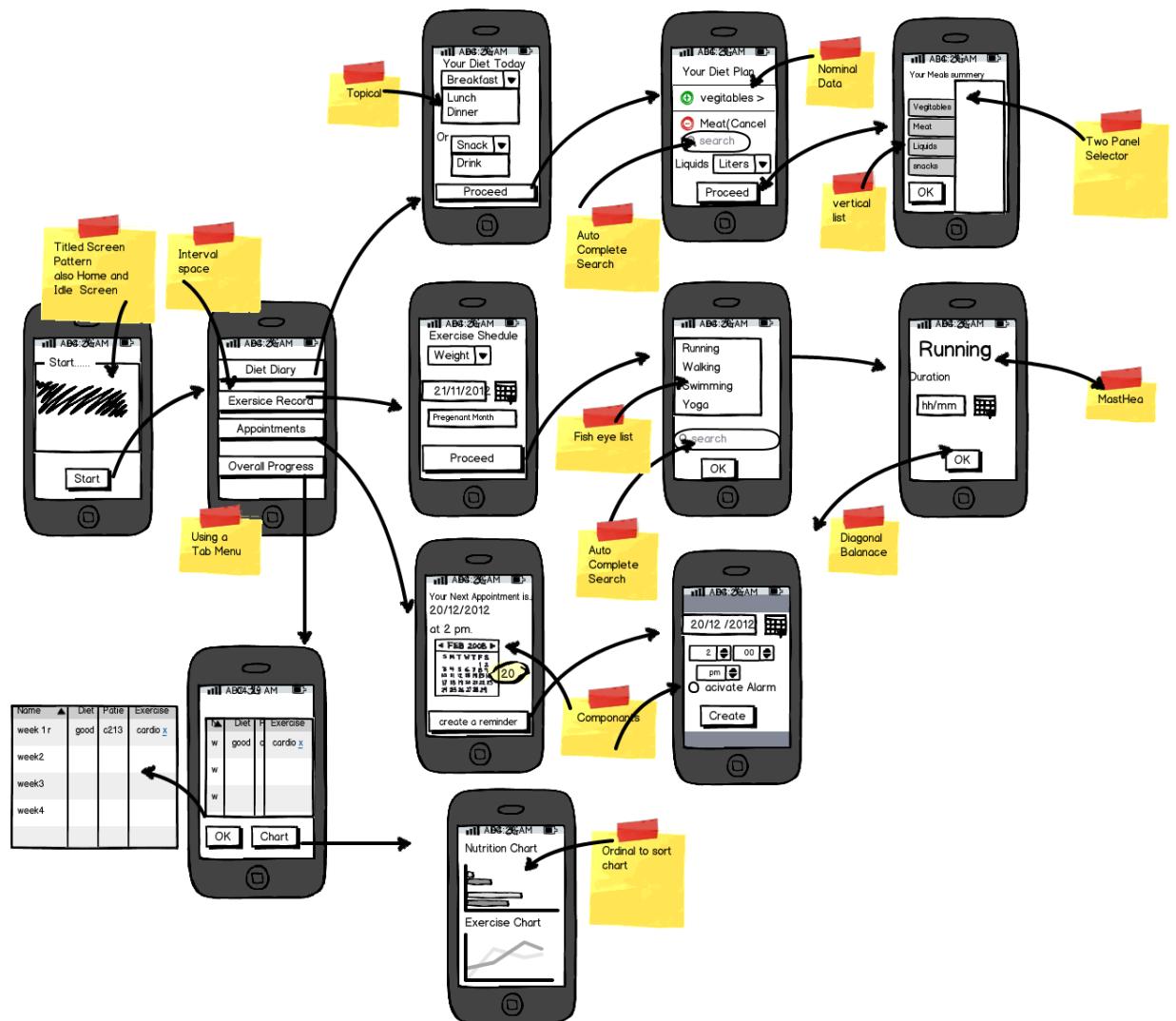


Abbildung 6: Beispiel Storyboard

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.6 Entity Relationship Diagram (ERD)

Im ERD wird die Datenbank designt, d.h. die Tabellen, Attribute, Datentypen, Beziehungen und Multiplizitäten festgelegt. Basierend auf dem ERD wird dann die Datenbank erstellt. Mit Tools wie MySQL-Workbench ist es sogar möglich, die Datenbank aus dem ERD automatisch zu generieren.

Das ERD kann aus dem Domänenmodell abgeleitet werden.

4.6.1 Vom Domänenmodell zum ERD

Das Domänenmodell ist eine abstrakte Repräsentation der Business-Klassen. Es bietet jedoch bereits eine gute Grundlage für die Erstellung des ERD. Gewisse Business-Klassen können fast 1:1 in Entitäten abgeleitet werden. Andere Business-Klassen werden jedoch auf mehrere Entitäten aufgeteilt.

Im ERD werden folgende Eigenschaften gegenüber dem Domänenmodell verfeinert:

- Konkrete Beziehungen definiert
- Konkrete Attribute inkl. Datentypen und Standardwerten definiert
- Konkrete Tabellen definiert und keine abstrakten Klassen mehr.
- Zwischentabellen eingefügt für n:m Beziehungen

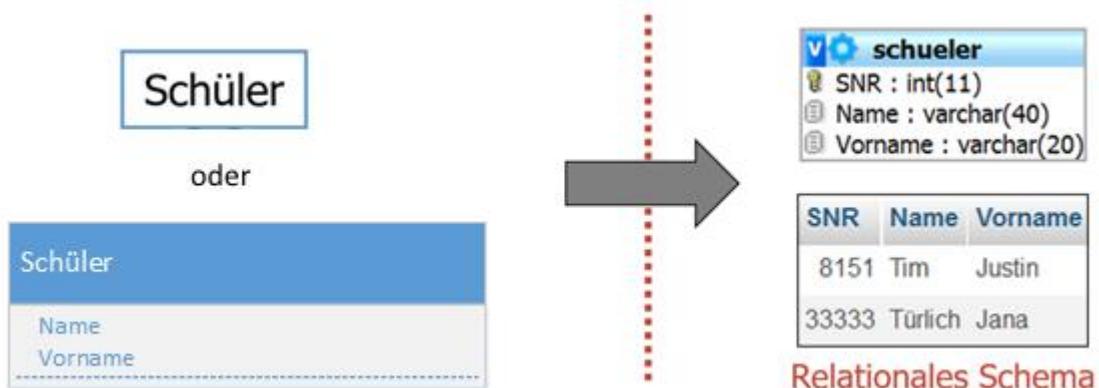


Abbildung 7: Vom Domänenmodell zum ERD

4.6.2 1:n Beziehungen

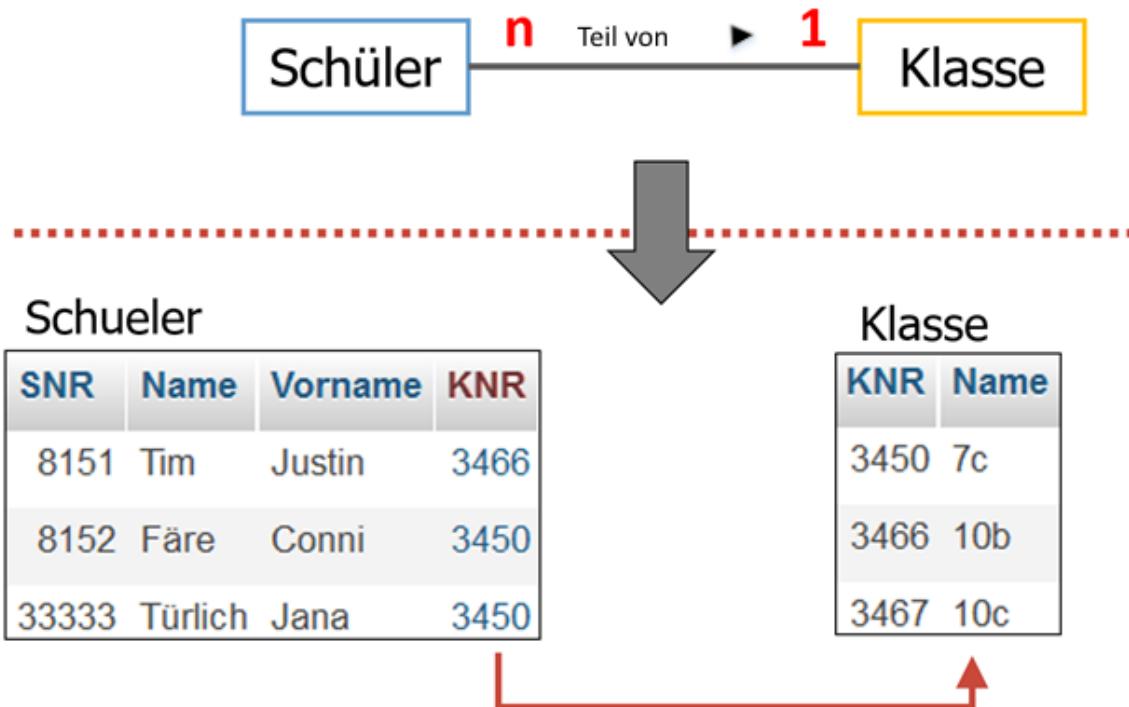


Abbildung 8: 1:n Beziehung Domänenmodell vs. ERD

4.6.3 n:m Beziehungen

Bei n:m Beziehungen wird im ERD eine Zwischentabelle hinzugefügt.

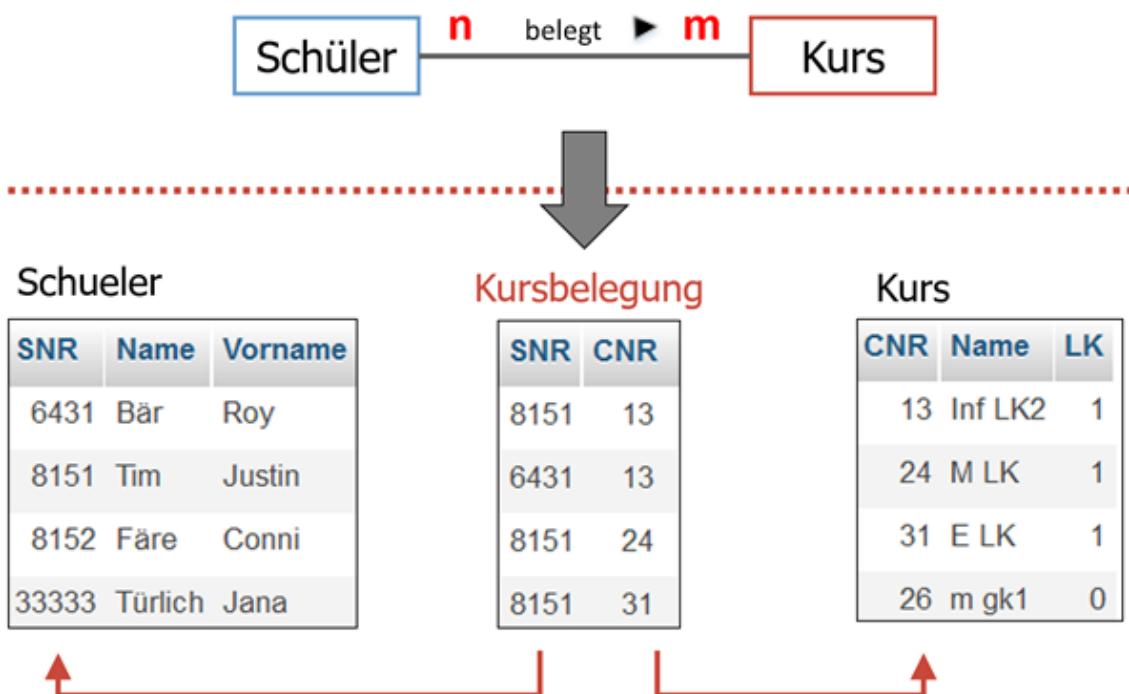


Abbildung 9: n:m Beziehungen

4.7 UML Klassendiagramm

Klassendiagramme können in verschiedenen Phasen eines Software-Projekts ihre Daseinsberechtigung haben, seien es in der Analyse, im Design, Realisierung etc. In der Analysephase kann z.B. ein Domänenmodell mit einem Klassendiagramm der Business-Klassen verfeinert werden. In der Designphase kann dann ein Design-Klassendiagramm erstellt werden, welches die konkret zu erstellenden Klassen, z.B. Models, DB-Klassen, oder GUI-Klassen beinhaltet. In der Realisierungsphase kann ein Klassendiagramm zur Dokumentation des realisierten Codes dienen.

Einsatz in verschiedenen Projektphasen:

- **Analyse:** Domänenmodell
- **Design:** Wie soll die Software realisiert werden? Klassendesign.
- **Implementierung:** Verfeinerung des Designs.
- **Test:** Als Referenz dienen.
- **Einsatz/Wartung:** Übersicht erhalten, Dokumentation

4.7.1 Beispiel eines UML Klassendiagramms

Nachfolgend ein einfaches Beispiel. Dies ist ein Teil einer Bankenapplikation.

Es gibt 2 Arten von Kunden, Privatkunde und Geschäftskunde. Jeder Kunde verfügt über eine Adresse und über mindestens ein Konto.

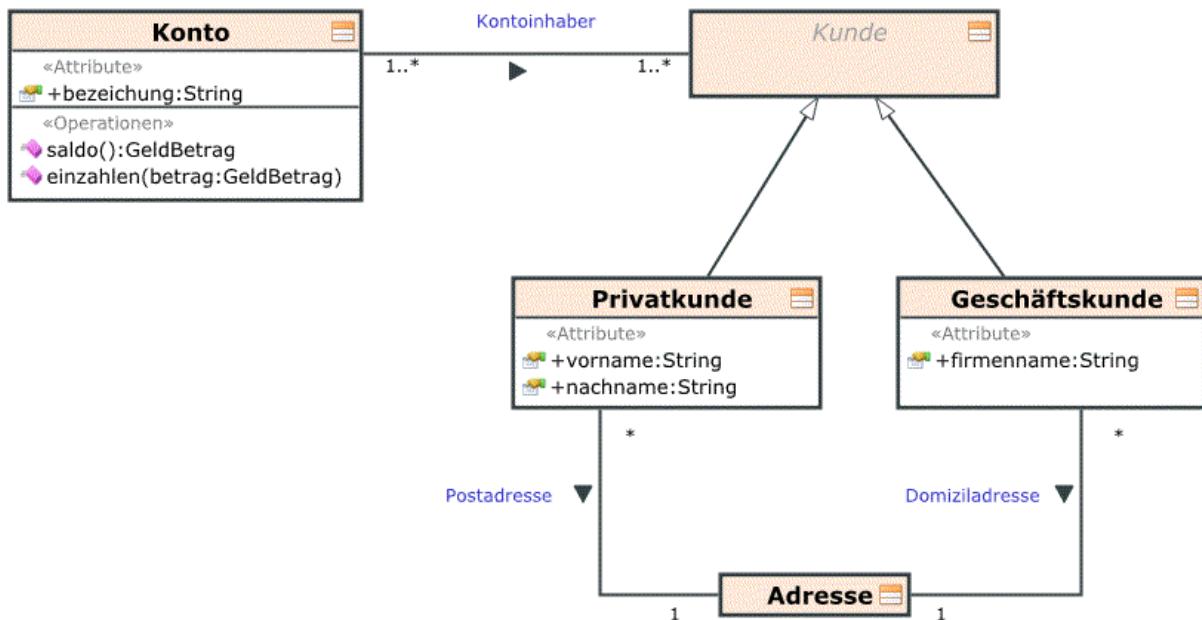


Abbildung 10: Beispiel UML Klassendiagramm

4.7.2 Elemente eines UML Klassendiagramms

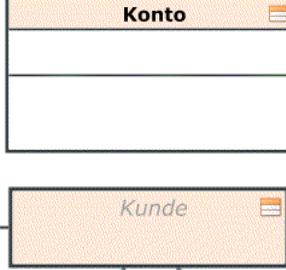
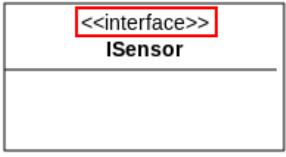
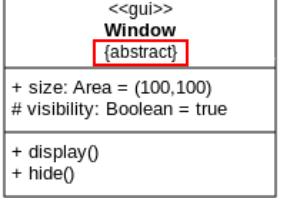
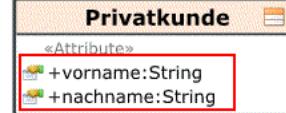
Element	Bild	Beschreibung
Klasse		Vorschrift / Bauplan
Interface		Interface.
Klasseneigenschaften		z.B. Abstract Klasse
Attribut		Eigenschaften einer Klasse, die dadurch von anderen Objektklassen unterschieden wird.
Methode		Reaktion auf Botschaft / Aktivität
Scihtbarkeit		Sichtbarkeit von Attributen und Methoden. <ul style="list-style-type: none"> - + für public - # für protected - - für private - ~ für package
Datentyp		Datentyp von Attributen, Parameter oder Rückgabewerten von Methoden.
Beziehungen		Siehe Kapitel 0

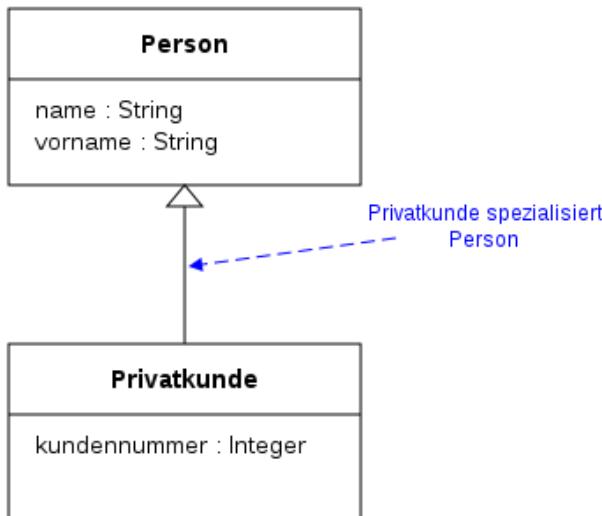
Tabelle 3: Elemente eines UML Klassendiagramms

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.7.3 Beziehungen

4.7.3.1 Generalisierung

Entspricht der Vererbung im Programmcode. Privatkunde erbt von Person und erhält daher alle Attribute der Person und hat zusätzlich noch sein eigenes Attribut.



Der Generalisierung-Pfeil hat folgendes Aussehen:

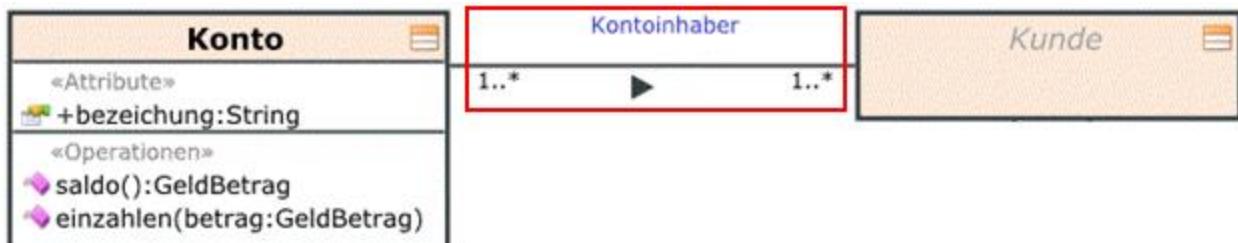


4.7.3.2 Assoziation

Beziehung zwischen zwei oder mehreren Klassen. An den Enden von Assoziationen sind häufig die Multiplizitäten (1..*, etc. vorhanden).

Im Beispiel ist die Beziehung zwischen Konto und Kunde festgehalten. Ein Kunde kann mehrere Kontos haben, wobei auch ein Konto mehrere Kunden haben kann (n:m-Beziehung).

Häufig trägt die Assoziation noch einen Namen. Dies kann z.B. der Name des Attributes sein, unter welchem das Fremdschlüssel-Objekt in der Java-Klasse hinterlegt wird.



4.7.3.3 Gerichtete Assoziation

Mit einem Pfeil an der Assoziation kann die Navigationsrichtung angegeben werden. Der Pfeil drückt die Zugriffsrichtung der Objekte aus. Objekt A greift auf B zu, B greift nie auf A zu.

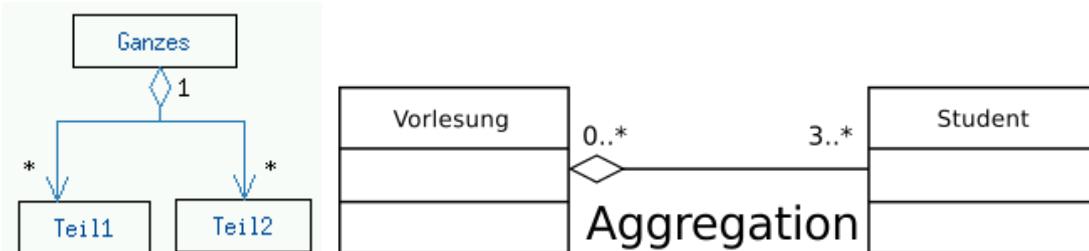


**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.7.3.4 Aggregation

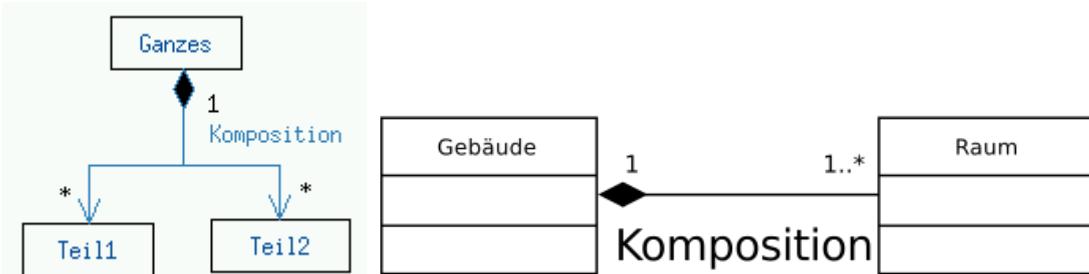
Eine Aggregation drückt eine Teile-Ganze-Beziehung aus. Das Ganze-Objekt besteht aus Teil-Objekten. Die Raute befindet sich an dem Ende des Ganzen. Die Aggregation ist eine spezielle Art der Assoziation.

Ein Student ist ein Teil einer Vorlesung. Der Student existiert jedoch weiter, auch wenn die Vorlesung nicht mehr existiert (stattfindet).


4.7.3.5 Komposition

Die Komposition ist auch eine Beziehung, die Teile zu einem Ganzen in Beziehung setzt. Die Teile und das Ganze sind bei dieser Beziehung existenzabhängig; die Teile können nicht ohne das Ganze existieren. Wird das Ganze gelöscht, so beenden auch die Teile ihre Existenz.

Ein Raum kann nicht existieren, wenn kein Gebäude existiert.



**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.8 UML Sequenzdiagramm

Sequenzdiagramme modellieren Interaktionen zwischen Objekten. Sie stellen Abläufe dar und konzentrieren sich auf den Nachrichtenaustausch zwischen Klassen.

Sequenzdiagramme können wie Klassendiagramme auch in verschiedenen Projektphasen vorkommen:

Einsatz in verschiedenen Projektphasen:

- **Analyse:** Abbildung der Use Cases mit System Sequenzdiagrammen (SSD).
- **Design:** Wie soll die Software realisiert werden?
- **Implementierung:** Verfeinerung des Designs.
- **Einsatz/Wartung:** Dokumentation. Wie wurde eine bestimmte Funktionalität umgesetzt.

4.8.1 System Sequenzdiagramm (SSD)

Ein SSD zeigt den Ablauf eines Use Cases in einem Sequenzdiagramm. Auf der einen Seite gibt es den Akteur, auf der anderen Seite das System. SSD können helfen einen schnellen Überblick über den Ablauf eines Use Cases zu erhalten (ohne Text zu lesen).

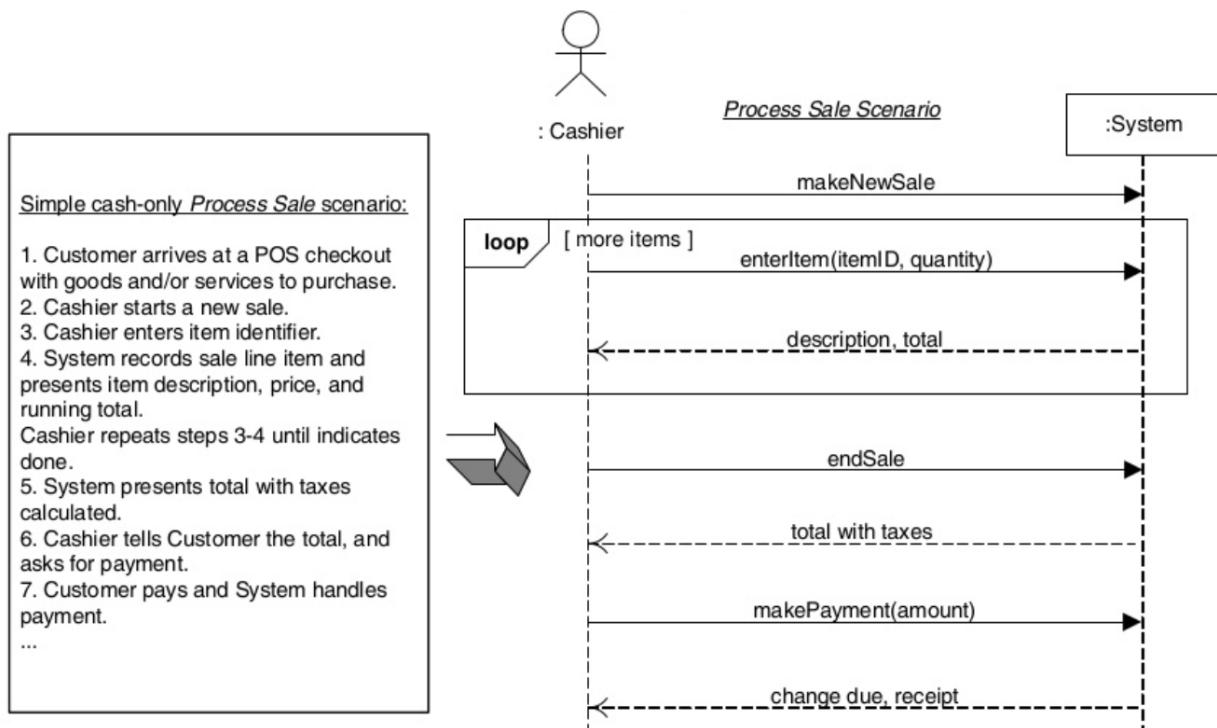


Abbildung 11: System Sequenzdiagramm (SSD)

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

4.8.2 Sequenzdiagramm auf Klassenebene für SW-Design / Realisierung

In einem Sequenzdiagramm zum SW-Design oder zur Realisierung sollen die Interaktionen zwischen 2 und mehr Objekte bildlich dargestellt werden, d.h. Welche Klasse ruft welche Methode auf und was gibt diese dann zurück.

Ein Sequenzdiagramm in der Design-Phase ist hilfreich, um sich bewusst zu werden, welche Methoden eine Klasse implementieren muss oder um Architekturfehler (z.B. starke Kopplung von Klassen) zu erkennen.

Ein Sequenzdiagramm in der Realisierung, respektive Dokumentation kann neuen Mitarbeitern die Einarbeitung in ein Projekt vereinfachen um bestimmte Abläufe besser zu verstehen.

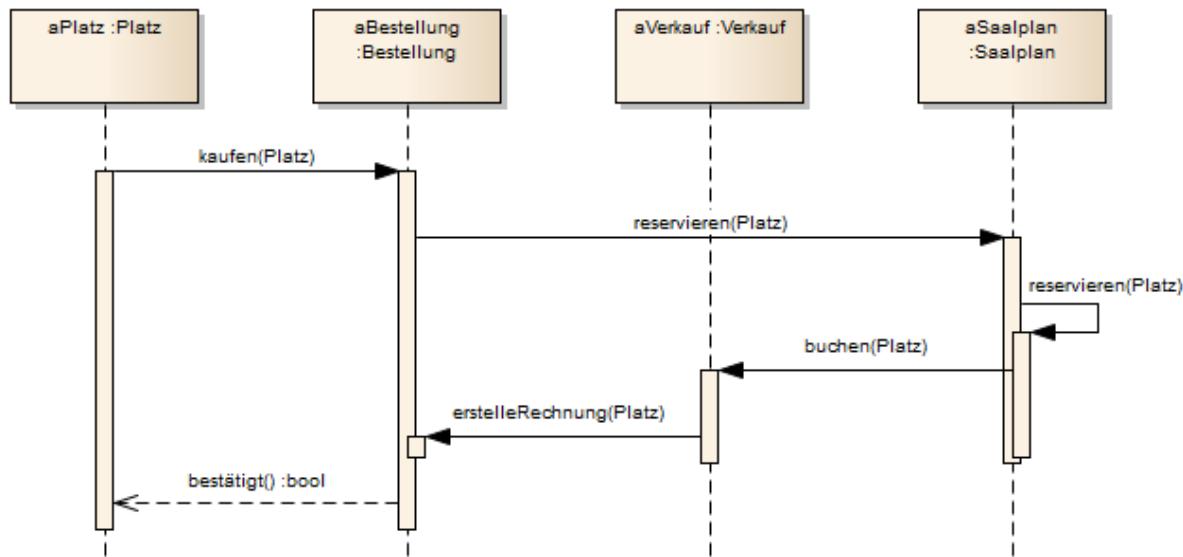
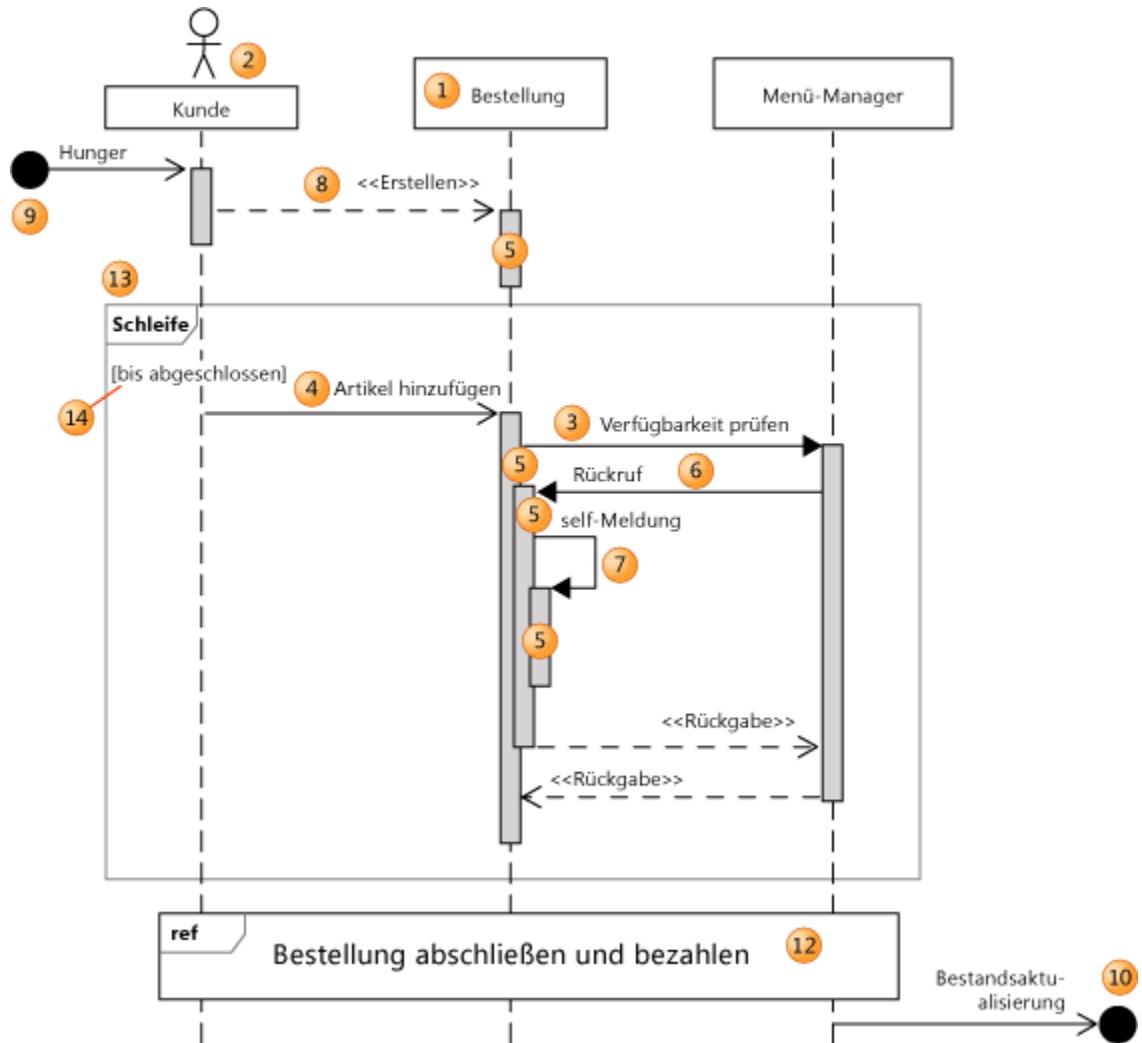


Abbildung 12: Beispiel Sequenzdiagramm

4.8.3 Elemente eines UML Sequenzdiagramm



Nr.	Element	Beschreibung
1	Lebenslinie	vertikale Linie, die die Sequenz von Ereignissen darstellt, die während einer Interaktion in einem Teilnehmer auftreten. Dieser Teilnehmer kann eine Instanz einer Klasse, einer Komponente oder eines Akteurs sein.
2	Akteur	Ein Teilnehmer ausserhalb des Systems, welches entwickelt wird.
3	Synchrone Nachricht	Absender wartet auf Antwort, bevor der Vorgang fortgesetzt wird. D.h. Aufruf, danach Rückgabe und erst dann läuft das Programm weiter.
4	Asynchrone Nachricht	Es wird nicht auf die Antwort gewartet, sondern der Aufruf abgesetzt (hier „Artikel hinzufügen“ und danach das Programm fortgesetzt. Häufig bei Asynchronen Server-Requests der Fall.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

5	Ausführungsvorkommen	Ein vertikales schattiertes Rechteck, das auf der Lebenslinie eines Teilnehmers erscheint und den Punkt darstellt, wenn der Teilnehmer eine Operation ausführt. Die Ausführung beginnt dort, wo der Teilnehmer eine Meldung empfängt. Wenn die initiiierende Meldung eine synchrone Meldung war, endet die Ausführung mit einem "Rückgabe"-Pfeil in Richtung des Absenders.
6	Rückrufmeldung	Antwort eines Methodenaufrufs.
7	Selbstmeldung	Aufruf einer eigenen Methode. Objekt ruft sich selber auf.
13	Kombinierte Fragmente	Fragmente innerhalb des Diagramms. Z.B. mehrfach ausgeführter Abschnitt (Schleife, z.B. for, while) oder if-Verzweigungen.

Tabelle 4: Elemente eines Sequenzdiagramm

5 RESTful-APIs - Basiswissen

5.1 Was ist eine RESTful-API?

REST steht für REpresentational State Transfer. Der als REST (oder auch ReST) bezeichnete Architekturansatz beschreibt, wie verteilte Systeme miteinander kommunizieren können. In diesem Sinne stellt eine REST API eine Alternative zu anderen Schnittstellen wie SOAP oder WSDL dar. REST selbst ist dabei allerdings weder Protokoll noch Standard.

Als „RESTful“ charakterisierte Implementierungen der Architektur bedienen sich allerdings standardisierter Verfahren, wie HTTP/S, URI, JSON oder XML.

Eine RESTful API ist also eine Schnittstelle (API), welche http(s)-Anfragen erhält (GET, POST, PUT, DELETE, etc.). Sie läuft auf einem Server. Der Client kann eine beliebige Applikation sein. Diese macht eine http(s) Abfrage an eine bestimmte URL auf dem Server. Die RESTful API nimmt die Anfrage entgegen und führt dann je nach URL eine unterschiedliche Aktion aus und gibt Daten an den Server zurück.

Beispiel:

1. Client macht http GET-Anfrage an <http://coolhomes.api.com/homes> (Hinweis: Seite funktioniert nicht)
2. RESTful API nimmt Anfrage entgegen. /homes gibt den Befehl, dass die RESTful-API alle Gebäude aus der Datenbank ermitteln. Limit 5 ist ein GET-Parameter, dass nur 5 Datensätze ermittelt werden sollen.
3. RESTful API ermittelt 5 Gebäude aus der DB.
4. RESTful API wandelt die Gebäude in ein JSON-Array (Mehrere JSON-Objekt) um und gibt dieses an den Client zurück.
5. Client stellt Ergebnisse im GUI dar.

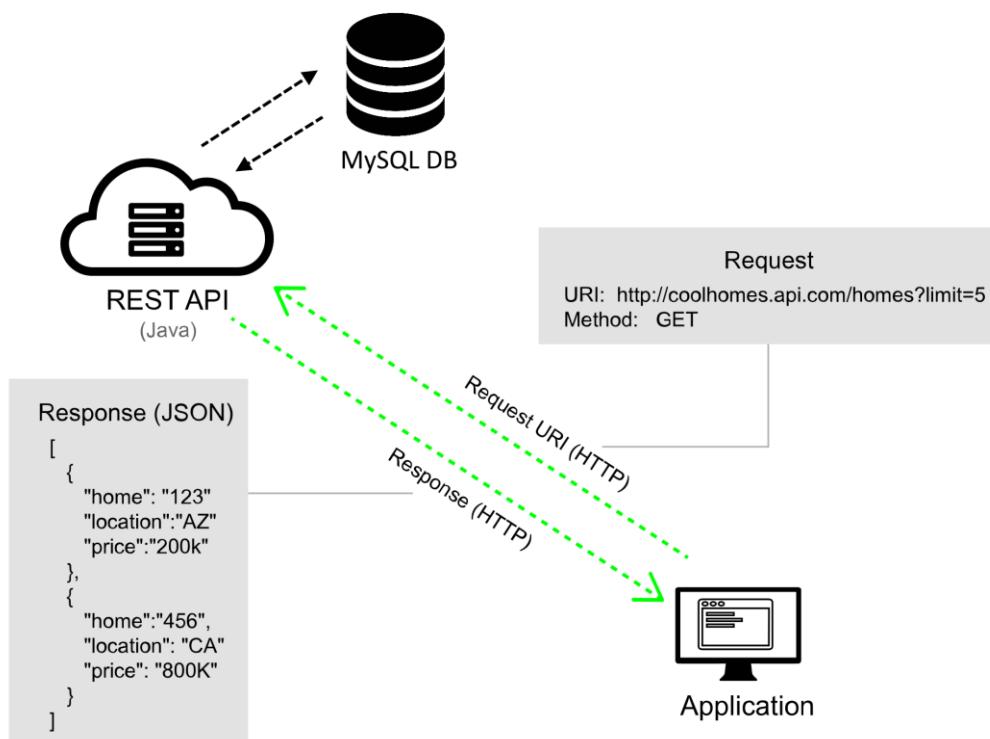


Abbildung 13: RESTful-API Request Beispie

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

Die RESTful-API ist eine Schnittstelle, welche einen Befehl im Rahmen einer http-URL erhält und Daten in einer standardisierte Form wie JSON zurückgibt. Daten vom Client an die API werden entweder in der URL (GET) oder auch als JSOIN (POST, PUT) übertragen.

Die Anzahl der Client ist unlimitiert. D.h. dieselbe RESTful-API kann z.B. sowohl von einer Webseite, einer Mobile App, einer Tablet App, einem Fernseher, einer Desktop App oder sonst einem Programm benutzt werden.

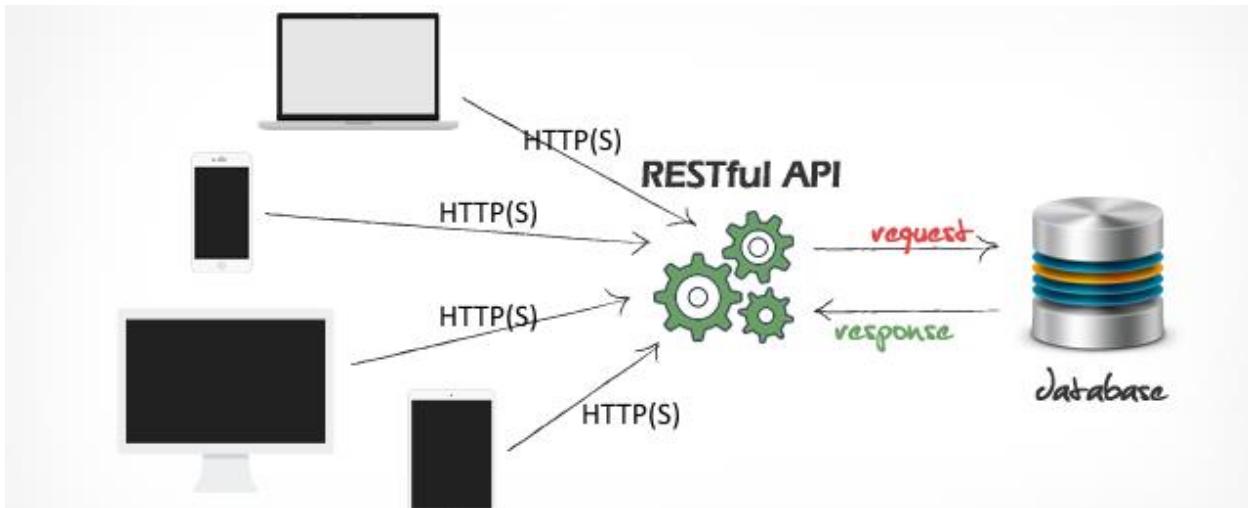


Abbildung 14: RESTful-API Beispiel

5.2 Architekturprinzipien von REST

- **Client-Server-Modell:**
REST verlangt ein Client-Server-Modell, will also das Nutzerinterface von der Datenhaltung getrennt sehen. Damit sollen sich Clients einerseits leichter auf verschiedenen Plattformen portieren lassen; vereinfachte Serverkomponenten sollen andererseits besonders gut skalieren.
- **Zustandslosigkeit:** Client und Server müssen zustandslos („stateless“) miteinander kommunizieren. Das bedeutet: Jede Anfrage eines Clients beinhaltet alle Informationen, die ein Server benötigt; Server selbst können auf keinen gespeicherten Kontext zurückgreifen. Dieses Constraint verbessert damit Visibilität, Zuverlässigkeit und Skalierbarkeit. Hierfür nimmt REST jedoch Nachteile bei der Netzwerkperformanz in Kauf; überdies verlieren Server die Kontrolle über ein konsistentes Verhalten der Client-App.
- **Caching:**
Um die Netzwerkeffizienz zu verbessern, können Clients vom Server gesendete Antworten auch speichern und bei gleichartigen Requests später erneut verwenden. Die Informationen müssen dem entsprechend als „cacheable“ oder „non-cacheable“ gekennzeichnet werden. Die Vorteile responsiverer Anwendungen mit höherer Effizienz und Skalierbarkeit werden dabei mit dem Risiko erkauft, dass Clients auf veraltete Daten aus dem Cache zurückgreifen.
- **Einheitliche Schnittstelle:** Die Komponenten REST-konformer Services nutzen eine einheitliche, allgemeine und vom implementierten Dienst entkoppelte Schnittstelle. Ziel des Ganzen sind eine vereinfachte Architektur und eine erhöhte Visibilität von Interaktionen. Dafür nimmt man eine schlechtere Effizienz in Kauf, wenn Informationen in ein standardisiertes Format gebracht – und nicht für die Bedürfnisse spezieller Anwendungen angepasst – werden.
- **Layered System:** REST setzt auf mehrschichtige, hierarchische Systeme („Layered System“) – jede Komponente kann ausschließlich jeweils direkt angrenzende Schichten sehen. Somit lassen sich beispielsweise Legacy-Anwendungen kapseln. Als Load Balancer agierende Vermittler („Intermediaries“) können überdies die Skalierbarkeit verbessern. Als Nachteile dieses Constraints gelten ein zusätzlicher Overhead und erhöhte Latenzen.

5.3 http Request Methoden – kurze Übersicht

- **GET:** GET wird zum Ermitteln von Daten verwendet. Die Parameter werden in der URL übergeben.
- **POST / PUT:** Wird für das Hinzufügen und Ändern von Daten verwendet. Parameter werden nicht in URL, sondern mit JSON übergeben.

POST ist besser geeignet wenn es um das Senden von Daten (z.B. Formulardaten) geht.
PUT ist besser geeignet für das Hochladen von Dokumenten oder Bilder auf den Server.

- **DELETE:** Zum Löschen einer Ressource (z.B. Dokument, Bild). Zum Löschen von Daten kann ebenfalls POST verwendet werden.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

5.4 JSON

JSON ist ein Datenformat. Es wird verwendet für den Datenaustausch zwischen Anwendungen. Bei RESTful APIs wird Daten zwischen Client und API meistens im JSON-Format ausgetauscht.

Bei SOAP-Webservices wurde XML verwendet für den Datenaustausch. Im Gegensatz zu XML ist JSON viel kompakter und leichter. Es beinhaltet mehr oder weniger nur die benötigten Daten und nicht noch viele andere Infos (Tags, etc.) wie bei XML.

Nachfolgend ein JSON einer Person. Eine Person verfügt über:

- Vorname
- Nachname
- Strasse + Nr.
- Wohnort
 - o PLZ
 - o Ort

```
{  
    "Vorname": "Joel",  
    "Nachname": "Holzer",  
    "Strasse": "Teststrasse 10  
    "Wohnort":  
    {  
        "PLZ": "2560",  
        "Ort": "Nidau"  
    }  
}
```

6 Erstes Projekt – Hello RESTful-API

Sie entwickeln mit Java und Spring Boot Ihre erste RESTful-API. Diese verfügt über eine GET-Methode, welcher der Name übergeben werden kann und als Antwort dann „Hello, Name“ zurückgibt.



Abbildung 15: Hello RESTful-API Postman-Request

Das Beispiel basiert auf folgendem Link: <https://spring.io/guides/gs/rest-service/>

6.1 Projekt erstellen

1. Spring Tool Suite starten
2. Klicken Sie auf das Menü „File → New → Spring Starter Project“. Machen Sie im Fenster folgende Angaben:

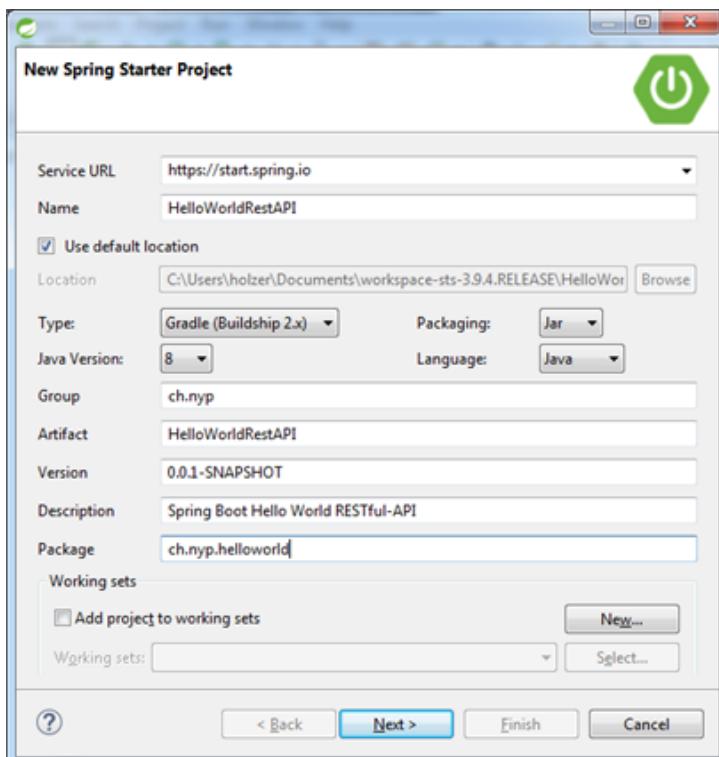


Abbildung 16: Neues Spring Projekt erstellen – Schritt 1

Klicken Sie auf „Next“.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

3. Klicken Sie die anderen Ansichten jeweils ohne Auswahl durch (immer „Next“) und im letzten Fenster „Finish“.
4. Das erste Spring Projekt wurde nun erstellt. Nun müssen noch die benötigten Klassen und Libraries hinzugefügt werden.

6.2 Dependencies hinzufügen

Damit das erstellte Projekt eine RESTful-API wird, respektive die benötigten Klassen genutzt werden können, müssen noch Dependencies (Libraries) hinzugefügt werden. Dies passiert im File „**build.gradle**“.

1. Im dependencies-Block muss folgende Dependency hinzugefügt werden:
`compile('org.springframework.boot:spring-boot-starter-web')`
2. Projekt neu refreshen. Dazu **Rechtsklick auf build.gradle → Gradle → Refresh Gradle Project**

Dies fügt die Libraries zur Entwicklung von Web-Applikationen, wie RESTful-APIs, hinzu. Auch wird ein Webserver-Container (Tomcat) hinzugefügt.

Jedes Spring Boot Projekt hat einen integrierten Webserver-Container. Wird die Applikation gestartet, so wird der integrierte Webserver-Container gestartet und die Applikation läuft dann auf diesem Webserver.

6.3 Klassen erstellen

6.3.1 Controller

Der Controller ist die Klasse, in welchem die Funktionen (REST-Requests), welche die API anbietet, implementiert werden. D.h. bei jedem REST-Request wird die entsprechende Methode im Controller aufgerufen. Erstellen Sie die Klasse „**GreetingController**“ mit folgendem Inhalt.

```
@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) {
        Greeting greeting = new Greeting(counter.incrementAndGet(),
            String.format(template, name));
        return greeting;
    }
}
```

@RestController: Definiert, dass diese Methoden REST-Funktionen beinhaltet.

@RequestMapping: URL-Zusatz, unter welchem die Methode vom Client aufgerufen werden kann. Hier <http://localhost:8080/greeting>

@RequestParam: Definiert, dass der Methodenparameter „name“ als GET-Parameter in der URL übergeben wird.

Beispiel: <http://localhost:8080/greeting?name=Joel>

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

Die Methode gibt ein Greeting-Objekt zurück an den Client. Genauer gesagt wandelt Spring Boot dieses Greeting-Objekt automatisch in JSON um und gibt dieses dann zurück.

6.3.2 Model

Erstellen Sie das Greeting-Model, wessen Objekt als JSON an den Client zurückgegeben wird.

```
public class Greeting {
    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() {
        return id;
    }

    public String getContent() {
        return content;
    }
}
```

Das JSON des Objekts, welches Spring Boot automatisch erstellt, ist wie folgt:

```
{
  "id": 1,
  "content": "Hello, Joel Holzer!
}
```

6.4 Projektstruktur

Nachfolgend die wichtigsten Verzeichnisse und Dateien des erstellten Projekts:

Verzeichnis/Datei	Sinn und Zweck
src/main/java	In diesem Verzeichnis liegen die Java-Dateien, d.h. die Logik der Applikation
Greeting.java	Model-Klasse für eine Begrüssung.
GreetingController.java	Der Controller ist die Klasse, in welchem die Funktionen (REST-Requests), welche die API anbietet, implementiert werden. D.h. bei jedem REST-Request wird die entsprechende Methode im Controller aufgerufen.
HelloWorldRestApi-Application	Main-Datei. Wird beim Start der Applikation aufgerufen. Wird generiert bei Projekterstellung. Keine Änderungen nötig.
src/main/resources	Alle anderen Dateien als Java, z.B. Konfigurationsfiles, Bilder, etc.

Modul 223: Multi-User-Applikationen objektorientiert realisieren

application.properties	Spring Konfigurationsdatei. Jegliche Spring-Konfigurationen. Werden wir später verwenden für die Konfiguration des DB-Zugriff (Zugangsdaten, DB-Name).
src/test/java	Unit-Testklassen.
build.gradle	Zentrale Gradle-Datei für die Erstellung (Build) des Java-Projekts. In dieser Datei müssen die verwendeten Plugins, die Dependencies (Libraries), die Versionsnummer der Applikation und weitere Angaben gemacht werden.

6.5 Applikation starten & testen

6.5.1 Applikation starten

Starten Sie Ihre RESTful-API direkt in der Spring Tool Suite. Gehen Sie dazu wie folgt vor:

Rechtsklick auf Projekt → run as → Spring Boot App

6.5.2 Applikation mit Postman testen

Postman ist ein Tool, mit welchem APIs getestet werden können. Mit Postman können Sie einen http-Request (GET, POST, etc.) an eine API senden und dann das erhaltene Resultat einsehen.

Senden Sie nun mit Postman einen http-GET-Request an ihre RESTful-API. Geben Sie Ihren Namen als Parameter mit. Analysieren Sie das erhaltene Resultat.

Nachfolgender Screenshot zeigt die durchgeführte Abfrage mit Postman.

The screenshot shows the Postman interface with the following details:

- Method: GET
- URL: <http://localhost:8080/greeting?name=joel%20Holzer>
- Params: None
- Send and Save buttons are visible.
- Authorization tab is selected.
- Headers, Body, Pre-request Script, and Tests tabs are present.
- Body tab is selected, showing the response content.
- Cookies and Code tabs are also present.
- Response status: 200 OK
- Time: 144 ms
- Size: 170 B
- Response body (Pretty JSON):

```
1  {  
2      "id": 1,  
3      "content": "Hello, Joel Holzer!"  
4 }
```

7 Datenbank-Anbindung mit JPA & MySQL

7.1 Was ist OR-Mapping?

Objektrelationales Mapping ist eine Technik aus der Softwareentwicklung, um Objekte aus einer objektorientierten Anwendung in einer relationalen Datenbank abzulegen.

Vereinfacht gesagt: Java-Objekte in einer relationalen Datenbank speichern. Um Objektrelationales Mapping einzusetzen, kommen Persistence Libraries zum Zug.

Die meisten Persistence Libraries bilden Klassen auf Tabellen ab. D.h. für jede Java-Klasse, wessen Objekte in der Datenbank gespeichert werden sollen, existiert eine Tabelle in der Datenbank. Die Java-Klassen, welche in der DB gespeichert werden sollen, müssen speziell gekennzeichnet werden. Zwischentabellen existieren meistens nicht als Java-Klassen, sondern können direkt aus der n:m-Beziehung zweier Klassen generiert werden.

Die wohl bekannteste Persistence Libraries für Java sind JPA und andere auf JPA basierte Implementationen (Hibernate, Spring Data).

Wir werden uns im nachfolgenden Beispiel auf die Umsetzung mit JPA & Spring Data konzentrieren.

O/R Mapping

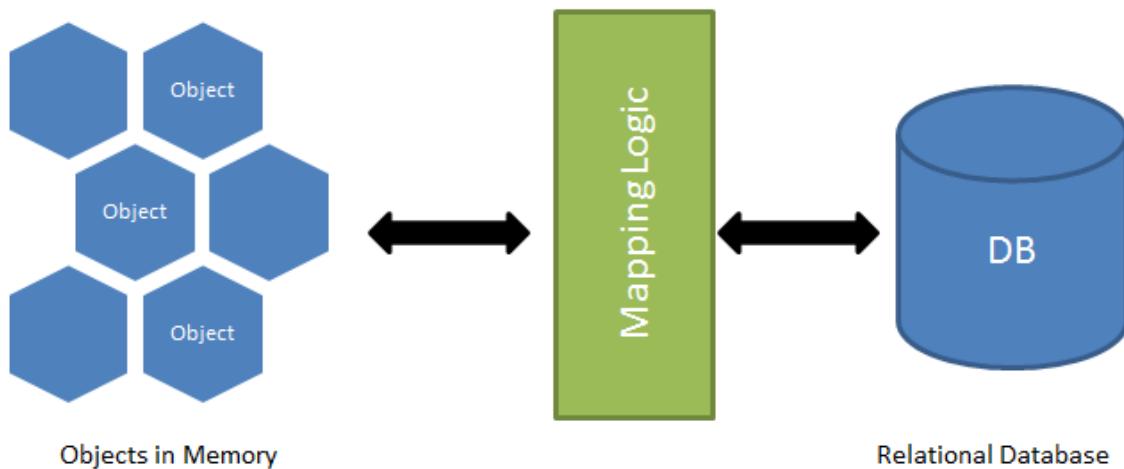


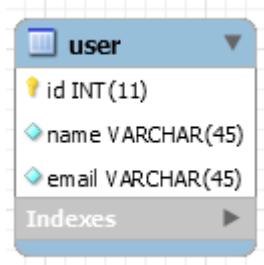
Abbildung 17: Objektrelationales Mapping

7.2 MySQL RESTful-API mit grundlegenden DB-Operationen

Sie entwickeln mit Java und Spring Boot eine RESTful-API mit folgenden Funktionen:

- User hinzufügen
- Alle User ermitteln

Die Daten werden in eine MySQL-DB geschrieben und daraus gelesen. Folgende Entitäten sind vorhanden:



Das Beispiel basiert auf folgendem Link:

<https://spring.io/guides/gs/accessing-data-mysql/>

7.2.1 Datenbank erstellen

Erstellen Sie die Datenbank „`springboot_firstmysql`“ mit der Tabelle „`user`“.

7.2.2 Projekt erstellen

Kapitel 6.1 ausführen mit anderen Angaben.

7.2.3 Dependencies hinzufügen

Damit das erstellte Projekt eine RESTful-API wird und die DB-Anbindung mit JPA von stattet geht, müssen noch Dependencies (Libraries) hinzugefügt werden. Dies passiert im File „`build.gradle`“.

1. Im dependencies-Block müssen folgende Dependencies hinzugefügt werden:


```
compile('org.springframework.boot:spring-boot-starter-web')
compile('org.springframework.boot:spring-boot-starter-data-jpa')
compile('mysql:mysql-connector-java')
```
2. Projekt neu refreshen. Dazu **Rechtsklick auf build.gradle → Gradle → Refresh Gradle Project**

Dies fügt die folgenden 3 Libraries hinzu:

- **Spring Boot Starter Web:** Library zur Entwicklung von Web-Applikationen, wie RESTful APIs, hinzu. Auch wird ein Webserver-Container (Tomcat)
- **Spring Boot Starter Data JPA:** JPA für den Zugriff auf die Datenbank (OR Mapping)
- **MySQL Connector Java:** Ermöglicht Zugriff auf MySQL-Datenbanken.

7.2.4 Klassen erstellen

7.2.4.1 Model-Klasse

Erstellen Sie die Java-Klassen (Models), wessen Objekte in der Datenbank gespeichert werden sollen. Für jede gewünschte DB-Tabelle (ausser Zwischentabellen) muss ein gleichnamiges Model erstellt werden. Erstellen Sie also eine Model-Klasse für „User“ mit den Attributen id, name und email.

Damit das Model zu einer Datenbanktabelle gemappt wird, muss dies nun mit Annotations gekennzeichnet werden. Nachfolgend sehen Sie den Code des User-Models (Klasse user.java):

```

@Entity
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String email;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
  
```

@Entity markiert die Klasse als Datenbanktabelle. Wird nichts anderes angegeben, so heisst die Tabelle in der DB gleich wie das Model, d.h. in diesem Fall „user“.

Heisst die Tabelle in der DB anders, kann dies mit **@Entity(name = „xyz“)** angegeben werden.

@Id markiert die Instanzvariable als ID-Feld in der Datenbank.

@GeneratedValue(strategy=GenerationType.IDENTITY) besagt, dass dies ein Autoincrement-Feld ist.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

Wenn bei einer Instanzvariable keine Annotation angegeben wird, so entspricht dies automatisch einem Attribut mit demselben Namen in der DB. Nachfolgend ein paar andere wichtige Annotations:

@Column(name="e_mail") Feld heisst in DB e_email und nicht email wie die Instanzvariable.

@Transient Feld ist in DB nicht vorhanden, nur in Model.

Nachfolgender Link gibt einen Überblick über alle JPA-Annotations:

<https://www.objectdb.com/api/java/jpa/annotations>

7.2.4.2 Repository-Klasse

Das Repository wird benötigt um auf die DB zuzugreifen. Solange Standard-Operationen, z.B. Select All oder Insert All Items ausreichend sind, können die Funktionen von JPA, Spring Data oder Hibernate genutzt werden.

Erstellen Sie dazu ein Interface, welches von einer der folgenden Klassen erbt:

- **CrudRepository**: Create, Read, Update, Delete für 1 Datensatz oder alle Datensätze.
- **JpaRepository**: Create, Read, Update, Delete für 1 Datensatz oder alle Datensätze. Zusätzliche Methoden. JpaRepository hat alle Funktionen von CrudRepository + Zusätzliche.
- **PagingAndSortingRepository**: Alle Methoden von CrudRepository + zusätzliche Methoden zum Sortieren und Blättern (Pagination) von Daten.

7.2.4.3

7.2.4.4 Nachfolgend der Code für das Repository im Beispielprojekt:

```
public interface BookRepository extends JpaRepository<Book, Long> {  
    7.2.4.5 }
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

7.2.4.6 Controller-Klasse

Der Controller ist wie beim ersten Beispielprojekt die Klasse, in welchem die Funktionen für die REST-Requests implementiert werden.

Hier sollen Methoden zum Hinzufügen eines neuen Users und zum Ermitteln aller vorhandenen User erstellt werden.

Nachfolgend der Code:

```
@Controller
@RequestMapping(path="/databaseFirstDemo")
public class MainController {

    @Autowired
    private UserRepository userRepository;

    @PostMapping(path="user/add")
    public @ResponseBody String addNewUser(@RequestBody User user) {
        userRepository.save(user);
        return "User saved";
    }

    @GetMapping(path="user/all")
    public @ResponseBody List<User> getAllUsers() {
        List<User> usersFromDB = userRepository.findAll();
        return usersFromDB;
    }
}
```

@Autowired:

Wird für das Dependency Injection (siehe Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**) genutzt. Anstelle dass jede Klasse, wo eine Instanz von UserRepository benötigt wird, diese mit „new“ selber erstellt, werden die Instanzen vom Spring Framework verwaltet. Mit @Autowired wird das Instanzmanagement an das Framework übergeben. Spring erstellt dann automatisch eine Instanz, sofern noch keine existiert und verwendet dann jeweils die bereits vorhandene Instanz (Singleton-Pattern).

userRepository.save(user):

Speichert den übergebenen User (User-Objekt) in die Datenbank (Insert-Statement).

userRepository.findAll():

Ermittelt alle User aus der Datenbank und gibt diese als List von User-Objekten zurück.

7.2.4.7 application.properties

Zu guter Letzt muss in der Datei „src/main/resources/application.properties“ noch der Zugriff auf die DB konfiguriert werden, d.h. die Zugangsdaten angegeben werden:

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/springboot_mysql
spring.datasource.username= springboot_firstmysql
spring.datasource.password=123456Aa!
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren****7.2.5 Applikation starten & testen****7.2.5.1 Applikation starten**

Starten Sie Ihre RESTful-API direkt in der Spring Tool Suite. Gehen Sie dazu wie folgt vor:

Rechtsklick auf Projekt → run as → Spring Boot App

7.2.5.2 Applikation mit Postman testen

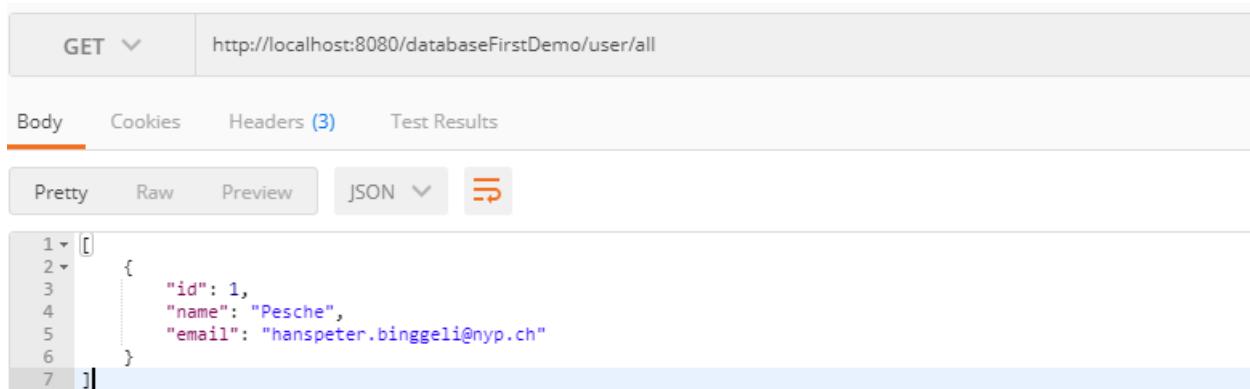
Nachfolgender Screenshot zeigt das Hinzufügen eines neuen Users.



A screenshot of the Postman application interface. The top bar shows 'POST' and the URL 'http://localhost:8080/databaseFirstDemo/user/add'. Below the URL, there are tabs for 'Authorization', 'Headers (1)', 'Body' (which is selected), 'Pre-request Script', and 'Tests'. Under 'Body', the 'raw' tab is selected, and the content type is set to 'JSON (application/json)'. The JSON payload is as follows:

```
1 {  
2   "name": "Pesche",  
3   "email": "hanspeter.binggeli@nyp.ch"  
4 }
```

Nachfolgender Screenshot zeigt das Ermitteln aller bestehenden User:



A screenshot of the Postman application interface. The top bar shows 'GET' and the URL 'http://localhost:8080/databaseFirstDemo/user/all'. Below the URL, there are tabs for 'Body' (selected), 'Cookies', 'Headers (3)', and 'Test Results'. Under 'Body', the 'Pretty' tab is selected, and the response is displayed as JSON:

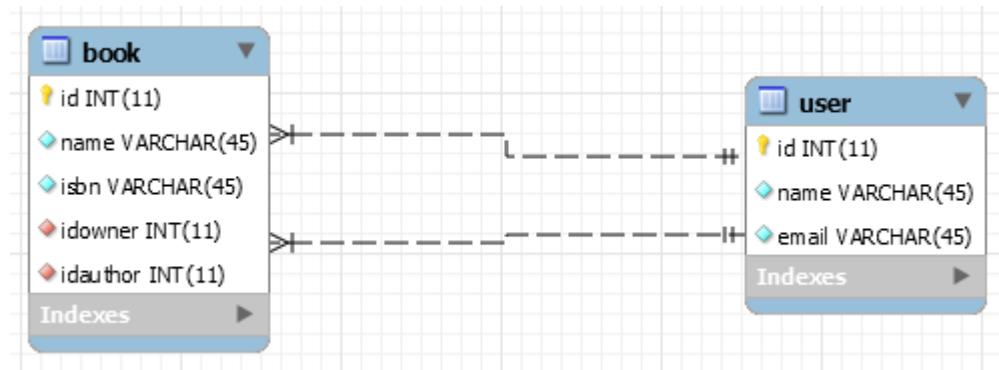
```
1 [  
2   {  
3     "id": 1,  
4     "name": "Pesche",  
5     "email": "hanspeter.binggeli@nyp.ch"  
6   }  
7 ]
```

7.3 MySQL RESTful-API mit Relationen

Sie entwickeln mit Java und Spring Boot eine RESTful-API mit folgenden Funktionen:

- User hinzufügen
- Buch hinzufügen
- Alle User ermitteln
- Alle Bücher ermitteln

Die Daten werden in eine MySQL-DB geschrieben und daraus gelesen. Folgende Entitäten sind vorhanden:



- idowner: Fremdschlüssel auf User-Tabelle
- idauthor: Fremdschlüssel auf User-Tabelle

Das Beispiel basiert auf einer Erweiterung des Projekts in Kapitel 7.2. In diesem Kapitel wird das bestehende Projekt erweitert. Daher wird nur auf die Erweiterungen eingegangen.

In diesem Projekt werden nicht direkt die DB-Models an den Client zurückgegeben, sondern mit Data Transfer Objects (DTO) gearbeitet.

Weitere Infos zu DTOs: Kapitel 0

Weitere Links:

<http://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application>

7.3.1 Dependencies hinzufügen

Zusätzlich zum Projekt aus Kapitel 7.2 folgende Dependency hinzufügen:

```
compile('org.modelmapper:modelmapper:2.0.0')
```

ModelMapper ist eine Library, welche die Transformation von JPA Entities in DTOs und umgekehrt vereinfacht.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

7.3.2 Klassen erstellen

7.3.2.1 Model-Klassen

Erstellen Sie die Java-Klassen (Models), wessen Objekte in der Datenbank gespeichert werden sollen. Diese Klassen werden JPA Entities genannt. Für jede gewünschte DB-Tabelle (ausser Zwischentabellen) muss ein gleichnamiges Model erstellt werden. Erstellen Sie also eine Model-Klasse für „User“ und „Book“.

- **User-Model:**

Entspricht 1:1 dem Model aus Kapitel 7.2.4.

- **Book-Model:**

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String isbn;

    @ManyToOne
    @JoinColumn(name = "idowner", nullable = false)
    private User owner;

    @ManyToOne
    @JoinColumn(name = "idauthor", nullable = false)
    private User author;

    //Alle Getter und Setter
}
```

@ManyToOne: Charakterisiert das Feld als Fremdschlüssel mit einer m:1 Beziehung. D.h. Buch hat 1 Besitzer, Besitzer kann mehrere Bücher haben.

@JoinColumn: In einer MySQL-DB können keine Objekte gespeichert werden. Im Model hingegen schon. Die Spalte in der MySQL-DB heisst „**idowner**“. Hiermit wird mitgeteilt, dass das Objekt „**owner**“ zur Spalte „**idowner**“ gehört.

JPA macht automatisch ein JOIN und füllt bei einem Select das owner-Objekt mit dem **ganzen User** und nicht nur mit der ID.

Weitere Infos zu JPA-Annotations im Hinblick auf Beziehungen finden Sie unter
<https://www.objectdb.com/api/java/jpa/annotations/relationship>

7.3.2.2 Repository-Klassen

Pro JPA Entity (Model-Klasse) ein Repository erstellen (BookRepository, UserRepository). Analog Kapitel 7.2.4.

7.3.2.3 Data Transfer Objects (DTOs)

Wie bereits erwähnt werden in diesem Projekt DTOs eingesetzt. D.h. für jede JPA-Entity muss ein DTO erstellt werden.

Weitere Infos zu DTO, siehe Kapitel 0.

- **UserDto:**

Entspricht der JPA-Entity, einfach ohne Annotations.

```
public class UserDto {
    private Long id;
    private String name;
    private String email;
    //Getter und Setter
}
```

- **BookDto:**

Weist gegenüber der JPA-Entity einige Unterschiede auf:

- o **idowner** wird auch abgelegt, jedoch nur bei POST. Möchte nämlich ein Buch-Objekt erstellt werden und der Owner gesetzt werden, geht dies einfacher über die ID als wenn das ganze Objekt gesetzt werden muss. Beim GET reicht es jedoch, wenn das ganze User-Objekt als Owner zurückgegeben wird.
- o **idauthor**: Analog idowner.
- o **owner & author**: Sind vom Typ „UserDto“ und nicht wie in der Entity vom Typ „User“.

```
public class BookDto {
    private Long id;
    private String name;
    private String isbn;

    @JsonProperty(access = Access.WRITE_ONLY) //Feld wird im JSON nur
beim Schreiben (POST) akzeptiert, d.h. im GET nicht zurückgegeben.
    private Long idowner;

    private UserDto owner;

    @JsonProperty(access = Access.WRITE_ONLY) //Feld wird im JSON nur
beim Schreiben (POST) akzeptiert, d.h. im GET nicht zurückgegeben.
    private Long idauthor;

    private UserDto author;
    //TODO Getter und Setter
}
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

7.3.2.4 DTO Converters

Für die Umwandlung von DTOs in JPA-Entities und umgekehrt wird die Library „ModelMapper“ verwendet. Es ist jedoch dennoch notwendig, Converter-Klassen zu schreiben. Für jedes DTO soll eine Converter-Klasse erstellt werden:

- UserDtoConverter: Wandelt die User-Entity in das UserDto um.
- BookDtoConverter. Wandelt die Book-Entity in das BookDto um.

```

public class UserDtoConverter {
    @Autowired
    private ModelMapper modelMapper;

    public UserDto convertToDto(User user) {
        UserDto userDto = modelMapper.map(user, UserDto.class);
        return userDto;
    }

    public User convertToEntity(UserDto userDto) {
        User user = modelMapper.map(userDto, User.class);
        return user;
    }
}

public class BookDtoConverter {
    @Autowired
    private ModelMapper modelMapper;

    public BookDto convertToDto(Book book) {
        BookDto bookDto = modelMapper.map(book, BookDto.class);
        return bookDto;
    }

    public Book convertToEntity(BookDto bookDto) {
        Book book = modelMapper.map(bookDto, Book.class);

        User author = new User();
        author.setId(bookDto.getIdauthor());
        book.setAuthor(author);

        User owner = new User();
        owner.setId(bookDto.getIdowner());
        book.setOwner(owner);

        return book;
    }
}

```

In der Buch-Klasse ist gut zu sehen, wie aus der in der DTO vorhandenen "idowner" und "idauthor" ein User-Objekt erstellt wird, da die JPA-Entity ein User-Objekt voraussetzt, um die Daten in die DB zu schreiben.

@Autowired: ModelMapper-Instanz wird von Spring erstellt (DI/IOC)

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

7.3.2.5 SpringConfig

Damit der ModelMapper, BookDtoConverter und UserDtoConverter von Spring per DI/IOC erstellt werden, muss eine Spring-Konfigurationsdatei erstellt werden. Diese kann entweder als Java-Klasse oder XML-Datei erstellt werden. Dieses Beispiel basiert auf einer Java-Klasse.

Mit der Annotation **@Configuration** wird die Java-Klasse als Spring-Konfigurationsdatei festgelegt.

Mit **@Bean** wird überhaupt erst das DI/IOC über die Annotation ermöglicht. Sobald Spring eine mit @Bean gekennzeichnete Methode findet, führt er diese aus und registriert den Rückgabewert in der BeanFactory. Der Bean-Name ist derselbe wie der Methodennamen. Dies ermöglicht dann erst die Instanzerzeugung über @Autowire per DI/IOC.

```
@Configuration
public class SpringConfig {

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }

    @Bean
    public UserDtoConverter userDtoConverter() {
        return new UserDtoConverter();
    }

    @Bean
    public BookDtoConverter bookDtoConverter() {
        return new BookDtoConverter();
    }
}
```

7.3.2.6 Controller-Klasse

7.3.2.7 Aus Komplexitätsgründen wird hier nicht die ganze Controller-Klasse erläutert, sondern nur Codeausschnitte:

- **Neuer User hinzufügen:**

```
@PostMapping(path="user/add")
public @ResponseBody String addNewUser(@RequestBody UserDto userDto) {
    User dbUser = userDtoConverter.convertToEntity(userDto);
    userRepository.save(dbUser);
    return "User saved";
```

7.3.2.8 }

Gut zu sehen ist, wie das erhaltene DTO in die JPA-Entity umgewandelt und diese dann dem Repository übergeben wird.

Modul 223: Multi-User-Applikationen objektorientiert realisieren

- **User ermitteln:**

```
@GetMapping(path="user/all")
public @ResponseBody List<UserDto> getAllUsers() {
    List<User> usersFromDB = userRepository.findAll();
    List<UserDto> users = usersFromDB.stream().map(user ->
        userDtoConverter.convertToDto(user)).collect(Collectors.toList());

    return users;
}
```

In diesem Beispiel wird zuerst eine Liste von User-Entities ermittelt. Danach wird die ganze Liste über den Converter in eine Liste von User-DTOs umgewandelt. Dies geht über einen Stream und Lambda Expressions.

Die Funktionen für das Hinzufügen eines Buches und das Ermitteln aller Bücher sind ähnlich wie die 2 erläuterten Methoden.

7.3.2.9 application.properties

Siehe Kapitel 7.2.4

7.3.3 Applikation starten & testen

7.3.3.1 Applikation starten

Starten Sie Ihre RESTful-API direkt in der Spring Tool Suite. Gehen Sie dazu wie folgt vor:

Rechtsklick auf Projekt → run as → Spring Boot App

7.3.3.2 Applikation mit Postman testen

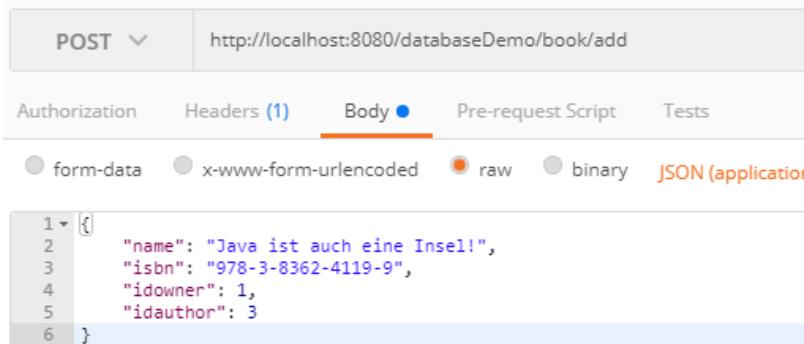
- **Hinzufügen von User:**

Analog Kapitel 7.2.5

- **Ermitteln von User:**

Analog Kapitel 7.2.5

- **Hinzufügen von Büchern:**



The screenshot shows the Postman interface with a POST request to `http://localhost:8080/databaseDemo/book/add`. The **Body** tab is selected, showing a JSON payload:

```

1 [{}]
2   "name": "Java ist auch eine Insel!",
3   "isbn": "978-3-8362-4119-9",
4   "idowner": 1,
5   "idauthor": 3
6 }
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

- Ermitteln aller Bücher:

GET <http://localhost:8080/databaseDemo/book/all>

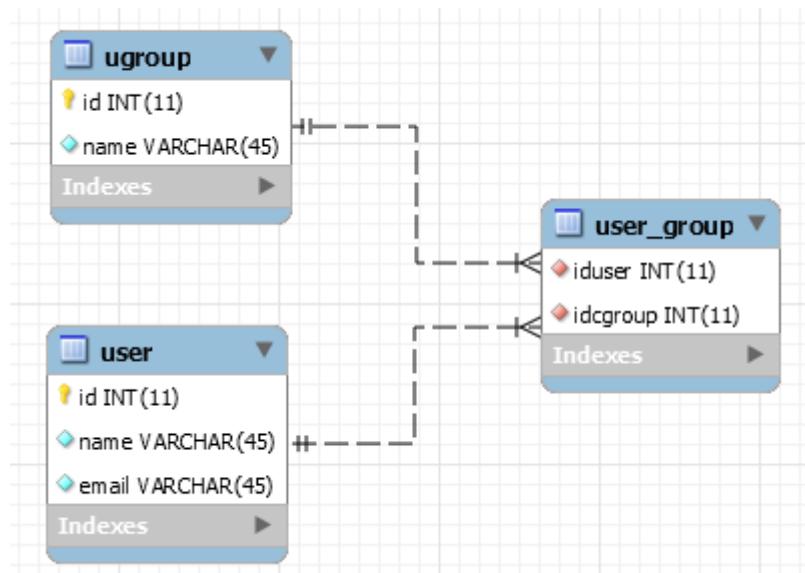
Body Cookies Headers (3) Test Results

Pretty Raw Preview JSON ↗

```
1 [ ]  
2 {  
3     "id": 2,  
4     "name": "Java ist auch eine Insel!",  
5     "isbn": "978-3-8362-4119-9",  
6     "owner": {  
7         "id": 1,  
8         "name": "Pesche",  
9         "email": "hanspeter.binggeli@nyp.ch"  
10    },  
11    "author": {  
12        "id": 3,  
13        "name": "Pesche3",  
14        "email": "hanspeter.binggeli@nyp.ch"  
15    }  
16 },  
17 {  
18     "id": 3,  
19     "name": "Java ist auch eine Insel!",  
20     "isbn": "978-3-8362-4119-9",  
21     "owner": {  
22         "id": 1,  
23         "name": "Pesche",  
24         "email": "hanspeter.binggeli@nyp.ch"  
25    },  
26     "author": {  
27        "id": 2,  
28        "name": "Pesche2",  
29        "email": "hanspeter.binggeli@nyp.ch"  
30    }  
31 },  
32 {  
33     "id": 4,  
34     "name": "Java ist auch eine Insel!",  
35     "isbn": "978-3-8362-4119-9",  
36     "owner": {  
37         "id": 1,  
38         "name": "Pesche",  
39         "email": "hanspeter.binggeli@nyp.ch"  
40    },  
41     "author": {  
42        "id": 3,  
43        "name": "Pesche3",  
44        "email": "hanspeter.binggeli@nyp.ch"  
45    }  
46 },  
47 ]
```

7.4 MySQL RESTful-API mit n:m Relationen

Sie entwickeln mit Java und Spring Boot eine RESTful-API, welche Daten aus einer Datenbank mit Zwischentabelle liest und schreibt. Folgende Entitäten sind vorhanden:



Entwickeln Sie dazu die RESTful-API mit folgenden Funktionen:

- Gruppe hinzufügen
- User hinzufügen inkl. Zuteilung zu Gruppe
- Gruppen auslesen inkl. zugeordnete User
- User auslesen, inkl. zugeordnete Gruppen

Dieses Beispiel basiert auf folgendem Link:

<http://www.baeldung.com/spring-data-rest-relationships>

7.4.1 Dependencies hinzufügen

Dieselben Dependencies wie in Kapitel 7.2.3

7.4.2 Klassen erstellen

7.4.2.1 Model-Klassen

Erstellen Sie die Java-Klassen (Models), wessen Objekte in der Datenbank gespeichert werden sollen. Diese Klassen werden JPA Entities genannt.

Wichtig: Für die Zwischentabelle wird kein eigenes Model erstellt. Es muss nur ein Model für Gruppe und User erstellt werden.

- **User-Model:**

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String email;

    @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.MERGE)
    @JoinTable(name = "user_group", joinColumns = @JoinColumn(name =
    "idcgroup", referencedColumnName = "id"),
    inverseJoinColumns = @JoinColumn(name = "iduser", referencedColumnName =
    "id"))
    private List<Group> groups;

    //Getter & Setter
}
```

- **Group-Model:**

```
@Entity(name = "ugroup")
public class Group {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "groups")
    private List<User> users;

    //Getter & Setter
}
```

@ManyToMany: Charakterisiert das Feld als M:n Beziehung. D.h. in der Liste werden beim User alle dem User zugeordneten Gruppen gespeichert und bei der Gruppe alle User.

@JoinTable: Mit JoinTable wird die Zwischentabelle angegeben. Denn über diese Zwischentabelle können z.B. die Gruppen dem User zugeordnet werden.

@JoinColumn: Angabe des Mapping von Zwischentabelle auf die eigene Tabelle, d.h. welches Attribut der Zwischentabelle entspricht welchem Attribut der eigenen Tabelle. So werden beide Attribute der Zwischentabelle gemappt. Das erste Attribut (idcgroup) wird auf die id der Gruppentabelle gemappt. Das zweite Attribut (iduser) wird auf die id der Usertabelle gemappt.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

mappedBy = "groups": Die Annotations für die m:n Beziehungen müssen nur in einem Model gemacht werden, und nicht in beiden an der Zwischentabelle beteiligten Models. Im zweiten Model, hier „Group“, wird dann einfach mit „**mappedBy**“ auf das Attribut (Instanzvariable) des anderen Models (dort wo JoinTable-Annotations) gemappt.

Weitere Infos zu JPA-Annotations im Hinblick auf Beziehungen finden Sie unter
<https://www.objectdb.com/api/java/jpa/annotations/relationship>

7.4.2.2 Repository-Klassen

Pro JPA Entity (Model-Klasse) ein Repository erstellen:

```
public interface GroupRepository extends JpaRepository<Group, Long> { }
```

```
public interface UserRepository extends JpaRepository<User, Long> { }
```

7.4.2.3 Data Transfer Objects (DTOs):

Für jede JPA-Entity muss wiederum ein DTO erstellt werden. Weitere Infos zu DTO, siehe Kapitel 0. Die DTOs entsprechen in diesem Beispiel 1:1 den Models (einfach ohne JPA-Annotations).

7.4.2.4 DTO Converters

Für die Umwandlung von DTOs zu JPA-Entites und umgekehrt wird wiederum „Modelmapper“ verwendet und pro DTO eine Converter-Klasse geschrieben. Die Converter-Klasse in diesem Beispiel sind jedoch um einiges grösser als bei Entities mit 1:n Beziehungen.

Das Problem bei m:n Beziehungen ist, dass es bei einem GET und der Umwandlung zu einem JSON zu einer InfiniteRecursion-Exception kommen kann, weil Gruppe und User beide miteinander in Beziehung stehen und sich unendlich selber aufrufen.

Nicht gewünschtes Verhalten:

```
[{
  "id": 1,
  "name": "Hanspeter Binggeli",
  "email": "hanspeter.binggeli@nyp.ch",
  "groups": [
    {
      "id": 1,
      "name": "Gruppe 1",
      "users": [
        {
          "id": 1,
          "name": "Adrian Krebs",
          "email": "adrian.krebs@nyp.ch"
          "groups": [
            {
              "id": 1,
              "name": "Gruppe 1",
              "users": "USW..."
            }
          ]
        }
      ]
    }
  ]
}
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

}
]]

Gewünschtes Verhalten:

Nach einer Iteration soll abgebrochen werden, damit wir folgendes JSON haben:

```
[{
  "id": 1,
  "name": "Hanspeter Binggeli",
  "email": "hanspeter.binggeli@nyp.ch",
  "groups": [
    {
      "id": 1,
      "name": "Gruppe 1",
      "users": null
    }
  ]
}]
```

D.h. wenn die User ermittelt werden, werden jeweils pro User gerade die Gruppen zurückgegeben. Dasselbe soll umgekehrt geschehen:

```
[{
  "id": 1,
  "name": "Gruppe 1",
  "users": [
    {
      "id": 2,
      "name": "Adrian Krebs",
      "email": "adrian.krebs@nyp.ch",
      "groups": null
    }
  ]
}]
```

Code der Converter:

In beiden Converter sollte es daher eine Methode geben, welche das User-DTO, respektive das Group-DTO für die Verschachtlung selber zusammenstellt und so nach einer Iteration abbricht

Nachfolgend der Beispielcode für den **UserDtoConverter**. Der GroupDtoConverter ist genau gleich aufgebaut, einfach für Group anstelle von User.

```
public class UserDtoConverter {
  @Autowired
  private ModelMapper modelMapper;

  public UserDto convertToDto(User user) {
    UserDto userDto = modelMapper.map(user, UserDto.class);
    return userDto;
  }

  public User convertToEntity(UserDto userDto) {
    User user = modelMapper.map(userDto, User.class);
    return user;
  }
}
```

Modul 223: Multi-User-Applikationen objektorientiert realisieren

```

public UserDto convertToDtoRead(User user) {
    UserDto userDto = new UserDto(user.getId(), user.getName(), user.getEmail());

    List<Group> userGroups = user.getGroups();
    List<GroupDto> userDtoGroups = new ArrayList<GroupDto>();
    for (Group group : userGroups) {
        GroupDto groupDto = new GroupDto(group.getId(), group.getName());
        userDtoGroups.add(groupDto);
    }
    userDto.setGroups(userDtoGroups);
    return userDto;
}
}
  
```

Die Methode „**convertToDtoRead**“ wandelt die übergebene User-Entity manuell in ein User-DTO um, indem die Instanzvariablen abgefüllt werden und dann für jede zugeordnete Gruppe ein GroupDTO erstellt und dem User-DTO zugeordnet wird. Bei den erstellten Group-DTOs werden bewusst keine User mehr angehängt, weil hier die Iteration abgebrochen werden soll.

7.4.2.5 SpringConfig

Damit der ModelMapper, UserDtoConverter und BookDtoConverter von Spring per DI/IOC erstellt werden, muss eine Spring-Konfigurationsdatei erstellt werden. Diese kann entweder als Java-Klasse oder XML-Datei erstellt werden. Siehe dazu Kapitel 7.3.2.

7.4.2.6 Controller-Klasse

In diesem Beispiel wird jeweils eine Controller-Klasse pro Model erstellt, d.h.

- UserController
- GroupController

Beide Controller sind gleich aufgebaut. Nachfolgend der Code des UserController.

Bei der GetAllUsers-Methode wird nun die vorhin erstellte DTO-Methode aufgerufen, damit eben keine InfiniteRecursion-Exception auftritt.

```

@Controller
@RequestMapping(path="/mnDatabaseDemo")
public class UserController {
    @Autowired
    private ModelMapper modelMapper;
    @Autowired
    private UserRepository userRepository;

    @GetMapping(path="/users/all")
    public @ResponseBody List<UserDto> getAllUsers() {
        List<User> users = userRepository.findAll();
        List<UserDto> userDtos = new ArrayList<UserDto>();
        for (User user : users) {
            userDtos.add(modelMapper.map(user, UserDto.class));
        }
        return userDtos;
    }

    @PostMapping(path="/users/add")
    public @ResponseBody String addNewUser(@RequestBody UserDto user) {
        User dbUser = modelMapper.map(user, User.class);
        userRepository.save(dbUser);
        return "User saved";
    }
}
  
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

```

public @ResponseBody Iterable<GroupDto> getAllGroups() {
    List<Group> groupsFromDB = groupRepository.findAll();

    List<GroupDto> groups = groupsFromDB.stream().map(group ->
        groupDtoConverter.convertToDtoRead(group)).collect(Collectors.toList());

    return groups;
}
}
  
```

7.4.2.7 application.properties

Siehe Kapitel 7.2.4

7.4.3 Applikation starten & testen

7.4.3.1 Applikation starten

Starten Sie Ihre RESTful-API direkt in der Spring Tool Suite. Gehen Sie dazu wie folgt vor:

Rechtsklick auf Projekt → run as → Spring Boot App

7.4.3.2 Applikation mit Postman testen

- Gruppe hinzufügen



POST http://localhost:8080/mnDatabaseDemo/group/add

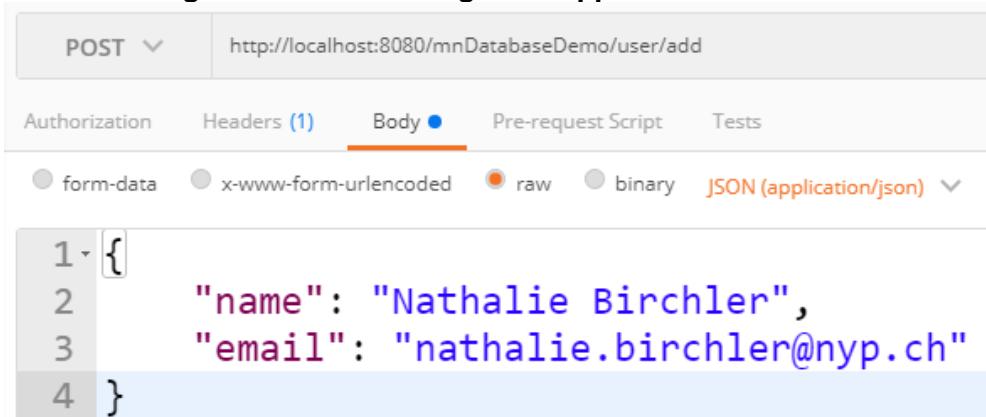
Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON

```

1 {
2   "name": "Gruppe 6"
3 }
  
```

- User hinzufügen ohne Zuordnung zu Gruppe:



POST http://localhost:8080/mnDatabaseDemo/user/add

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json) ▼

```

1 {
2   "name": "Nathalie Birchler",
3   "email": "nathalie.birchler@nyp.ch"
4 }
  
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

- User hinzufügen mit Zuordnung zu Gruppe:



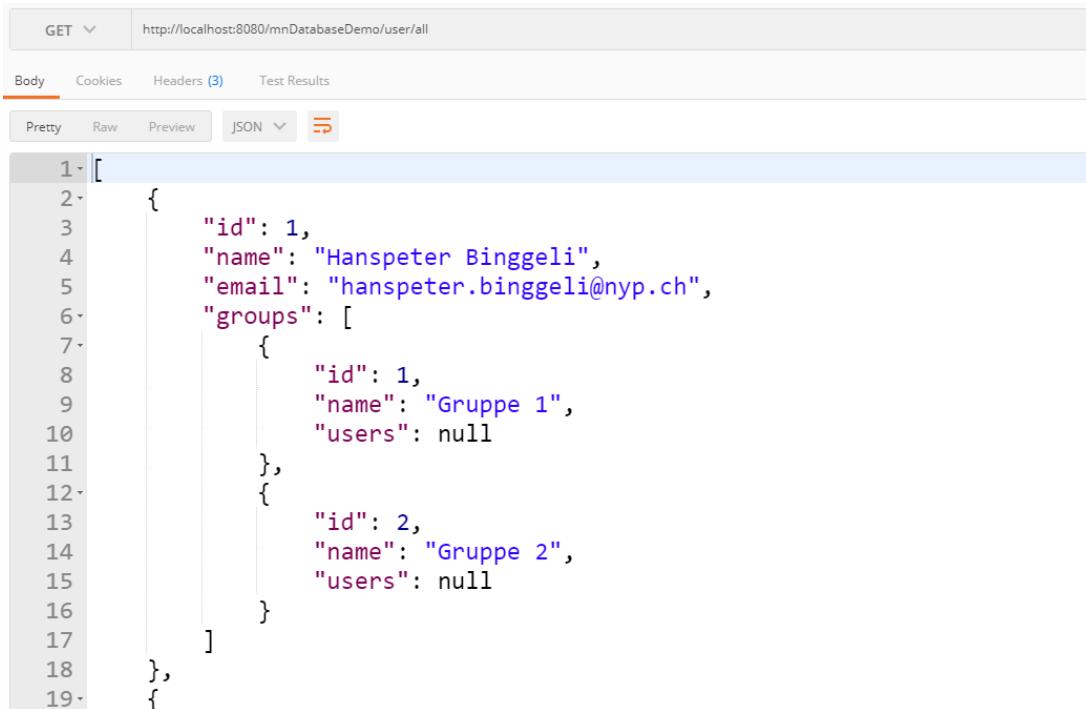
POST <http://localhost:8080/mnDatabaseDemo/user/add>

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
1 {  
2   "name": "Nathalie Birchler 2",  
3   "email": "nathalie.birchler2@nyp.ch",  
4   "groups": [  
5     {  
6       "id": "1"  
7     },  
8     {  
9       "id": "12"  
10    }  
11  ]  
12 }
```

- Alle User ermitteln



GET <http://localhost:8080/mnDatabaseDemo/user/all>

Body Cookies Headers (3) Test Results

Pretty Raw Preview JSON

```
1 [  
2   {  
3     "id": 1,  
4     "name": "Hanspeter Binggeli",  
5     "email": "hanspeter.binggeli@nyp.ch",  
6     "groups": [  
7       {  
8         "id": 1,  
9         "name": "Gruppe 1",  
10        "users": null  
11      },  
12      {  
13        "id": 2,  
14        "name": "Gruppe 2",  
15        "users": null  
16      }  
17    ],  
18    {  
19      }
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren****- Alle Gruppen ermitteln**

GET <http://localhost:8080/mnDatabaseDemo/group/all>

Body Cookies Headers (3) Test Results

Pretty Raw Preview JSON ↗

```
1 [  
2 {  
3     "id": 1,  
4     "name": "Gruppe 1",  
5     "users": [  
6         {  
7             "id": 2,  
8             "name": "Adrian Krebs",  
9             "email": "adrian.krebs@nyp.ch",  
10            "groups": null  
11        },  
12        {  
13            "id": 1,  
14            "name": "Hanspeter Binggeli",  
15            "email": "hanspeter.binggeli@nyp.ch",  
16            "groups": null  
17        },  
18        {  
19            "id": 14,  
20            "name": "Adrian Krebs",  
21            "email": "adrian.krebs@nyp.ch",  
22            "groups": null  
23        },  
24    ]
```

7.5 Eigene Repository-Abfragen

Die Standard-Repositories bieten Funktionen zum Ermitteln von allen Datensätzen oder einem Datensatz mit einer bestimmten ID, dasselbe für das Erstellen von einem oder mehreren Datensätzen, das Aktualisieren und das Löschen.

Möchte ich aber z.B. ein anderes WHERE haben als die ID, oder eine Sortierung, oder mehr als 1 Datensatz, so muss ich eigene Repository-Abfragen, d.h. eigene Queries erstellen.

In diesem Beispiel wurde das Beispiel aus Kapitel 7.3 um eigene Repository-Funktionen erweitert, weil die Standard-Funktion von JpaRepository nicht mehr reichten. Die Datenbank, die Models und die DTOs wurden nicht verändert. Nur der Controller und das Repository haben Änderungen erfahren. Eingesetzt wurden folgende Varianten von eigenen Queries:

- Derived Queries
- Named Queries

Link: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories>

7.5.1 Derived Queries

Spring Data JPA bietet sogenannte „Derived Queries“ an. Falls die Methoden der JPA Basisklassen (z.B. JpaRepository) wie findById, findAll, etc. nicht mehr reichen, ist dies eine einfache Möglichkeit, ohne viel Code die gewünschten Statements abzusetzen.

Anstelle dass SQL geschrieben werden muss oder mehrere Codezeile für eine Abfrage nötig sind reicht es, im jeweiligen Repository-Interface eine Methode mit bestimmten Methodennamen zu definieren.

Am Aufbau des Methodennamens erkennt Spring Data JPA dann automatisch, was für ein Query er ausführen muss.

7.5.1.1 Beispiel Derived Query:

Alle User mit einem bestimmten Namen ermitteln

SQL wäre: Select * from user where name =

Repository-Code:

```
public interface UserRepository extends JpaRepository<User, Long> {
    public List<User> findByName(String name);
}
```

Es reicht die Methode „findByName“ zu definieren und diese dann im Controller wie folgt aufzurufen:

```
List<User> usersFromDB = userRepository.findByName(name);
```

Im Repository kann alleine durch die korrekte Definition des Methodennamens das gewünschte Query ausgeführt werden.

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

7.5.1.2 Aufbau von Derived Queries:

Nachfolgende Tabelle zeigt die verschiedenen Möglichkeiten von Derived Queries.

Keyword	Beispiel	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false

Tabelle 5: Derived Queries Spring Data JPA

7.5.2 Named Queries

Named Queries sind eine andere Möglichkeit um die gewünschten SQL-Statements abzusetzen.

In diesem Fall wird mit einer Annotation im Repository ein SQL-Statement einer Methode zugeordnet.

7.5.2.1 Beispiel Named Query - Select

Alle User mit einem bestimmten Namen ermitteln

SQL wäre: Select * from user where name =

Repository-Code:

```
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("select u from User u where u.name = ?1")
    public List<User> findByNameNamed(String name);

}
```

Die Methode wird im Controller wie folgt aufgerufen:

```
List<User> usersFromDB = userRepository.findByNameNamed(name);
```

Das gewünschte SQL-Statement wurde auf die Methode gemappt. ?1 entspricht dem 1.Methodenparameter (name).

Bei 2 Parameter wäre das Query:

```
//Named Query
@Query("select u from User u where u.name = ?1 and u.email = ?2")
public List<User> findByNameAndEmailNamed(String name, String email);
```

7.5.2.2 Beispiel Named Query – Update & Delete

Named Queries funktionieren nicht nur für Selects, sondern auch für Update oder Delete. Nachfolgend ein paar Beispiele:

```
@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);

@Modifying
@Query("delete from Employee e where firstName = ?1")
void deleteUsingSingleQuery(String firstName);
```

Beim Update und Delete ist die Annotation @Modifying nötig.

8 RESTful-API - Data Transfer Objects (DTO)

8.1 Variante ohne DTO

Bei der Variante ohne DTOs gibt es nur die Entities (Model-Klasse mit JPA-Annotations).

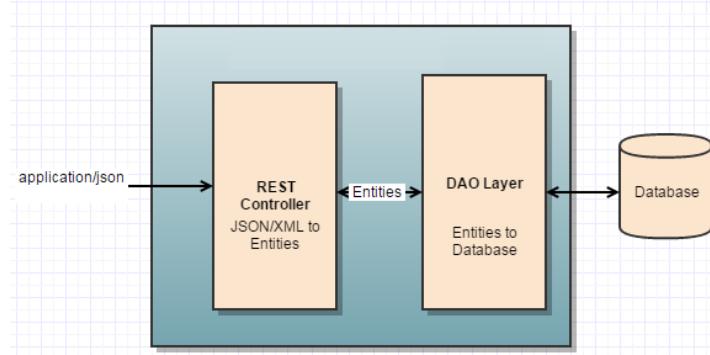


Abbildung 18: Datentransfer ohne DTO

Der REST Controller erhält einen JSON-String und wandelt diesen in ein Model-Objekt (JPA Entity) um. Diese JPA-Entity wird dann über den DAO Layer (Repository) direkt in die DB geschrieben.

8.2 Variante mit DTO

Es wird ein DTO-Layer erzeugt. Anstelle, dass der JSON-String direkt in eine JPA Entity umgewandelt wird, wird dieser zuerst in ein DTO umgewandelt und danach das DTO in die JPA Entity. Der JSON-String muss also denselben Aufbau aufweisen wie das DTO, kann jedoch anders sein als die JPA Entity.

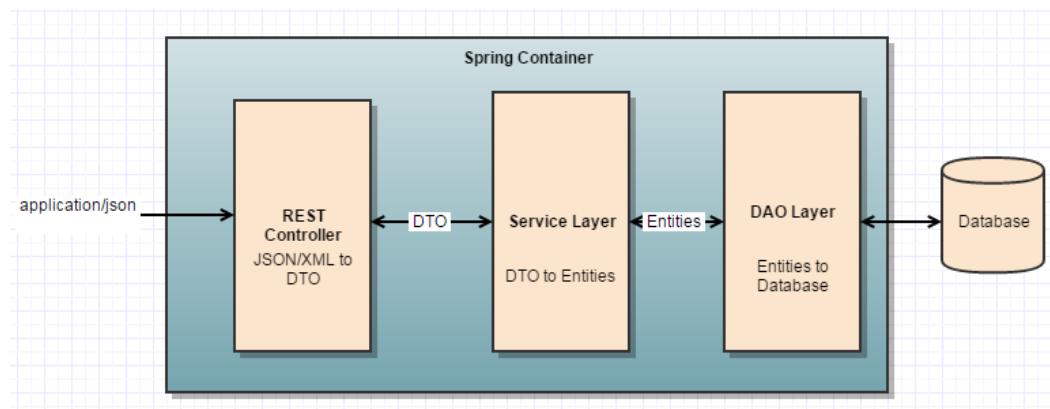


Abbildung 19: Datentransfer mit DTO

8.3 Vorteile von DTOs

Das Problem bei der Variante ohne DTO ist, dass die Objekte, welche als JSON an die RESTful-API gesendet werden zwingend so aufgebaut sein müssen wie die Datenbank, da sie 1:1 auch für die DB-Kommunikation verwendet werden. Möchte ich jedoch andere Objekte übergeben (z.B. anderer Aufbau, zusätzliche Attribute, weniger Attribute) so stösse ich an die Grenzen mit Variante 1. Die DTOs kann ich so aufbauen, wie ich das JSON schlussendlich auch wünsche.

Andere Vorteile sind:

- DTOs können den Anforderungen angepasst werden. Sie sind hervorragend geeignet, wenn im JSON-String nur ein Teil der DB-Felder vorhanden sein muss.
- DTOs verhindern, dass im Persistence-Layer (Entities) Annotations gebraucht werden, welche nichts mit der Persistenz zu tun haben (z.B. `JsonIgnore` um DB-Felder im JSON zu ignorieren)
- Vollständige Kontrolle über die Attribute im JSON.
- Bieten mehr Flexibilität beim Zuordnen von Beziehungen.
- Verwendung verschiedener DTOs für verschiedene Medientypen.
- Einfachere Erstellung einer Swagger-Dokumentation (<https://swagger.io/>)

8.4 Nachteile von DTOs

- Zusätzlicher Code nötig, auch wenn DTO grösstenteils ähnlich wie Entity.
- Kann zu Datenleck oder Sicherheitsproblem führen wenn unsauber gearbeitet wird.

9 Authentifizierung & Autorisierung mit JWT

9.1 Authentifizierung vs. Autorisierung

Authentifizierung und Autorisierung werden häufig verwechselt oder gleichgestellt. Sie sind jedoch nicht dasselbe! Nachfolgende Tabelle zeigt die Unterschiede.

Authentifizierung	Autorisierung
Identifizierung eines Benutzers.	Überprüfung, ob Benutzer eine Funktion ausführen darf.
Beispiel: Login bei einer Webseite mit Username & Passwort. Webseite identifiziert Benutzer eindeutig anhand Zugangsdaten	Beispiel: Webseite mit Admin- und User-Rechte. Webseite prüft bei jeder Funktion, ob eingeloggter User die nötigen Rechte hat, diese auszuführen.
Wer bist du?	Was darfst du?

Tabelle 6: Authentifizierung vs. Autorisierung

→ Wir benötigen Beides bei einer REST-API

9.2 Authentifizierung bei REST-API

9.2.1 Übersicht

REST ist Statuslos, d.h.:

- Jede Abfrage steht für sich
- Jede Abfrage muss Identität und Berechtigungen prüfen (Authentifizierung & Autorisierung)

D.h. bei jedem REST-Request muss der Client dem Server die Informationen mitgeben, damit dieser die Identität und die Berechtigung sicherstellen kann.

Bei REST bieten sich u.a. folgende Möglichkeiten zur Authentifizierung an:

- **Basic Authentication:**
Bei jedem Request werden Username & Passwort mitgeschickt.
- **Token Authentication:**
Client fordert bei der REST-API einen Token an (Token = Irgendein String, z.B. zufällig). Bei jedem weiteren Request sendet der Client den Token mit und der Server identifiziert User anhand von Token.

9.2.2 Basic Authentication

Bei der Benutzung der HTTP Basic Authentication werden bei jedem REST-Request der Username und das Passwort mitgegeben. D.h. der Server identifiziert bei jeder Abfrage den User neu und prüft dann anhand Username & Passwort dessen Rechte. Username und Passwort werden für den Versand als Base64 kodiert und dann auf der REST-API wieder dekodiert.

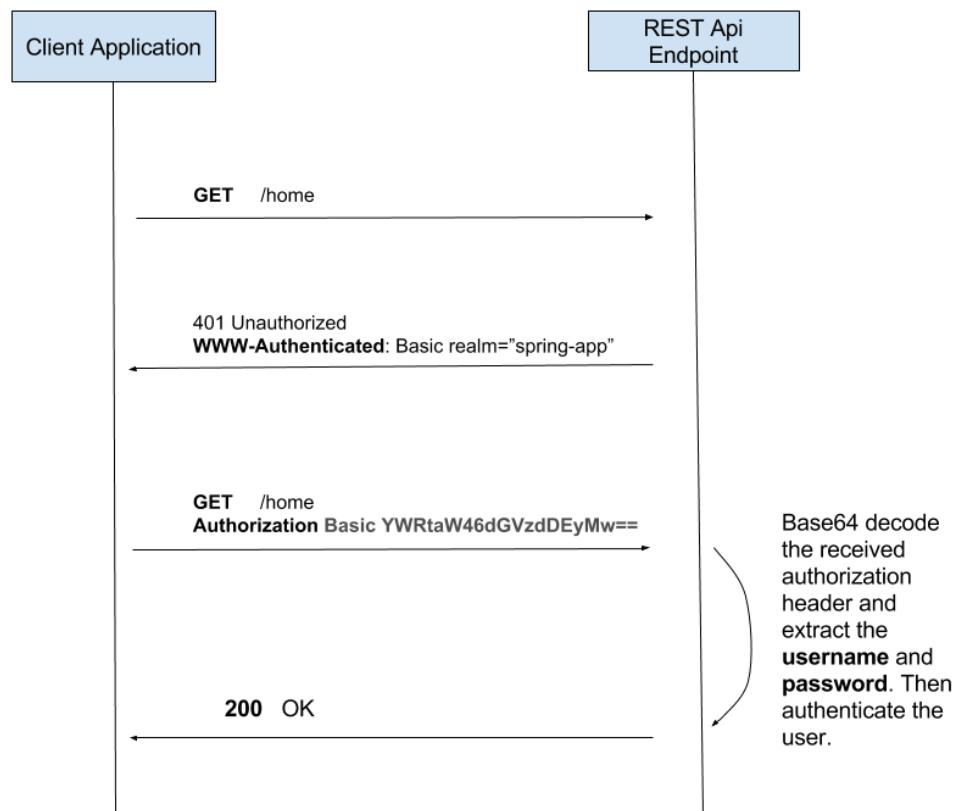


Abbildung 20: Basic Authentication

9.2.2.1 Vor- und Nachteile von Basic Authentication

Vorteile +	Nachteile -
Leicht zu implementieren.	PW wird praktisch im Klartext übertragen. Nur Base64-Kodierung.
Viele Libraries vorhanden.	PW wird jedes Mal übertragen. Das macht das System anfällig für Man in the Middle Attacken.
PW kann auf Server sicher gespeichert werden.	Passwort muss im Client gespeichert werden, damit es jedes Mal mitgegeben werden kann.
Schnell	SSL/TSL ist Pflicht, damit Daten verschlüsselt übertragen werden

Tabelle 7: Vor- und Nachteile von Basic Authentication

9.2.3 Token Authentication

Bei der Token-Authentication macht der Client eine Abfrage an den Server um sich einzuloggen (Username, Passwort) und erhält dann ein zufälliger String, das sogenannte Token, vom Server. Bei allen weiteren Anfragen sendet der Client nur noch diesen String (Token) und der Server kann dann anhand des Tokens den User identifizieren.

1. Client ruft /login Endpoint auf und übergibt Username und Passwort.
2. Server generiert ein Token. Dies ist irgend ein String (Random, verschlüsselt, etc., je nach Methode)
3. Server gibt das Token an den Client zurück.
4. Client speichert bei sich das Token.
5. Client gibt bei jedem Aufruf eines REST-Endpoints (z.B. /users um alle User zu ermitteln) das Token im Header mit.
6. Server weiss anhand Token, welcher User den Endpoint aufgerufen hat und ob dieser berechtigt ist, die Aktion auszuführen.

9.2.3.1 Vorteile von Token Authentication:

- Benutzername und Passwort werden nur einmal übertragen
- PW-Übertragung kann verschlüsselt geschehen.
- Token kann verschlüsselt werden.
- Username und Passwort müssen nicht im Client gespeichert werden.
- Third Party Authentication Server (z.B. Google Login, Facebook Login) ist möglich.

9.3 JSON Web Token (JWT)

9.3.1 Was ist JWT?

JWT steht für JSON Web Token. JSON Web Token sind ein wichtiger Standard zur Sicherung von REST-APIs. Es dient dazu, JSON-Objekte verschlüsselt und signiert zwischen Client und Server zu übertragen. Ein JWT besteht aus einem Header, einer Payload und einer Signatur. Weitere Infos hierzu später.

Der wichtigste Anwendungsfall für JWT ist die Authentifizierung, d.h. die Verwendung eines Access Tokens.

Bei der Token-Authentication macht der Client ja bekanntlich eine Abfrage an den Server um sich einzuloggen (Username, Passwort) und erhält dann ein zufälliger String, das sogenannte Token, vom Server. Bei allen weiteren Anfragen sendet der Client nur noch diesen String (Token) und der Server kann dann anhand des Tokens den User identifizieren.

JWT ist die empfohlene Art der Token Authentication.

9.3.2 Token Authentication ohne JWT

Es gibt diverse Wege, eine Token Authentication umzusetzen. Eine Variante ohne JWT wäre z.B. wie folgt:

7. Client ruft /login Endpoint auf und übergibt Username und Passwort.
8. Server generiert ein Token aus einem zufälligen String und legt das Token in der Datenbank ab (Beispiel-Token: kdfkllasdf23923030223934ldfalsfl12!ä\$12!à
9. Server gibt das Token an den Client zurück.
10. Client speichert bei sich das Token.
11. Client gibt bei jedem Aufruf eines REST-Endpoints (z.B. /users um alle User zu ermitteln) das Token im Header mit.
12. Server prüft jedes Mal über die DB ob das Token valid ist und ermittelt anhand des Tokens über die DB den eingeloggten User.

Probleme dieser Variante:

Das Problem dieser Variante ist, dass das Token keine Informationen über den User beinhaltet. Alle User-Informationen muss die REST-API anhand des Tokens aus der DB ermitteln.

Es wäre aber auch möglich, User-Informationen in das Token zu füllen. Dann jedoch, wären diese nicht verschlüsselt und für alle lesbar (ausser wenn eigene Verschlüsselung implementiert wird). Auch verfügt diese Variante über keine Signatur, mit welcher überprüft werden kann, ob das Token zwischen Client und Server unerlaubterweise verändert wurde (z.B. bei Man in the Middle-Attacke)

9.3.3 Token Authentication mit JWT

1. Client ruft /login Endpoint auf und über gibt Username und Passwort.
2. Server generiert ein JWT. Das JWT beinhaltet u.a. die ID des eingeloggten Users und andere gewünschte User-Informationen als JSON-Objekt. Diese Daten werden jedoch verschlüsselt und dann zudem noch signiert. Nur der Server, welcher das JWT erstellt hat, kennt dessen Aufbau und die Signatur.
Das generierte JWT ist unleserlicher Hash.
3. Server gibt das JWT an den Client zurück.
4. Client speichert bei sich das JWT.
5. Client gibt bei jedem Aufruf eines REST-Endpoints (z.B. /users um alle User zu ermitteln) das JWT im Authorization Header mit dem Bearer Schema, mit.
6. Server prüft jedes Mal die Gültigkeit des JWT, d.h. u.a. ob die Signatur stimmt. Zudem kann der Server aus dem JWT die User-Informationen auslesen und die benötigten Abfragen durchführen.

Nachfolgende Abbildung zeigt das Login-Prozedere und den Aufruf einer geschützten URL:

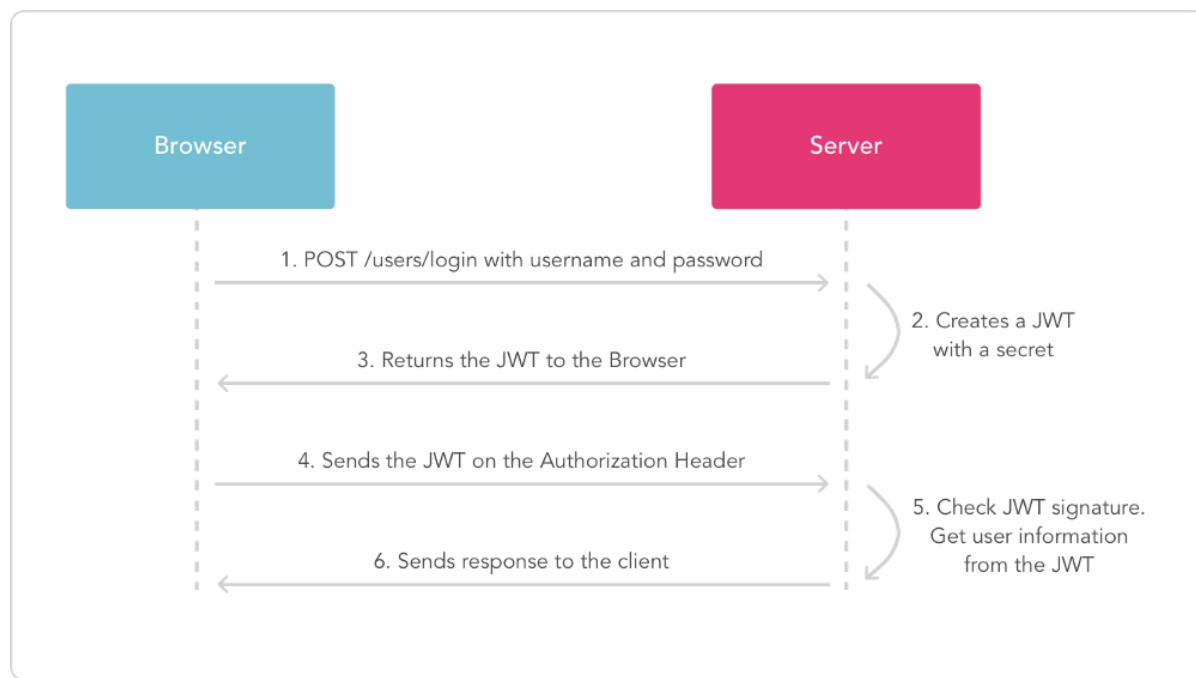


Tabelle 8: Token Authentication mit JWT

9.3.3.1 Ein paar Vorteile von JWT Authentication auf einen Blick:

- Sichere Datenübertragung von JSON-Objekten (Verschlüsselung). Verschlüsselungsalgorithmen kann aus einer Anzahl möglichen selber ausgewählt werden.
- User-Informationen sind in Token erhalten und müssen nicht immer Anderweitig (z.B. über DB) ermittelt werden.
- Signatur gewährleistet, dass die Daten vom korrekten Absender kommen. Nur der Server, welcher die Signatur kennt, kann das Token lesen.
- Gültigkeitsdauer eines Tokens kann bei Generierung angegeben werden und ist beschränkt. Token, welches abläuft, erhöht die Sicherheit der REST-API (Auto-Logout wenn lange inaktiv).
- Leichtgewichtig, d.h. kurzer JWT-Token (String) trotz viel Informationen (JSON-Objekt)

9.3.4 Aufbau eines JWT

Das JWT ist ein einfacher String, der im Header übertragen wird. Ein JWT hat den folgenden Aufbau:

HEADER . PAYLOAD . SIGNATURE

Ein JWT-Token besteht also aus drei Teilen, die durch Punkte (.) voneinander getrennt sind. Hier sind beispielhafte Belegungen dieser Teile:

- Header: "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9"
- Payload: "eyJpZCI6IjEyMzQ1Njc4OTAiLCJuYW1lIjoiTWFydGhhIFRlc3RlcJ9"
- Signatur: "AfF8fGzbhhpS9k-rgJt7RIzaUnP9phwmnPTku2fs4o0"

9.3.4.1 Header:

Der Header beschreibt den Signatur- und/oder Verschlüsselungsalgorithmus und den Token-Typ. Die Daten werden im JSON-Format abgelegt und Base64-kodiert. Das vorliegende Beispiel ([eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9](#)) sieht im Klartext so aus:

```
{"alg": "HS256", "typ": "JWT"}
```

Im Beispiel wurde die Payload des JWT mit HMAC SHA256 (abgekürzt "HS256") signiert.

9.3.4.2 Payload:

Als Payload wird das JSON-Objekt bezeichnet, das aus einer "beliebigen" Anzahl von Key/Value-Paaren besteht. Diese Key/Value-Paare werden "Claims" genannt. Der Payload wird ebenfalls Base64-kodiert. Im Beispiel

([eyJpZCI6IjEyMzQ1Njc4OTAiLCJuYW1lIjoiTWFydGhhIFRlc3RlcJ9](#)) hat der Payload folgenden Inhalt:

```
{ "id": "1234567890", "name": "Joel Holzer" }
```

9.3.4.3 Signatur

Das letzte Element ist die Signatur, die aus dem Header und der Payload berechnet wird. Der Aufbau der Signatur wird von einem Standard (RFC 7515) definiert. Die Signatur wird dadurch erzeugt, dass der Header und der Payload im Base64 kodierten und durch einen Punkt getrennten Format mit der spezifizierten Hashmethode gehashed wird.

Das JWT von diesem Beispiel ist nun also wie folgt:

[eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9](#).
[eyJpZCI6IjEyMzQ1Njc4OTAiLCJuYW1lIjoiTWFydGhhIFRlc3RlcJ9](#). AfF8fGzbhhpS9k-rgJt7RIzaUnP9phwmnPTku2fs4o0

Auf der Webseite <https://jwt.io/> können Sie ein JWT für eine gewünschte Payload (JSON-Objekt) erzeugen. Dies kann helfen, den Aufbau eines JWT besser nachvollziehen zu können.

9.4 Authentifizierung mit JWT & Spring Boot

9.4.1 Beispielprojekt

Sie finden auf GIT-Hub das Projekt „**LoginJWTRestAPI**“. Dieses stellt folgende Funktionen zur Verfügung:

- Registrierung
- Login mit E-Mail Adresse und Passwort. REST-API generiert JWT-Token mit User-ID als Inhalt und gibt dieses an Client zurück.
- Ermitteln aller in der DB vorhandenen User. Nur nach Login möglich. JWT-Token von Login muss als Authorization Header (Bearer) übergeben werden.

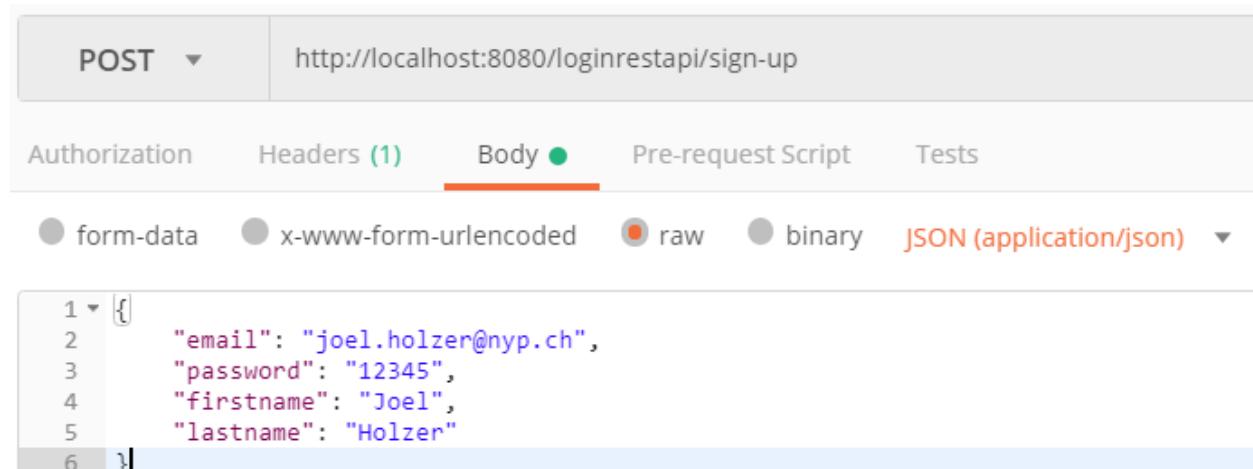
Das Beispielprojekt basiert auf folgendem Link:

<https://auth0.com/blog/implementing-jwt-authentication-on-spring-boot/>

9.4.2 Beispielprojekt testen

Nachfolgend sehen Sie die Postman-Requests zum Testen des Beispielprojekts.

9.4.2.1 Registrierung



POST <http://localhost:8080/loginrestapi/sign-up>

Authorization Headers (1) Body Tests

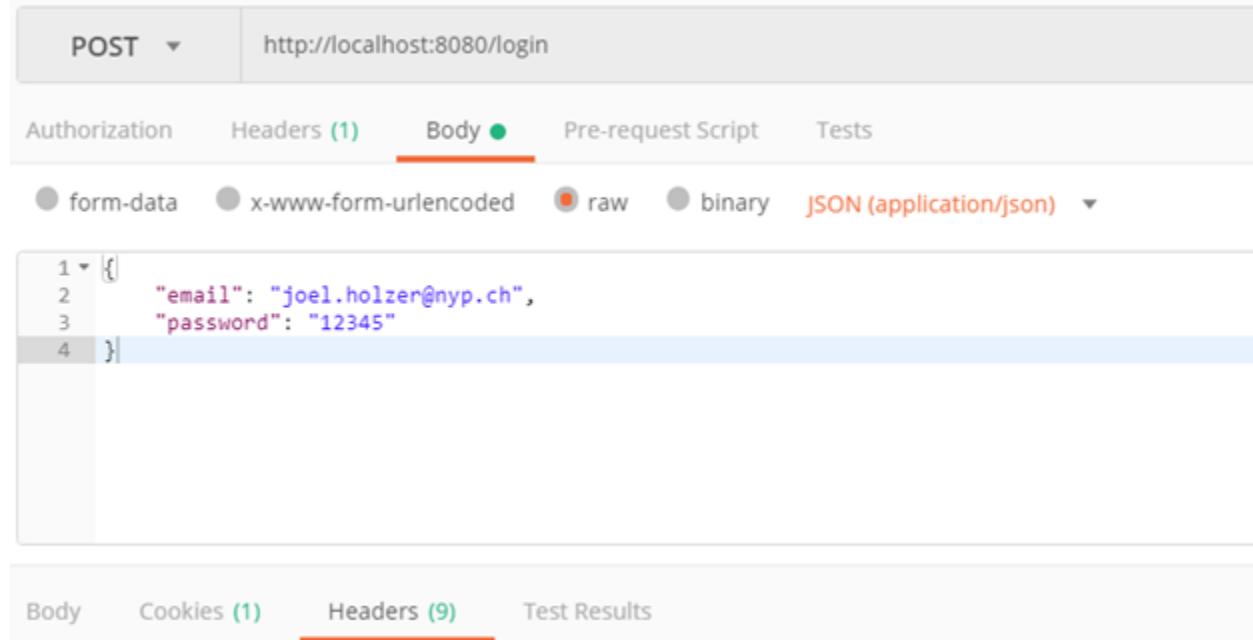
form-data x-www-form-urlencoded raw binary **JSON (application/json)**

```
1 {  
2   "email": "joel.holzer@nyp.ch",  
3   "password": "12345",  
4   "firstname": "Joel",  
5   "lastname": "Holzer"  
6 }
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

9.4.2.2 Login

▶ Login



The screenshot shows a POST request to `http://localhost:8080/login`. The Body tab is selected, showing a JSON payload:

```

1 [{}]
2   "email": "joel.holzer@nyp.ch",
3   "password": "12345"
4 ]
  
```

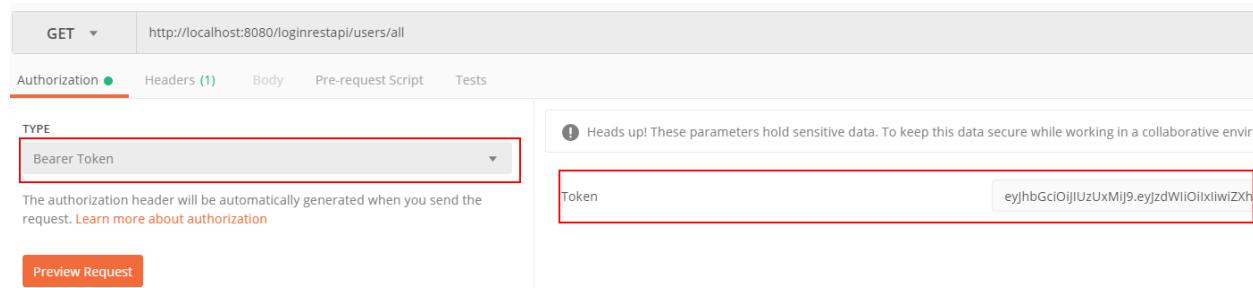
Below the request, the Headers tab is selected, showing a single header:

Authorization → Bearer eyJhbGciOiJIUzUxMiJ9eyJzdWliOiIxliwiZXhwIjoxNTMwNDczMzY5fQ.6W28B66-c_eE81b_S

Beim Login gibt die REST-API das generierte JWT im Authorization-Header zurück. Dieses muss nun bei jedem weiteren Request im Authorization Header mitgegeben werden.

9.4.2.3 Alle User ermitteln

Ist nur mit JWT-Token aufrufbar.



The screenshot shows a GET request to `http://localhost:8080/loginrestapi/users/all`. The Authorization tab is selected, showing a dropdown set to "Bearer Token". A note says: "The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)". To the right, a note says: "Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, consider using a secure connection or using a local machine." A "Token" input field contains the JWT token: `eyJhbGciOiJIUzUxMiJ9eyJzdWliOiIxliwiZXhwIjoxNTMwNDczMzY5fQ.6W28B66-c_eE81b_S`.

9.4.3 Weitere Links

JWT: <https://jwt.io/>

JWT & Spring Boot-Beispiele:

- <https://github.com/murraco/spring-boot-jwt>
- <https://medium.com/@xoor/jwt-authentication-service-44658409e12c>

10 Sicherheit bei REST-APIs

Im vorhergehenden Kapitel haben Sie bereits einige Möglichkeiten zur Authentifizierung und Autorisierung von Clients bei REST-APIs kennengelernt.

Neben der Authentifizierung und Autorisierung gibt es eine Menge anderer Punkte, welche beachtet werden sollen um eine möglichst sichere REST-API zu programmieren.

Nachfolgend werden einige der Empfehlungen der OWASP erläutert. Weitere Informationen, siehe https://www.owasp.org/index.php/REST_Security_Cheat_Sheet.

10.1 HTTPS

REST-APIs sollten nur HTTPS-Endpunkte zur Verfügung stellen, d.h. Daten zwischen Client und REST-API nur über HTTPS übertragen werden. Dies schützt einerseits die hin- und her transferierten Daten, andererseits wird die Integrität der transferierten Daten gewährleistet.

10.2 Authentifizierung & Autorisierung

Siehe Kapitel 9.

10.3 Input Validierung

Input-Felder, sei es ein GET-Parameter in der URL oder ein JSON-Objekt, sollte nie vertraut werden. Es gelten folgende Grundregeln:

- Länge, Range, Format und Typ validieren
- Regex-Validierung
- Validation/Sanitation Libraries verwenden
- Größenlimit für REST-Request festlegen
- Validierungsfehler loggen. Ein Client, welcher in kurzer Zeit viele Validierungsfehler produziert, ist meistens nicht vertrauenswürdig.
- SQL-Injection und Cross Site Scripting verhindern

Weitere Informationen zur Umsetzung einer Input-Validierung mit Spring Boot finden Sie im Kapitel 11.

10.4 Content-Type validieren

Der Content-Typ, d.h. das Format in welchem die Daten übertragen werden (z.B. JSON) sollte fest definiert werden. Andere Content-Typen als die definierten sollten nicht möglich sein und zurückgewiesen werden.

10.5 Endpoint-Aufrufe einschränken

Nicht jeder Endpoint, d.h. jede URL/Funktion der REST-API muss über das Internet zugreifbar sein. Häufig wird die REST-API auch nur von einer Webseite oder einem System im gleichen Netz (z.B. Intranet, DMZ) benutzt. Die Endpoints sollten daher mit Firewall-Regeln und IP-Konfigurationen vor unerlaubtem Zugriff geschützt werden.

11 Input Validierung mit Spring Boot

11.1 Warum eine Input Validierung?

Jeder REST-Endpoint hat bestimmte Input-Parameter, sei es in Form eines GET-Parameters in der URL oder bei POST/PUT/etc. im Body als JSON-Objekt. Nachfolgend ein Beispiel-JSON zum Hinzufügen eines Users:

```
{  
    "name": "Joel Holzer",  
    "email": "joel.holzer@nyp.ch"  
}
```

Es ist wichtig, dass so keine unerwünschten Werte ins System kommen, welche z.B. eine falsche Darstellung, eine Exception oder ein Security Leak verursachen können.

Im obenstehenden Beispiel müsste z.B. folgendes validiert werden:

- Attribut „name“ ist vorhanden und nicht leer.
- Attribut „email“ ist vorhanden und hat ein gültiges E-Mail-Format
- E-Mail Adresse existiert noch nicht im System, d.h. kein anderer User mit gleicher E-Mail Adresse vorhanden

Falls ein Validierungsfehler auftritt, soll die REST-API ein http-Error-Code zurückgeben (z.B. 400) und die entsprechende Fehlermeldung.

11.2 Standard-Validierung von Spring Boot

Spring Boot führt bereits von Haus aus automatisch ein paar Input-Validierungen durch. Z.B.:

- Wird der Content Type JSON gewählt, so werden andere Content Typen nicht akzeptiert und ein http-Error 415 – Unsupported Media Type, zurückgegeben.
- Fehlerhafter JSON-Aufbau, z.B. User-Objekt erwartet, Buch-Objekt übergeben, führt zu http-Error 400 – Bad Request.

11.3 Weitere Input-Validierungen mit Spring Boot

11.3.1 Bean Validation

Für die weitere Validierung empfiehlt sich eine Bean Validation mit dem Hibernate Validator. Dieser ist bereits in der Dependency „Spring Boot Starter Web“, welche für die REST-API genutzt wird, enthalten.

Bei der Bean-Validation werden die Attribute der Java-Beans validiert. D.h. es werden die Attribute der DTOs validiert. Dazu werden Annotations verwendet.

11.3.1.1 Schritt 1: Annotations im Bean hinzufügen

Das Bean ist meistens das DTO, welches zwischen Client und REST-API transferiert wird.

```
public class UserDto {
    private Long id;

    @NotNull
    @Size(min=2, message="name muss mindestens 2 Zeichen aufweisen.")
    private String name;

    @NotNull
    @Email(message="email muss ein korrektes Format aufweisen.")
    private String email;

    //usw...
}
```

In diesem Beispiel muss der Name ≥ 2 Zeichen lang sein und das E-Mail dem E-Mail-Format entsprechen. Ansonsten wird die in den Klammern definierte Fehlermeldung zurückgegeben.

Unter folgendem Link sind die verfügbaren Annotations für die Validierung aufgelistet:

<http://www.springbootutorial.com/spring-boot-validation-for-rest-services>

11.3.1.2 Schritt 2: Annotation in Controller-Methode hinzufügen

11.3.1.3 Damit die Validierung auch im Zusammenhang mit dem REST-API-Aufruf funktioniert, muss nun im Controller die Annotation **@Valid** vor dem Objekt, welches validiert werden soll, eingefügt werden.

```
@PostMapping(path="user/add")
public @ResponseBody String addNewUser(@Valid @RequestBody UserDto
userDto) { ... }
```

11.3.1.4

11.3.2 Eigene Bean Validierungen (Custom Validation)

Häufig reichen die Standard-Validierungen mit den Annotations wie @Size, @Max, @Email, etc. nicht aus für komplexere Validierungen. Ein gutes Beispiel ist die Validierung, ob eine E-Mail Adresse in der DB bereits existiert.

Hier muss ja beim Validieren in der DB geprüft werden, ob die E-Mail Adresse bereits existiert.

Für solche erweiterten Validierungen können eigene Annotations erstellt werden, welche dann auch im Bean (DTO) angewendet werden.

Beispiel:

```
@UniqueEmail(message="email muss ein einmalig sein.")
private String email;
```

Hier wurde die Annotation **@UniqueEmail** erstellt, welche dann eine eigene Validierung ausführt für die E-Mail Adresse.

Für eine solche Validierung gehen Sie wie folgt vor:

11.3.2.1 Schritt 1: Annotation-Typ definieren

Wir definieren die eigene Annotation **@UniqueEmail**. Dazu erstellen wir eine neue Java-Datei mit folgendem Inhalt:

```
@Documented
@Constraint(validatedBy = UserIdValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface UniqueEmail {
    String message() default
    "{ch.nyp.validation.validator.UniqueEmail.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

11.3.2.2 Schritt 2: Validator-Klasse erstellen

Klasse, welche die Validierung durchführt. In der Methode isValid wird validiert.

```
public class UserIdValidator implements
ConstraintValidator<UniqueEmail, Long> {
    @Override
    public boolean isValid(Long value, ConstraintValidatorContext
context) {
        hier ist die eigene Valdierung drin, z.B. DB-Überprüfung. Value ist
        der Wert, welcher das Attribut, welchem die Annotation gesetzt wurde
        (d.h. dann das email-Attribut vom UserDto), gesetzt hat.
        Return True, wenn Wert OK, False wenn Validierungsfehler.
    }
}
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

11.3.2.3 Schritt 3: Eigene Annotation in Bean setzen

11.3.2.4 Nun noch die eigene Annotation @UniqueEmail beim Attribut, welches validiert werden soll, setzen.

```
@UniqueEmail(message="email muss ein einmalig sein.")  
private String email;
```

11.3.2.5

Weitere Informationen zur Erstellung eigener Bean Validation Annotations finden Sie unter
<http://javasampleapproach.com/spring-framework/create-custom-validation-spring>

11.3.3 Beispielprojekt

Auf GitHub finden Sie das Beispielprojekt „ValidationRestAPI“, welches mit Bean Validation (inkl. eigener Annotation) JSON-Objekte (DTOs) validiert.

12 DB-Transaktionen

12.1 Einleitung

Wenn wir ein Datenbanksystem ohne Verwendung von Transaktionen einsetzen, gehen wir von folgenden „falschen“ Annahmen aus:

- **Isolation:**
 - o Nur ein Nutzer greift auf die Datenbank zu
 - Lesend
 - Schreibend

→ In Wahrheit: Viele Nutzer und Anwendungen lesen und schreiben gleichzeitig
- **Atomizität:**
 - o Anfragen und Updates bestehen aus einer einzigen Aktion. DBMS kann nicht mitten in der Aktion ausfallen.

→ In Wahrheit: Auch einfache Anfragen bestehen oft aus mehreren Teilschritten

12.1.1 Beispiel zu Atomizität:

Beispiel aus der Bankenwelt. Überweisung von 500 CHF von Konto „Joel Holzer“ auf Konto „Adrian Krebs“:

Ausgangslage:

Konto-Nr.	Konto	Saldo
1	Joel Holzer	5'000 CHF
2	Adrian Krebs	10'000 CHF

Datenbank-Abfragen für Verschiebung von 500 CHF:

```
update Konto set Saldo = Saldo - 500 where KtoNr = 1;
update Konto set Saldo = Saldo + 500 where KtoNr = 2;
```

D.h. eine Abfrage für -500 bei Joel Holzer und zweite Abfrage für +500 bei Adrian Krebs.

Was passiert, wenn der Rechner mittendrin abstürzt?

Genau für solche Fälle wurden Transaktionen eingeführt. Diese führt eine Gruppe von Statements (hier die beiden Inserts) entweder komplett oder gar nicht aus.

12.2 Was ist eine DB-Transaktion?

Eine Transaktion wird dann benötigt, wenn man eine Gruppe von Statements entweder komplett, oder gar nicht ausführen möchte. Hierbei wird eine Menge der Datenbankänderungen zusammengefasst und als eine Gruppe ausgeführt. Wie z. B. eine Kontobuchung von einem Konto A auf ein Konto B. Erst wenn der Betrag von Konto A abgebucht wird und bei Konto B zugeht, ist die Transaktion beendet, ansonsten muss der Ausgangszustand der Datenbank wieder hergestellt werden.

Weiter wird mit Transaktionen auch die Isolation gewährleistet, d.h. verhindert, dass durch gleichzeitiges Lesen & Schreiben mehrerer Anwendungen auf dieselbe DB ein Problem auftritt.

Ein Datenbankmanagement-System (DBMS) wie MySQL führt automatisch jede SQL-Abfrage in einer Transaktion aus, damit gleichzeitiges Schreiben/Lesen möglich ist.

Möchten jedoch mehrere Abfragen in derselben Transaktion ausgeführt werden, müssen Sie dies explizit selber angeben, entweder per SQL-Befehl oder mit den Befehlen der eingesetzten Library (z.B. JPA).

BEGIN TRANSACTION

```
update Konto set Saldo = Saldo - 500 where KtoNr = 1;
```

```
update Konto set Saldo = Saldo + 500 where KtoNr = 2;
```

COMMIT TRANSACTION

12.3 Eigenschaften von Transaktionen - ACID-Prinzip

Transaktionen weisen folgende Eigenschaften auf:

1. **Atomicity (Atomarität):**
Eine Transaktion wird komplett ausgeführt, oder überhaupt nicht.
2. **Consistency (Konsistenz):**
Die Datenbank ist vor und nach einer Transaktion stets in konsistentem Zustand, d.h. alle Integritätsbedingungen sind erfüllt.
3. **Isolation (Isolation):**
Transaktion läuft isoliert gegenüber dem Einfluss anderer Transaktionen. Jede Transaktion wird nacheinander ausgeführt. Das DBMS regelt diese. D.h. auch wenn 2 Anwendungen gleichzeitig ein Insert machen, führt DBMS die Requests nacheinander aus.
4. **Durability (Dauerhaftigkeit):**
Ergebnisse einer Transaktion (nach Commit) sind dauerhaft gesichert, d.h. in DB gespeichert.

12.4 Fehler bei Transaktionsausführung

Es ist möglich, dass eine Transaktion abgebrochen wird, z.B. durch:

- **Verletzung von Constraints:**
Statement kann nicht ausgeführt werden, weil ein Constraint verletzt wird. z.B. kann User nicht gelöscht werden, wenn diesem noch ein Buch zugeordnet ist.
- **Prüfungen in Transaktion:** Komplexe Abfragen, welche Prüfungen (z.B. If, else) beinhalten und zu einem Abbruch führen.
- **Systemfehler:** z.B. Division durch Null in einer Transaktion
- **DBMS-Absturz:**
Datenbank Management System stürzt ab, z.B. wegen Stromausfall, Netzausfall, Softwarefehler, Hardware-Ausfall (Festplatte, andere Komponente).

Wenn nicht die ganze Transaktion ausgeführt werden kann, weil in der Mitte ein Fehler aufgetreten ist, wird die ganze Transaktion rückgängig gemacht.

Dies wird **Rollback** genannt.

12.4.1 Commit & Rollback

12.4.1.1 Rollback

Ein Rollback geschieht immer dann, wenn in der Datenbank ein Fehler auftritt. Die Datenbank wird auf den Status vor der Transaktion zurückgesetzt.

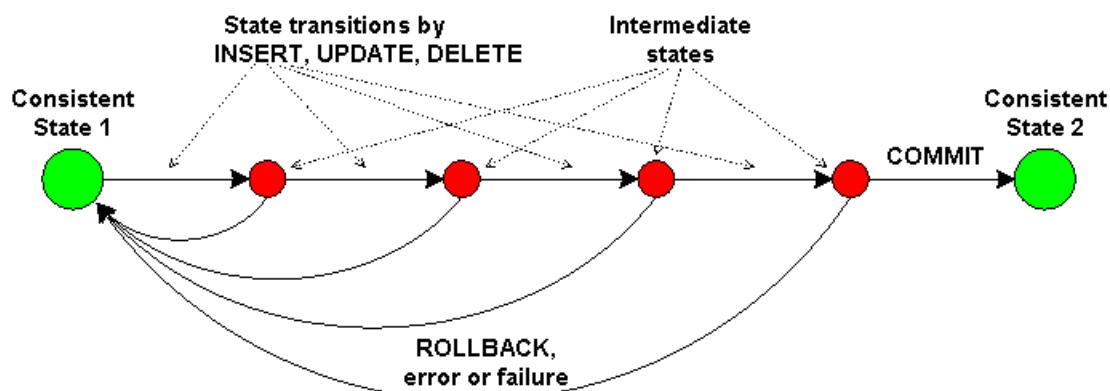


Abbildung 21: Transaktion Rollback

Es ist jedoch auch möglich, nur einen Teil der Transaktion zurückzusetzen. D.h. die Transaktion verfügt über verschiedene Savepoints, z.B. ein Savepoint nach Ausführung der Löschung, ein Savepoint nach Ausführung des Inserts in 3 Tabellen, etc. Es wird dann nur bis zu einem bestimmten Savepoint zurückgesetzt. Dies kann der Entwickler festlegen.

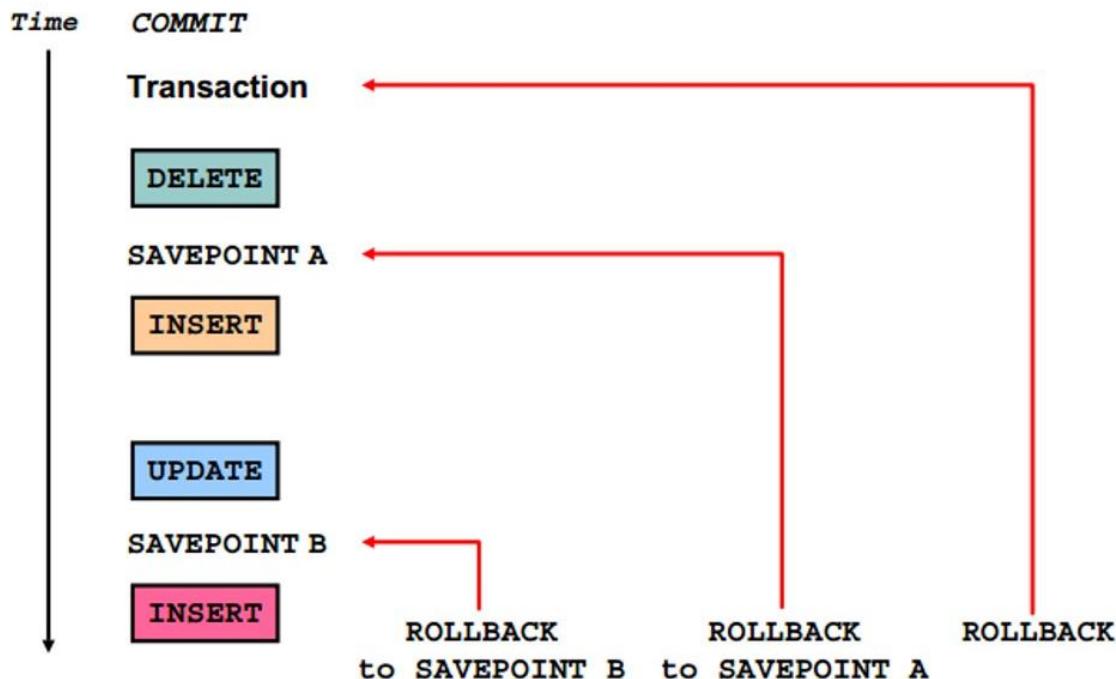


Abbildung 22: Transaktionen Rollback mit Savepoint

12.4.1.2 Commit

Wird die Transaktion vollständig ausgeführt, so erfolgt die Speicherung der Arbeit in Form einer Bestätigung durch COMMIT. Erst COMMIT speichert die Daten dauerhaft in der Datenbank ab.

12.5 DB-Transaktionen mit MySQL

DB-Transaktionen können direkt per SQL-Befehle im DBMS gemacht werden.

Wichtigste Befehle:

START TRANSACTION: Hier beginnt die Transaktion.

COMMIT: Commit. Dauerhafte Speicherung der Daten in DB.

ROLLBACK: Rollback von DB-Änderungen.

SAVEPOINT: Definition eines Savepoints, d.h. Zwischenziel, zu welchem zurückgesprungen werden kann mit dem Rollback.

Beispiele:

- **Start und Commit:**

```
START TRANSACTION;
UPDATE Konto SET saldo = saldo + 10 WHERE kontoid = 1;
UPDATE Konto SET saldo = saldo - 10 WHERE kontoid = 2;
COMMIT;
```

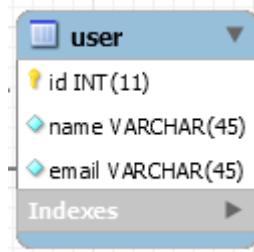
- **Savepoint und Rollback:**

```
START TRANSACTION;
LOCK TABLES orders WRITE;
INSERT DATA INFILE '/tmp/customer_info.sql'
INTO TABLE orders;
SAVEPOINT orders_import;
INSERT DATA INFILE '/tmp/customer_orders.sql'
INTO TABLE orders;
SELECT...
SAVEPOINT orders_import1;
INSERT DATA INFILE '/tmp/customer_orders1.sql'
INTO TABLE orders;
SELECT...
ROLLBACK TO SAVEPOINT orders_import1;
```

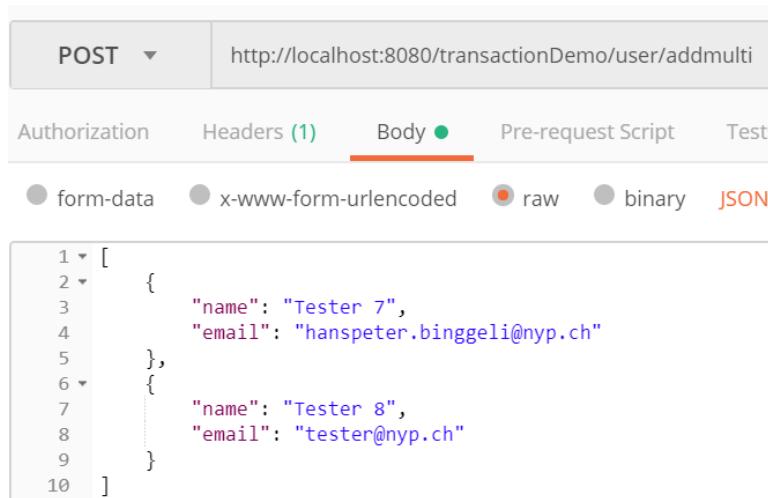
12.6 DB-Transaktionen mit Spring Boot

Nachfolgendes Beispiel soll zeigen, wie mit Spring Boot und JPA mehrere Statements innerhalb einer Transaktion ausgeführt werden können. Es handelt sich um das Projekt „DbTransactionRestAPI“ auf dem GitHub-Repository.

Das Beispiel besteht aus einer DB-Tabelle „User“.



In diesem Beispiel soll es möglich sein mit einem REST-Request mehrere User auf einmal in die Datenbank zu füllen. Die verschiedenen INSERT-Statements sollen in einer Transaktion ausgeführt werden und bei einem Fehler rückgängig gemacht werden.



```

1 [ 
2   {
3     "name": "Tester 7",
4     "email": "hanspeter.bingeli@nyp.ch"
5   },
6   {
7     "name": "Tester 8",
8     "email": "tester@nyp.ch"
9   }
10 ]
  
```

Abbildung 23: Postman – mehrere User einfügen (Transaktion)

Folgende Links können bei der Umsetzung von Transaktionen mit Spring und JPA helfen:

<https://spring.io/blog/2011/02/10/getting-started-with-spring-data-jpa/>

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#transactions>

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren****12.6.1 Schritt 1: Model, Repository, DTO, Controller erstellen**

Zuerst müssen Sie wie bei allen DB-Projekten folgende Klassen/Interfaces erstellen:

- Controller-Klasse für die Definition der REST-Endpoints
- User-Model
- UserRepository
- UserDto
- UserDtoConverter

12.6.2 Schritt 2: Service-Klasse und Service-Implementation erstellen

Bisher haben wir SQL-Statements jeweils über eine Repository-Methode ausgeführt, d.h. bisher galt: eine Repository-Methode = ein SQL-Statement.

Jeder Repository-Methodenaufruf wird von JPA standardmäßig in einer Transaktion ausgeführt.

Möchte ich mit einem REST-Call mehrere User hinzufügen, so könnte ich im Controller für jeden User die Save-Methode des Repositories aufrufen:

```
for (User user : users) {  
    userRepository.save(user);  
}
```

Problem: Jetzt wird für jeden User ein Insert gemacht, welches jedoch nicht in einer einzigen Transaktion ausgeführt wird. Für jeden Save-Befehl, d.h. für jeden Schleifendurchlauf wird eine neue Transaktion erstellt. Tritt beim Hinzufügen von 4 Usern beim 3.User ein Fehler auf, so wurden die beiden ersten bereits hinzugefügt und werden nicht mehr rückgängig gemacht.

Damit die Insert-Statements in einer Transaktion hinzugefügt werden muss eine Service-Klasse hinzugefügt werden. Diese wird dann vom Controller aufgerufen und führt die Transaktion aus:

12.6.2.1 Interface UserService:

```
public interface UserService {  
    void saveUsers(List<User> users);  
}
```

12.6.2.2 Klasse UserServiceImpl:

```
@Service
public class UserServiceImpl implements UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Transactional()
    @Override
    public void saveUsers(List<User> users) {
        for (User user : users) {
            userRepository.save(user);
        }
    }
}
```

12.6.2.3 Aufruf UserService in Controller:

```
@Controller
@RequestMapping(path="/transactionDemo")
public class MainController {
    @Autowired
    private UserRepository userRepository;

    @Autowired
    private UserDtoConverter userDtoConverter;

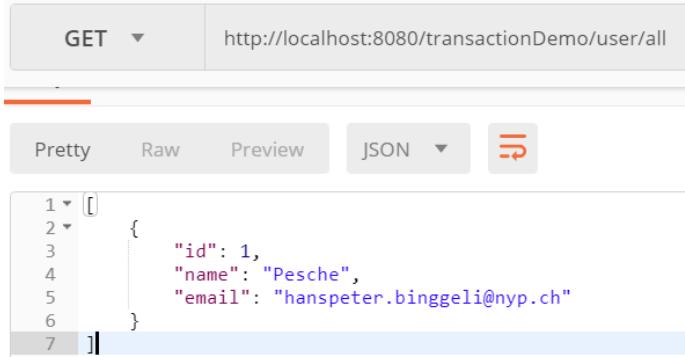
    @Autowired
    private UserService userService;

    @PostMapping(path="user/addmulti")
    public @ResponseBody String addNewUsers(@RequestBody List<UserDto>
                                            userDtos) {
        List<User> users = userDtos.stream().map(user ->
            userDtoConverter.convertToEntity(user))
            .collect(Collectors.toList());
        userService.saveUsers(users);
        return "Users saved";
    }
    //weitere Controller-Methoden
}
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

12.6.3 Testing mit Postman

Bevor Sie neue User hinzufügen, können Sie die bestehenden User in der DB wie folgt ermitteln:

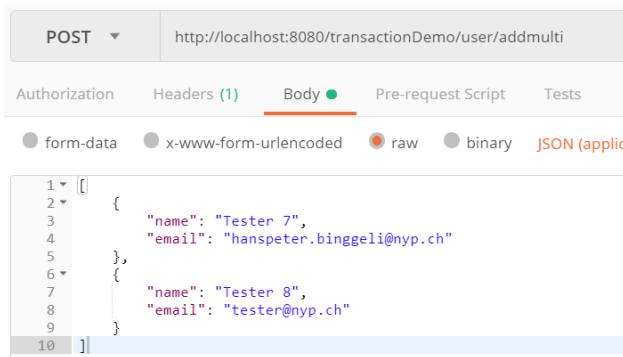


```

1 [ ]
2 {
3   "id": 1,
4   "name": "Pesche",
5   "email": "hanspeter.binggeli@nyp.ch"
6 }
7 ]
  
```

Nun sollen zuerst 2 Einträge erfolgreich hinzugefügt werden und dann 2 Einträge wo ein Eintrag fehlerhaft ist.

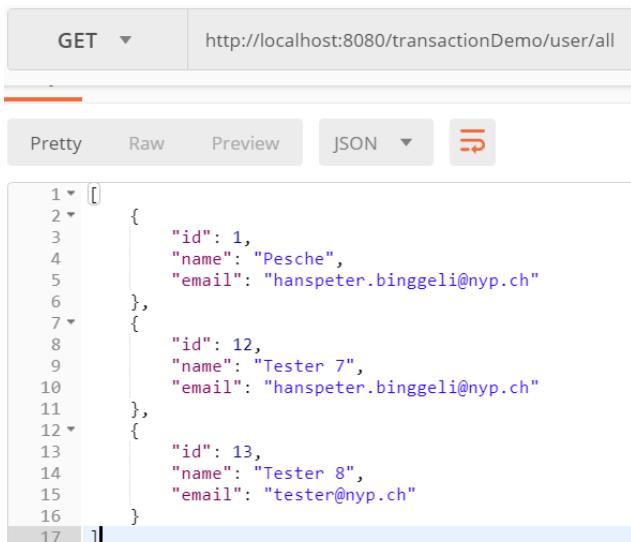
12.6.3.1 Erfolgreiche Abfrage



```

1 [ ]
2 [
3   {
4     "name": "Tester 7",
5     "email": "hanspeter.binggeli@nyp.ch"
6   },
7   {
8     "name": "Tester 8",
9     "email": "tester@nyp.ch"
10 }
  
```

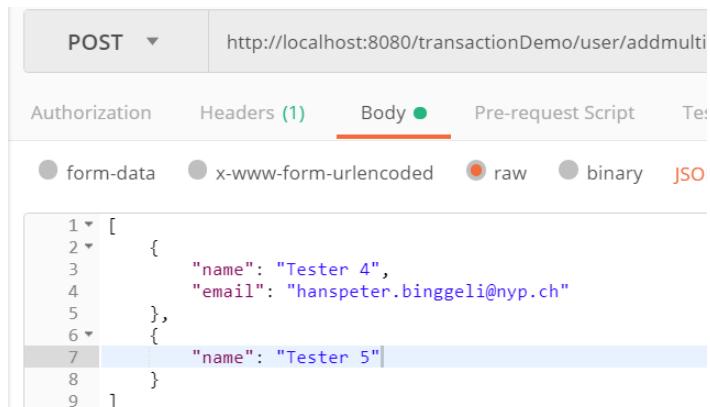
Nun sind in der Datenbank 3 User vorhanden:



```

1 [ ]
2 [
3   {
4     "id": 1,
5     "name": "Pesche",
6     "email": "hanspeter.binggeli@nyp.ch"
7   },
8   {
9     "id": 12,
10    "name": "Tester 7",
11    "email": "hanspeter.binggeli@nyp.ch"
12  },
13  {
14    "id": 13,
15    "name": "Tester 8",
16    "email": "tester@nyp.ch"
17  }
  
```

12.6.3.2

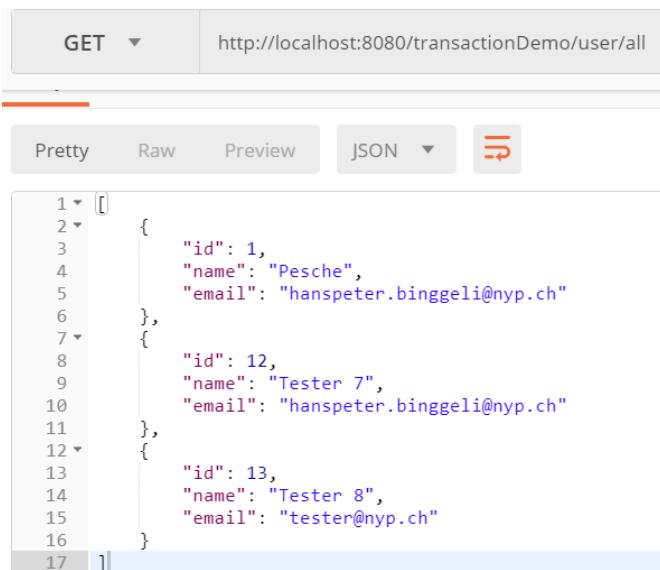
**Modul 223: Multi-User-Applikationen objektorientiert
realisieren****12.6.3.3 Fehlerhafte Abfrage**

```
POST http://localhost:8080/transactionDemo/user/addmulti

Body (1)
1 [ 
2   {
3     "name": "Tester 4",
4     "email": "hanspeter.binggeli@nyp.ch"
5   },
6   {
7     "name": "Tester 5"
8   }
9 ]
```

12.6.3.4

Die Zweite Person ist fehlerhaft. Da der Request nun in einer Transaktion ausgeführt wird, wird keiner der beiden User hinzugefügt. Daher sind weiterhin die bisherigen 3 User vorhanden.



```
GET http://localhost:8080/transactionDemo/user/all

Pretty Raw Preview JSON (1)

1 [ 
2   {
3     "id": 1,
4     "name": "Pesche",
5     "email": "hanspeter.binggeli@nyp.ch"
6   },
7   {
8     "id": 12,
9     "name": "Tester 7",
10    "email": "hanspeter.binggeli@nyp.ch"
11  },
12  {
13    "id": 13,
14    "name": "Tester 8",
15    "email": "tester@nyp.ch"
16  }
17 ]
```

13 REST-API Deployment auf Heroku

Heroku ist eine Could Application Plattform as a Service. Sie ermöglicht den Betrieb von REST-APIs, welche mit Spring Boot und Java entwickelt wurden. Auch können auf Heroku MySQL-Datenbanken betrieben werden.

Damit Sie die im üK entwickelte REST-API auch von Ihrer App ansprechen können, müssen Sie diese auf Heroku deployen, d.h. hochladen. Die REST-API ist dann über das Internet erreichbar.

Nachfolgende Abbildung zeigt den Systemaufbau.

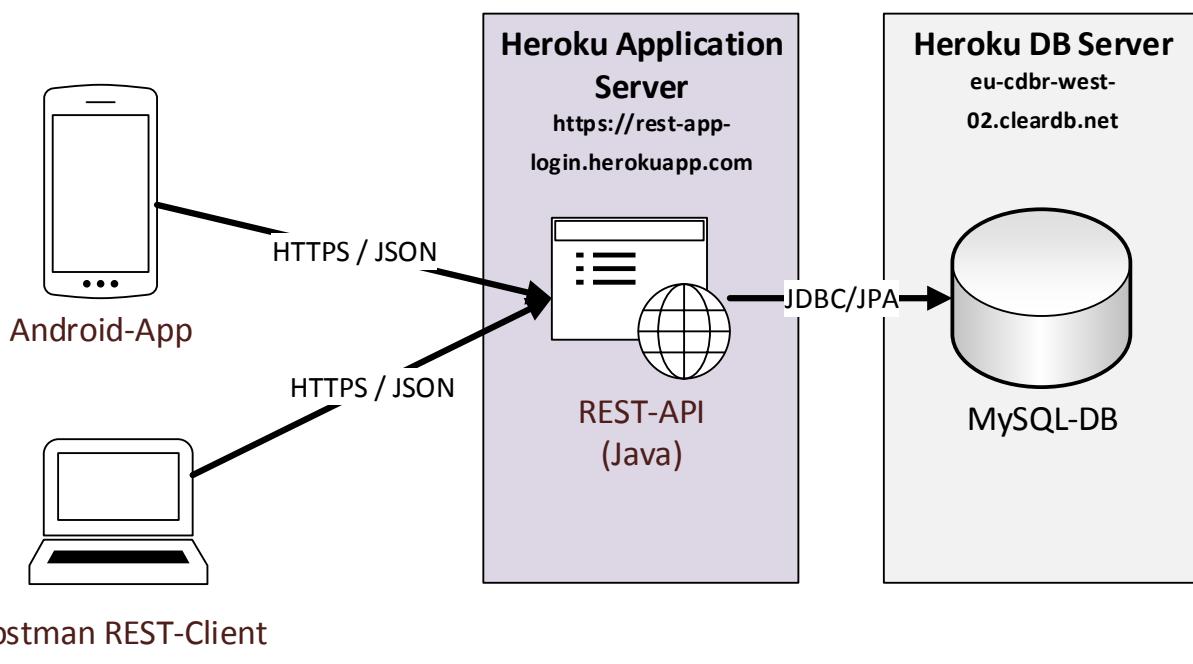


Abbildung 24: Heroku REST-API

13.1 REST-API deployen

13.1.1 Vorbereitungen

- Heroku Account erstellen und via. Kursleiter zum Heroku-Projekt hinzufügen.
- Heroku CLI herunterladen und installieren:
<https://devcenter.heroku.com/articles/heroku-cli>

13.1.2 Schritt 1: DB-Zugangsdaten in REST-API ändern

Bevor Sie die REST-API hochladen können, müssen Sie im **application.properties** die DB-Zugangsdaten ändern auf die DB von Heroku (siehe oben).

Beispiel für die REST-APP-Demo auf GitHub:

```

spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://eu-cdbr-west-
02.cleardb.net:3306/heroku_d173ecace30a069
spring.datasource.username=b5d741d263cb7d
spring.datasource.password=148b22ba
  
```

13.1.3 Schritt 2: Datenbank importieren

Verbinden Sie sich mit MySQL-Workbench und den Zugangsdaten (siehe Zusatzblatt) auf Ihre Heroku-Datenbank. Erstellen Sie ihre Datenbank mit einem SQL-Script (z.B. Export von lokaler DB)

13.1.4 Schritt 3: Ordner für GIT-Repository erstellen

Sie laden die App via GIT-Repository auf den Heroku-Server hoch. Dies muss in einem Verzeichnis gemacht werden, in welchem noch kein anderes GIT-Repository vorhanden ist. Dies ist bei Ihnen wahrscheinlich aber der Fall. Gehen Sie wie folgt vor:

1. Erstellen Sie am gewünschten Ort auf Ihrem PC einen neuen Ordner für das Heroku-Git-Repository.
2. Kopieren Sie den Code Ihrer REST-API (d.h. Projektordner) in das neu erstellte Verzeichnis.
3. Löschen Sie innerhalb des Projektordners das Verzeichnis **.git**

13.1.5 Schritt 4: Deploy der REST-API auf Server

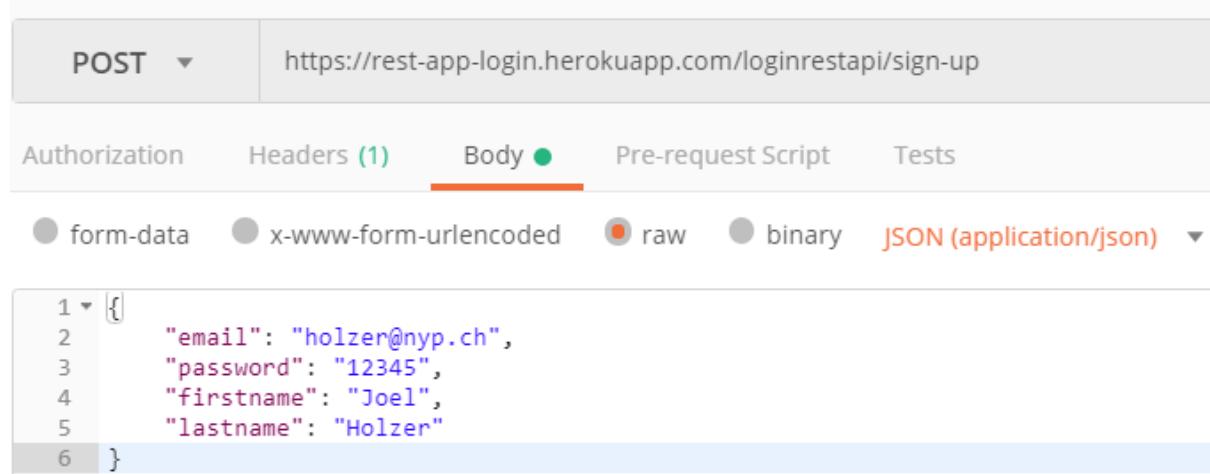
1. Konsole öffnen
2. In den bei Schritt 3 erstellten Ordner (In Projektordner) navigieren mit cd.
3. Eingabe von „heroku login“, Enter klicken
4. E-Mail Adresse und Passwort eingeben, Enter klicken
5. Nun folgende Befehle eingeben um das GIT-Repository zu erstellen:

```
$ cd my-project/  
$ git init  
$ heroku git:remote -a uek223-gruppe1
```

6. GIT-Repository pushen nach Heroku-Master

```
$ git add .  
$ git commit -am "first push  
$ git push heroku master
```

7. GIT-Repo wird nun gepusht und REST-API automatisch gestartet (kann ein paar Minuten dauern)

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren****13.2 REST-API testen****13.2.1 Registrieren**

POST ▾ https://rest-app-login.herokuapp.com/loginrestapi/sign-up

Authorization Headers (1) Body Tests

form-data x-www-form-urlencoded raw binary **JSON (application/json)** ▾

```
1 ▪ {  
2   "email": "holzer@nyp.ch",  
3   "password": "12345",  
4   "firstname": "Joel",  
5   "lastname": "Holzer"  
6 }
```

13.2.2 Login

POST ▾ https://rest-app-login.herokuapp.com/login

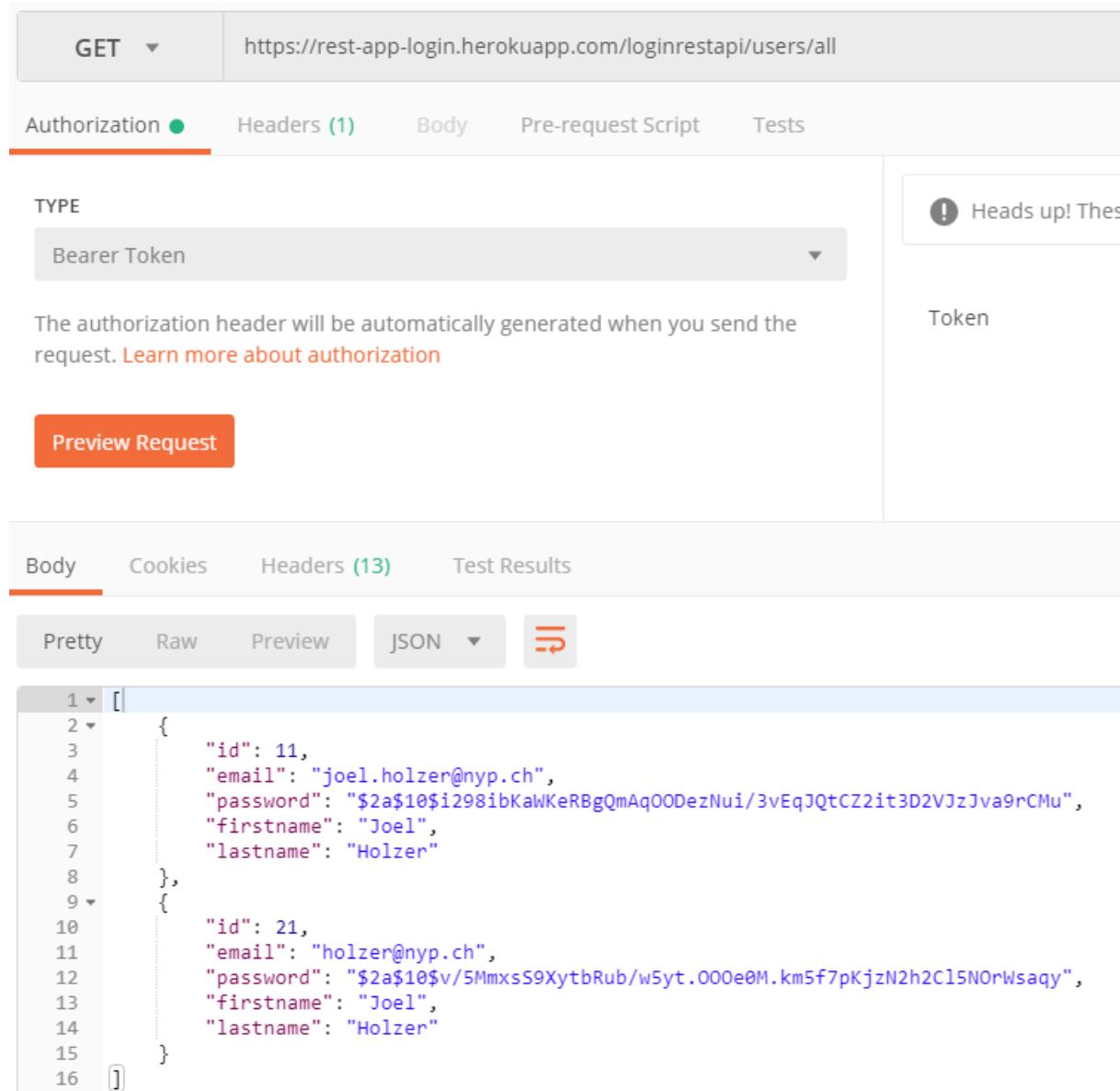
Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary **JSON (application/json)** ▾

```
1 ▪ {  
2   "email": "holzer@nyp.ch",  
3   "password": "12345"  
4 }
```

Body Cookies Headers (13) Test Results

Authorization → Bearer eyJhbGciOiJIUzUxMiJ9eyJzdWIiOiYMSIisImV4cCI6MTUzMjQ2Nzk1MH0.lgrLCPbiWYMUDXKhIA5WIRY9QiTr70vcEGs3LkeJLAn9GtubtAQ1j6YZlcOoQUtZWWUpvALHFk9-ndUtWQoEQ

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren****13.2.3 User-Liste abfragen**

GET ▾ https://rest-app-login.herokuapp.com/loginrestapi/users/all

Authorization ● Headers (1) Body Pre-request Script Tests

TYPE

Bearer Token

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Heads up! This token

Preview Request

Body Cookies Headers (13) Test Results

Pretty Raw Preview JSON

```
1 [ ]  
2 {  
3   "id": 11,  
4   "email": "joel.holzer@nyp.ch",  
5   "password": "$2a$10$i298ibKaWKeRBgQmAq00DezNui/3vEqJQtCZ2it3D2VJzJva9rCMu",  
6   "firstname": "Joel",  
7   "lastname": "Holzer"  
8 },  
9 {  
10  "id": 21,  
11  "email": "holzer@nyp.ch",  
12  "password": "$2a$10$v/5MmxsS9XytbRub/w5yt.000e0M.km5f7pKjzN2h2C15NOrWsaqy",  
13  "firstname": "Joel",  
14  "lastname": "Holzer"  
15 }  
16 ]
```

14 Android-App – REST-API anbinden

Dieses Kapitel zeigt anhand des Beispielprojekts „**AndroidRestApp**“ auf GitHub die Anbindung einer Android-App an eine REST-API, welche auf Heroku betrieben wird. Die App steuert die im Kapitel 13 erstellte und hochgeladene REST-API an.

Die App besteht aus 2 Activities, dem Login und einer Liste mit User. Der Ablauf ist wie folgt:

1. User loggt sich mit seinen Zugangsdaten ein. Beispiel:
Username: joel.holzer@nyp.ch
Passwort: 12345
App schickt den /login Request an die REST-API.
2. REST-API generiert ein JWT-Token und gibt dieses im Authorization Header an die App zurück.
3. App speichert den Authorization Header in den Shared Preferences, wechselt auf die UsersActivity (Anzeige aller User von der REST-API).
4. Die UsersActivity macht einen zweiten REST-Request zum Ermitteln der User. Dabei wird der Authorization Header (JWT-Token) mitgegeben, welcher aus den Shared Preferences ausgelesen wird.
5. REST-API gibt die User in JSON-Array von User-Objekten zurück.
6. App stellt die User in ListView dar.

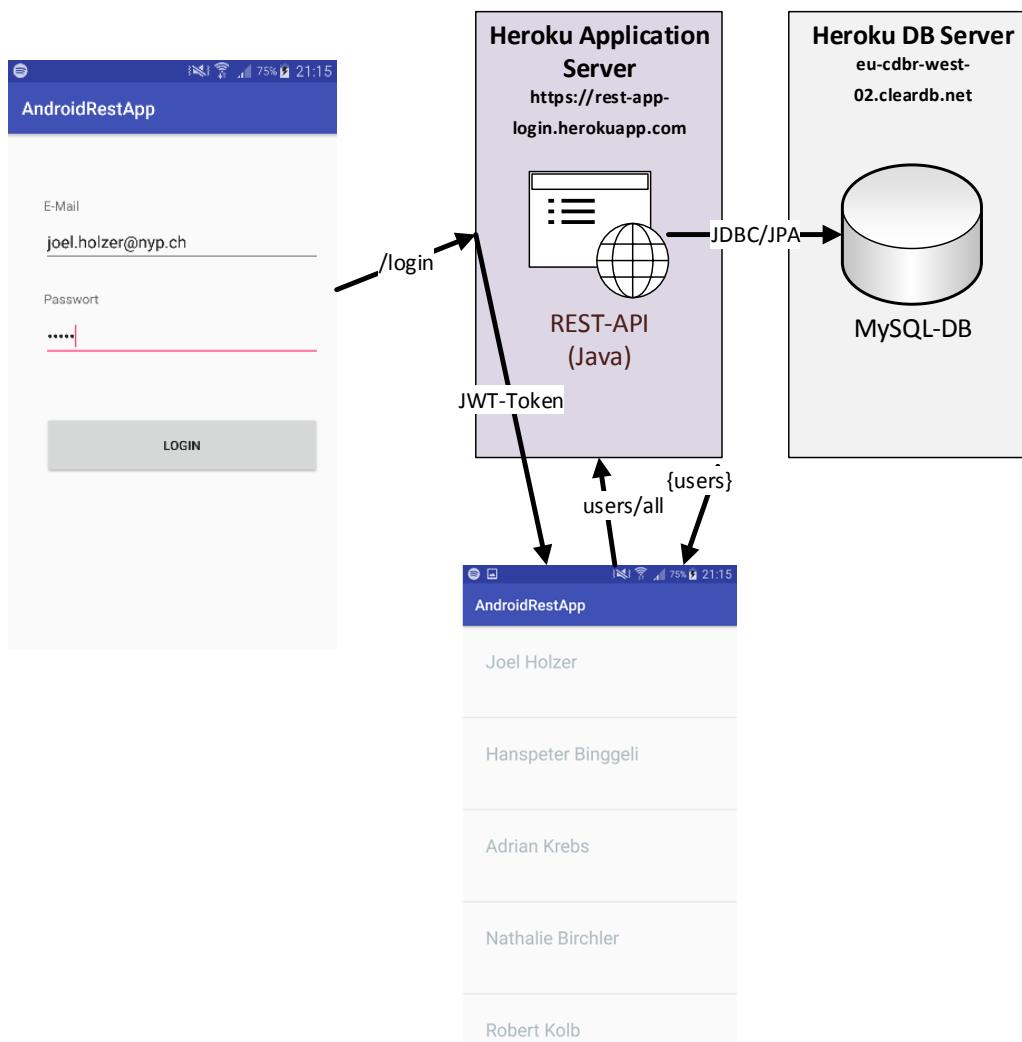


Abbildung 25: Android-App & REST-API auf Heroku

14.1 Android-App – REST-Anbindung umsetzen

Nachfolgend werden die Schritte zur Anbindung einer Android-App an eine REST-API erläutert (benötigte Klassen, Libraries, etc.)

14.1.1 Schritt 1: Dependencies hinzufügen

Folgende Einträge sind im „dependencies“-Block des build.gradle nötig:

```
implementation 'com.squareup.retrofit2:retrofit:2.4.0'
implementation 'com.squareup.retrofit2:converter-gson:2.4.0'
implementation 'com.squareup.okhttp:okhttp:2.4.0'
```

Damit werden folgende Libraries eingebunden:

- **Retrofit 2:** REST-Client für Android.
- **Retrofit2 JSON Converter:** Wandelt JSON-Objekte und Arrays in Java-Objekte um und umgekehrt.
- **OKHTTP:** Auch ein REST-Client für Android. Unterstützt Retrofit, damit erweiterte Funktionen, wie z.B. Authentifizierung, möglich sind.

14.1.2 Schritt 2: Internet-Berechtigung hinzufügen

Da REST-Request über das Internet laufen, muss im AndroidManifest (manifests/AndroidManifest.xml) folgende Permission hinzugefügt werden:

```
<uses-permission android:name="android.permission.INTERNET" />
```

14.1.3 Schritt 3: Models erstellen

Für alle JSON-Objekte, welche gesendet oder empfangen werden, muss das entsprechende Java-Model vorhanden sein. Das JSON wird dann mit GSON in das Java-Model umgewandelt.

```
public class User {

    private long id;
    private String email;
    private String password;
    private String firstname;
    private String lastname;
    //Todo: Getter & Setter
}
```

14.1.4 Schritt 4: Interface für REST-Requests erstellen

Erstellen Sie ein Interface, welches die REST-API-Endpoints (URLs) auf eine Java-Methode mappen.

```
public interface RestApi {  
  
    @POST("login")
    Call<Void> login(@Body User user);  
  
    @GET("loginrestapi/users/all/")
    Call<List<User>>
    getUsers(@Header(RestClient.AUTHORIZATION_HEADER) String
        authorization);  
}
```

@POST und @GET definiert die HTTP-Methode des Endpoints. In den Klammern folgt die URL (Nur alles, was nach der Domain kommt)

@Body definiert, dass dieser Methodenparameter dem Body des REST-Requests entspricht. D.h. bei login wird das User-Objekt in JSON-Form als Body mitgegeben.

@Header definiert die Header-Infos, welche mitgegeben werden. Im Beispiel wird der Authorization Header mitgegeben (JWT-Token)

14.1.5 Schritt 5: REST-Verbindung aufbauen

Damit die REST-Request abgesendet werden können, d.h. die in Schritt 5 definierten Methoden aufgerufen werden können, muss eine Retrofit-Instanz erstellt werden. Ich empfehle die Erstellung einer Klasse nach dem Singleton-Pattern. Diese ist sozusagen für die Herstellung der Verbindung zur REST-API zuständig.

```
public class RestClient {
    public static final String AUTHORIZATION_HEADER =
"Authorization";  
  
    private static final String REST_API_URL = "https://rest-app-
login.herokuapp.com";  
  
    private static RestApi restApiInstance;  
  
    public static RestApi getRestApi() {
        if (restApiInstance == null) {
            Retrofit retrofit = new Retrofit.Builder()
                .baseUrl(REST_API_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build();
            restApiInstance = retrofit.create(RestApi.class);
        }
        return restApiInstance;
    }
}
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

```

}
}
```

14.1.6 Schritt 6: REST-Requests durchführen und Daten im GUI anzeigen

Nachfolgender Beispielcode zeigt den Aufruf der Login-Funktion bei der REST-API.

Zuerst wird über die bei Schritt 5 erstellte Klasse die aktuelle Instanz von RestApi geholt.

Nun kann über diese Instanz die im Schritt 4 definierte Interface-Methode aufgerufen werden (hier login).

Damit der Request ausgeführt wird, ist der Befehl „enqueue“ nötig. Dieser erhält als Parameter einen Callback mit den beiden Teilen „onResponse“ und „ onFailure“.

REST-Request laufen asynchron ab, d.h. der Client (App) wartet nicht bis die Antwort zurückkommt, sondern setzt die Ausführung anderen Codes fort. Sobald die Antwort zurückkommt, wird der Callback aufgerufen.

onResponse: Wird im Erfolgsfall (Statuscode 200) aufgerufen. Hier können die Daten dann ins GUI gefüllt werden oder die entsprechende Aktion in der App ausgeführt werden.

onFailure: Wird im Fehlerfall aufgerufen (z.B. Statuscode 4xx und 5xx). Hier folgt die Ausgabe einer Fehlermeldung, Logging oder was auch immer bei einem Fehler geschehen soll

```

RestApi service = RestClient.getRestApi();
Call<Void> loginCall = service.login(user);
loginCall.enqueue(new Callback<Void>() {
    @Override
    public void onResponse(Call<Void> call, Response<Void>
        response) {
        String authorizationHeader =
            response.headers().get(RestClient.AUTHORIZATION_HEADER);

        //JWT abspeichern in Shared Preferences
        SharedPreferencesHelper.setAuthorizationHeader(getApplicationContext(),
            authorizationHeader);

        //Weiterleitung auf UsersActivity
        Intent intent = new Intent(getApplicationContext(),
            UsersActivity.class);
        startActivity(intent);
    }

    @Override
    public void onFailure(Call<Void> call, Throwable t) {
        //Falls Fehler aufgetreten ist: Fehlermeldung anzeigen
        Toast.makeText(getApplicationContext(),
            R.string.login_error, Toast.LENGTH_LONG).show();
    }
});
```

**Modul 223: Multi-User-Applikationen objektorientiert
realisieren**

Das obige Codebeispiel hat keinen Body als Resultat zurückgegeben. Daher ist die Methode „Void“. Wird ein Body zurückgegeben, sieht das wie folgt aus:

```
userCall.enqueue(new Callback<List<User>>() {
    @Override
    public void onResponse(Call<List<User>> call,
    Response<List<User>> response) {
        List<User> usersFromRestApi = response.body();
    }

    @Override
    public void onFailure(Call<List<User>> call, Throwable t) {
        ...
    }
});
```

Der Body vom Parameter „response“ von onResponse entspricht dem Rückgabewert der definierten Interface-Methode, d.h. dem Java-Objekt vom zurückgegebenen JSON. In obigen Beispiel wird ein JSON-Array von Users zurückgegeben, d.h. in Java eine List von User-Objekte.

14.1.7 Dateiübersicht des Beispielprojekt

app	Datei	Beschreibung
manifests	AndroidManifest.xml	Berechtigung für Internet setzen.
java	SharedPrefHelper	Lesen und Schreiben des Authorization Headers (JWT) in SharedPrefs.
ch.nyp.androidrestapp	User	Model für User-Objekte
helper	RestApi	Interface. Mapping von REST-Endpoints auf Java-Methoden.
model	RestClient	Erstellung des Rest-API-Objekts zum Ausführen von REST-Calls.
User	LoginActivity	Login-Ansicht & Aktion bei Klick auf Login-Button.
rest	UsersActivity	Darstellung aller User in einer ListView. Erscheint nach Login.
RestApi	UserAdapter	Füllt die User-Objekte in die ListView ab.
RestClient		
LoginActivity		
UserAdapter		
UsersActivity		

15 Abbildungsverzeichnis

Abbildung 1: Analyse und Design von Applikationen	8
Abbildung 2: Grobes Domänenmodell für eine Bibliothek	10
Abbildung 3: Grobes Domänenmodell für XelHa-Beobachtungsmodul	10
Abbildung 4: UML Klassendiagramm auf Domain-Ebene	11
Abbildung 5: Beispiel Use Case Diagramm Grundmodel	13
Abbildung 6: Beispiel Storyboard.....	25
Abbildung 7: Vom Domänenmodell zum ERD	26
Abbildung 8: 1:n Beziehung Domänenmodell vs. ERD	27
Abbildung 9: n:m Beziehungen.....	27
Abbildung 10: Beispiel UML Klassendiagramm	28
Abbildung 11: System Sequenzdiagramm (SSD)	32
Abbildung 12: Beispiel Sequenzdiagramm	33
Abbildung 13: RESTful-API Request Beispie.....	36
Abbildung 14: RESTful-API Beispiel	37
Abbildung 15: Hello RESTful-API Postman-Request	40
Abbildung 16: Neues Spring Projekt erstellen – Schritt 1	40
Abbildung 17: Objektrelationales Mapping.....	44
Abbildung 18: Datentransfer ohne DTO.....	68
Abbildung 19: Datentransfer mit DTO.....	68
Abbildung 20: Basic Authentication	71
Abbildung 21: Transaktion Rollback	85
Abbildung 22: Transaktionen Rollback mit Savepoint	86
Abbildung 23: Postman – mehrere User einfügen (Transaktion).....	88
Abbildung 24: Heroku REST-API.....	93
Abbildung 25: Android-App & REST-API auf Heroku.....	97

16 Tabellenverzeichnis

Tabelle 1: Änderungsgeschichte	3
Tabelle 2: Code-Beispiele / Demo-Projekte auf GitHub	6
Tabelle 3: Elemente eines UML Klassendiagramms.....	29
Tabelle 4: Elemente eines Sequenzdiagramm	35
Tabelle 5: Derived Queries Spring Data JPA.....	66
Tabelle 6: Authentifizierung vs. Autorisierung	70
Tabelle 7: Vor- und Nachteile von Basic Authentication.....	71
Tabelle 8: Token Authentication mit JWT	74