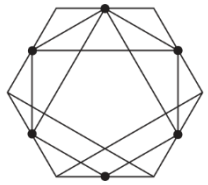
JAKARTA EE 8 PART I

ALLAITHY RAED @CODING-AKADEMIE

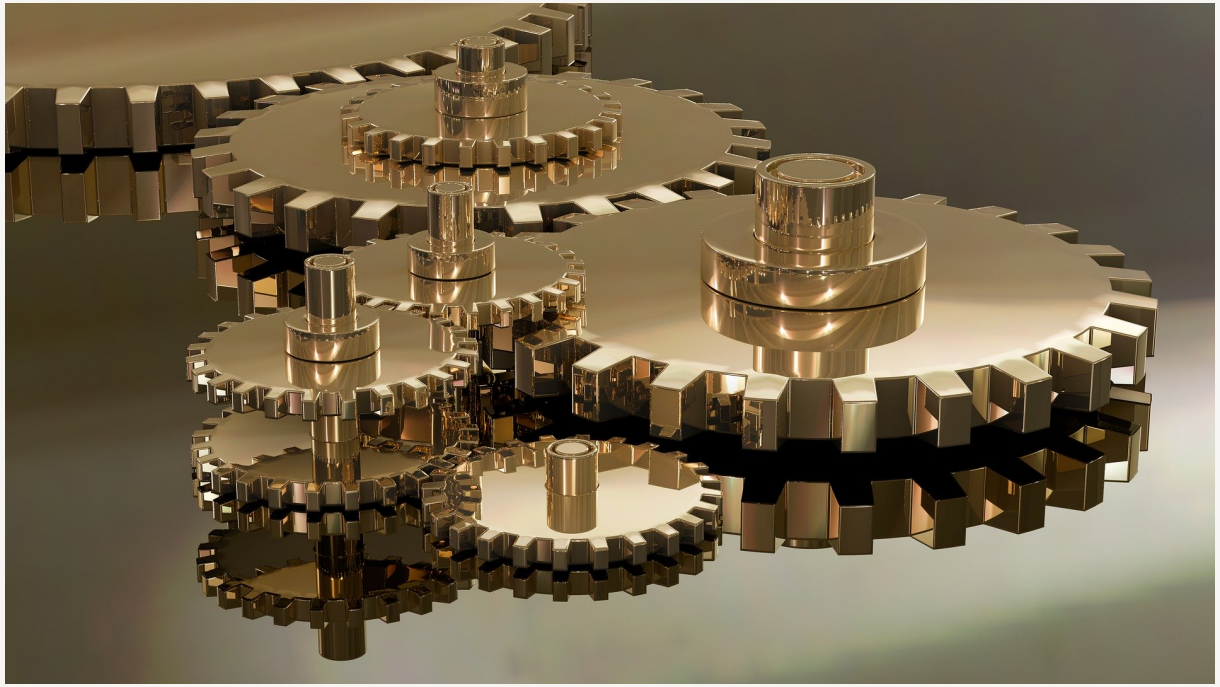
WAS IST JAVA EE?

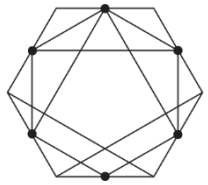
- Sammlung von Spezifikationen
- Webservices, Transactions, Security,..
- APIs = Application Programming Interface



APPLICATION SERVER?

- Software: Payara, Glasfisch, etc.
- Implementiert JAVA EE APIs
- Ermöglicht JAVA EE Services





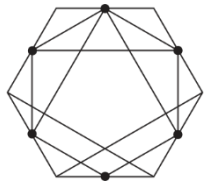
JAVA EE VS. JAKARTA

JAVA EE

- ❑ Bis 2019 bei Oracle
- ❑ Schwergewichtet
- ❑ Backend Framework

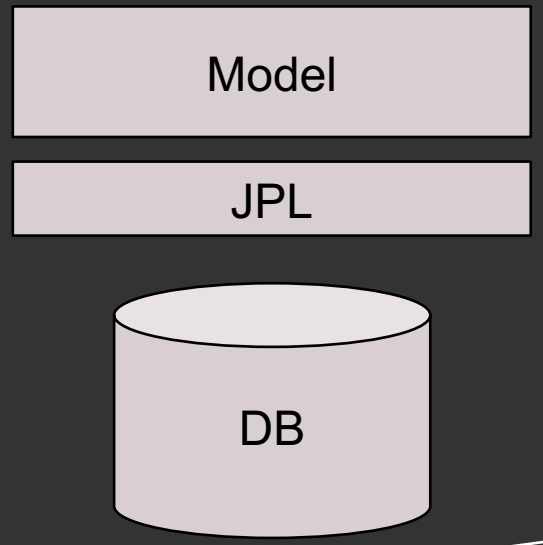
JAKARTA EE

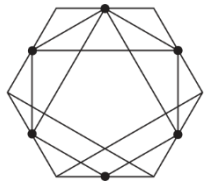
- ❑ Ab 2019 übernahm Eclipse Foundation Java EE
- ❑ Schlanke Version von JAVA EE
- ❑ Optimiert für native Cloud Application



WAS IST JAVA PERSISTENCE LANGUAGE (JPL)

- ❑ Eine Schicht zwischen Model und Datenbank
- ❑ Bildet Objekte auf relationale Datenbank ab
- ❑ Generiert DB Tabellen und deren Beziehungen





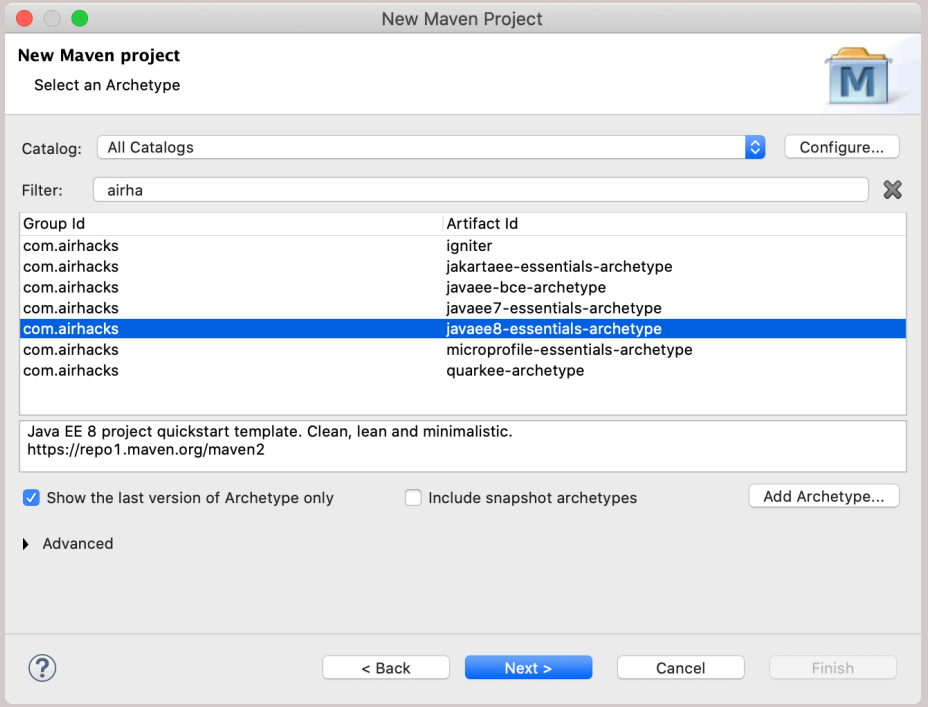
WAS IST JPQL?

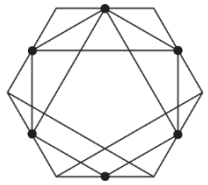
- String-basierte Anfrage Sprache
- "select c from Customer c where c.age > {age}"
- Wird intern in SQL Statements übersetzt

DAS PROJEKT

MAVEN PROJEKT ERSTELLEN

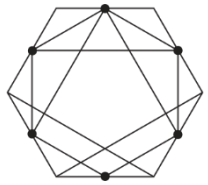
- File ➔ New ➔ Other
- Maven ➔ Maven Projekt
- Archetype: airhacks
- javaee-essential-archtype
- Group Id : com.bmw
- Artifact Id: caro
- Finish





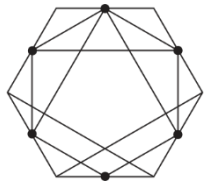
ENTITY ERZEUGEN

```
public class Customer {
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;
        private String name;
        private LocalDate createdDate;
        private boolean active;

        + Setter and Getterns
}
```



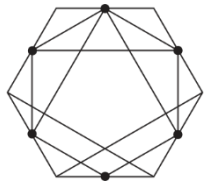
ENTITY ERZEUGEN

```
@Entity
public class Vehicle {
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;
        private String identifier;
        private Color color;
        private EngingeType engineType;
}
```



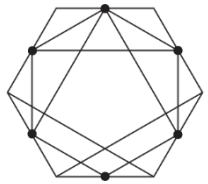
WICHTIGE JPL ANNOTATIONEN I

Annotation	Bedeutung
@Entity	Mappt eine Klasse auf eine Datenbank Tabelle
@Table(name = “tablename”)	Definiert einen Table Name (Standard: Klassename)
@Column(name = spaltenname)	Definiert einen Spaltennamen (Standard: Instanzvariable)
@MappedSuperclass	Annotiert die Superklasse
@Enumerated(EnumType.String)	Enumeration werden als String in Datenbank gespeichert
@Tranient	Variable wird nicht gespeichert
@PrePersist	Lifecycle Callback

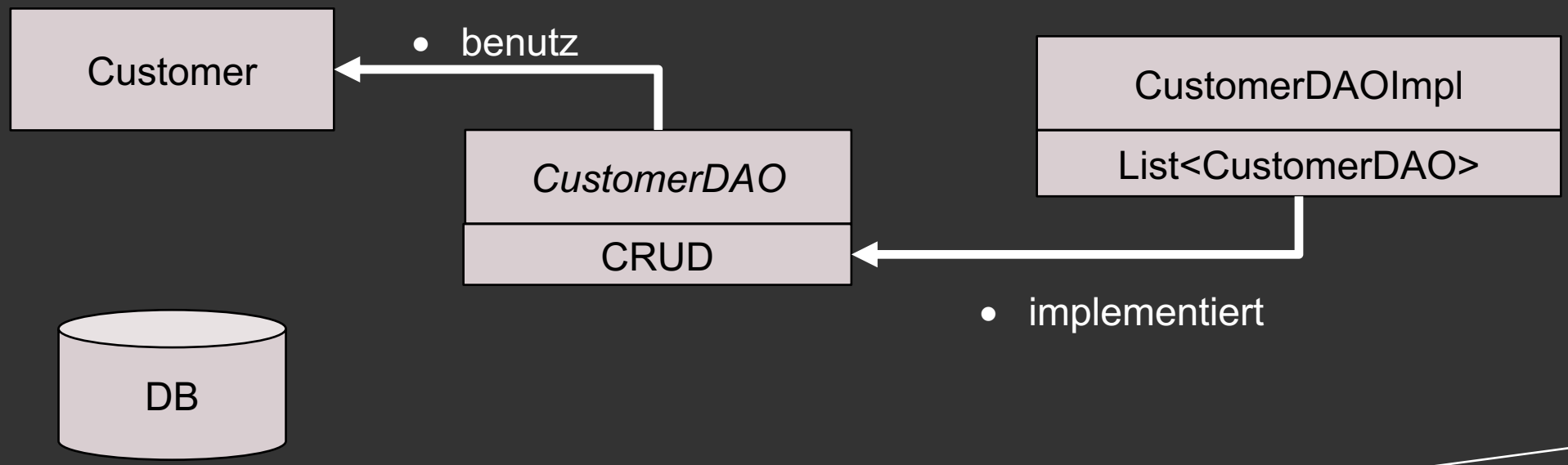


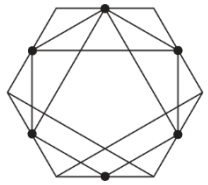
WICHTIGE JPL ANNOTATIONEN II

Annotation	Bedeutung
@OneToONe	1-1 Beziehung
@OneToMany	1-N Beziehung
@ManyToOne	N-1 Beziehung
@ManyToMany	N-M Beziehung
@cascade(CascadeType.)	ALL, PERSIST, REMOVE,…
@mappedBy(“otherKlass_refer”)	Welche Klasse verwaltet die Beziehung? (Fremdschlüssel)
@fetch	EAGER | LAZY
@JointColumn (name = “schluessel”)	Fremdschlüssel Spaltenname



DATA ACCESS OBJECT PATTERN (DAO IN JAVA SE)





PERSISTENCE IN JAVA EE

JPL

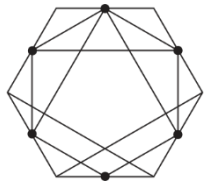
- @entity
- class Customer

EJB

- @stateless
- class CustomerService

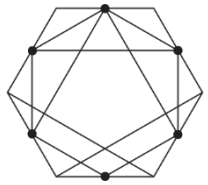
CDI

- class CustomerService
- @PersistenceContext(unitName = “pu”)
- private EntityManager entityManager
- CRUD = create, read, update, delete
- class PersistenceService für alle
- Oder pro class ein PersistenceService



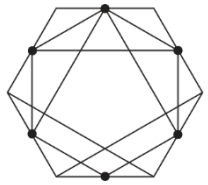
PERSISTENCE XML

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns=http://xmlns.jcp.org/xml/ns/persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

<persistence-unit name=”unit-name" transaction-type="JTA">
<jta-data-source>java:app/caro/datasource</jta-data-source>
<!-- class>com.bmw.entity.customer</class -->
<exclude-unlisted-classes>false</exclude-unlisted-classes>
<properties>
<property
name="javax.persistence.schema-generation.database.action” value="drop-and-create" />
</properties>
</persistence-unit>
</persistence>
```



EJB STATELESS

```
package com.bmw.service;


@Stateless
public class CustomerService {
@PersistenceContext
EntityManager entityManager;
```



STATEFULL VS. STATELESS

Statefull

- ❑ Schneller bei dem Client
- ❑ Brauchen backing Storage
- ❑ Server-side State abhängig

Stateless

- ❑ Einfacher Server Design
- ❑ Merkt keine Details und Sessions
- ❑ Keine Anfragen Abhängigkeiten

CRUD: CREATE

```
public Customer create(Customer customer) {
        entityManager.persist(customer);
      return customer;
}
```

CRUD: READ

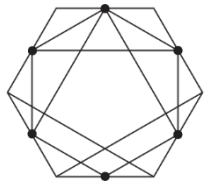
```
public Customer read(long id) {
    return entityManager.find(Customer.class, id);
}
```

CRUD: UPDATE

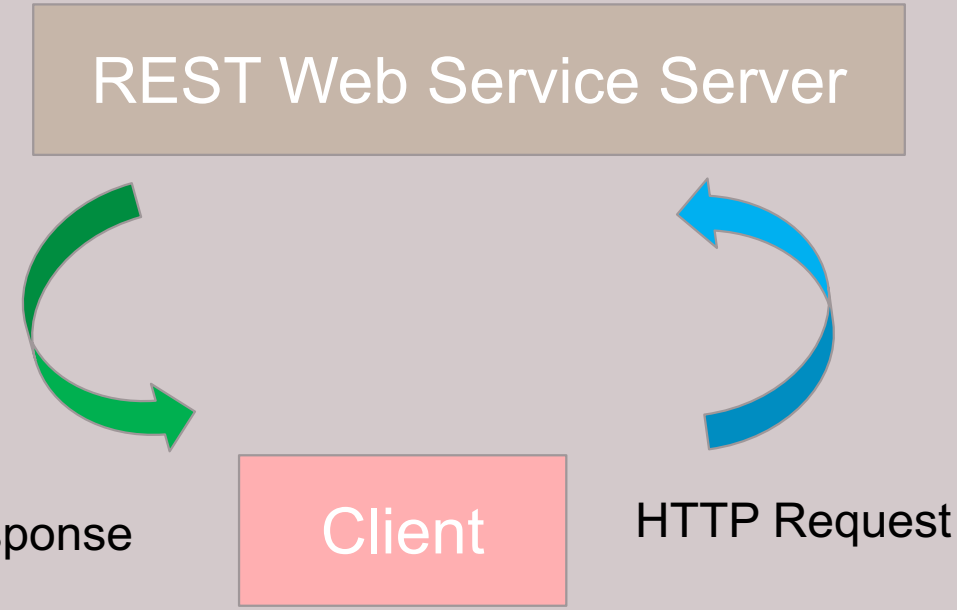
```
public Customer update(Customer customer) {
      entityManager.merge(customer);
   return customer;
}
```

CRUD: DELETE

```
public void delete(Customer customer) {
      entityManager.remove(customer);
}
```

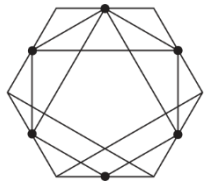


REST ENDPOINT



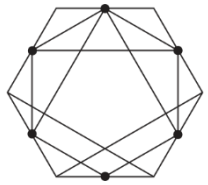
REST ENDPOINT: DEFINIERE DIE REST KLASSE(N)

- `@Path("MyClassRest-Path")`
- Communication MediaType festlegen
- `@Consumes(MediaType.APPLICATION_JSON)`
- `@Produces(MediaType.APPLICATION_JSON)`
- `class CustomerRest {…}`



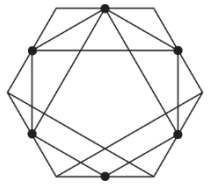
REST ENDPOINT: DEFINIERE DIE REST KLASSE(N)

```
@Path("customer")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)


public class CustomerRest {
@Inject
CustomerService customerService;
```

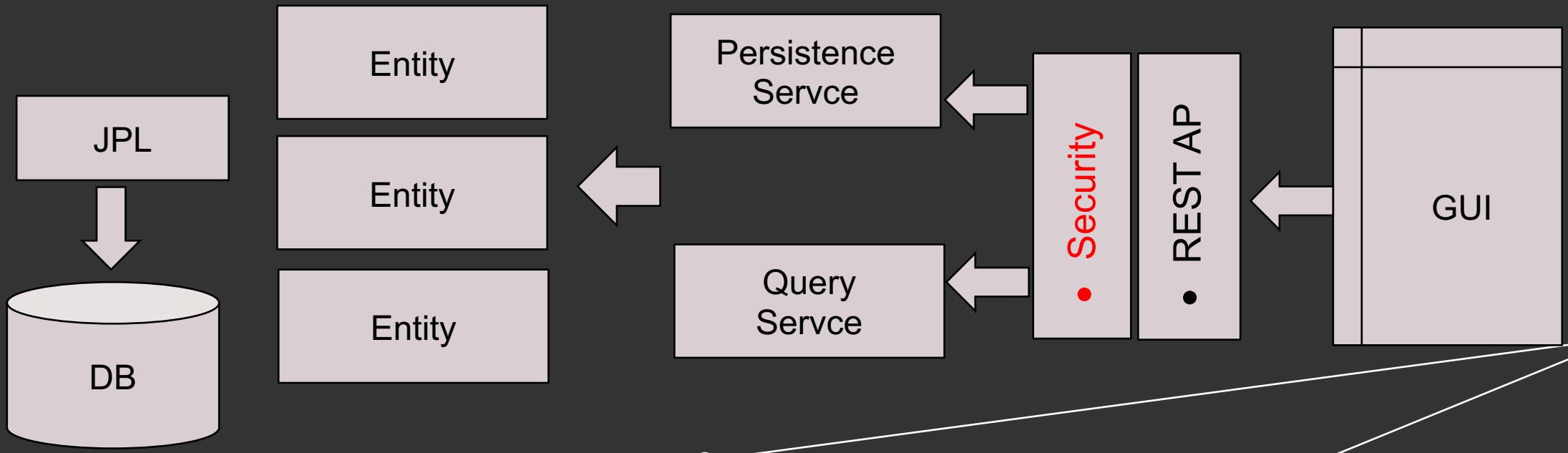


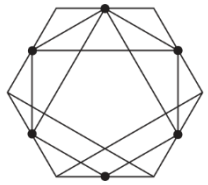
WICHTIGE REST API ANNOTATION

Annotation	Bedeutung
@Path(path-to-the-world)	URL Adresse über die das Dienst erreichbar ist
@POST	Für das erstmalige Speichern: Write-Anfragen
@PUT	Für update: Write-Anfragen
@GET	Für Read-Only Anfragen



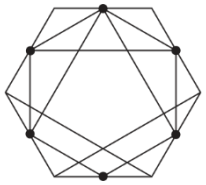
CRUD OPERATION REST API





CUSTOMER-REST CRUD: CREATE

```
@Path("create")
@POST
public Response create(Customer customer) {
      customerService.create(customer);
    return Response.ok(customer).build();
}
```

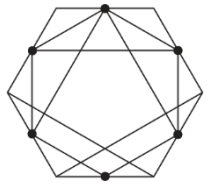


CUSTOMER-REST CRUD: READ

```
@Path(”update")
@PUT
public Response update(Customer customer) {
      customerService.update(customer);
   return Response.ok(customer).build();
}
```

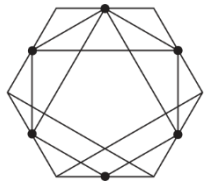
CUSTOMER-REST CRUD: UPDATE

```
@Path(”update")
@PUT
public Response update(Customer customer) {
      customerService.update(customer);
   return Response.ok(customer).build();
}
```



LIFECYCLE CALLBACK

```
@PrePersist
private void init() {
setDateCreated(LocalDate.now());
}
```



QUERY-SERVICE:

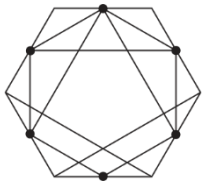

```
@Stateless
public class QueryService {

    @PersistenceContext
     private EntityManager entityManager;
```

QUERY-SERVICE: FIND ALL

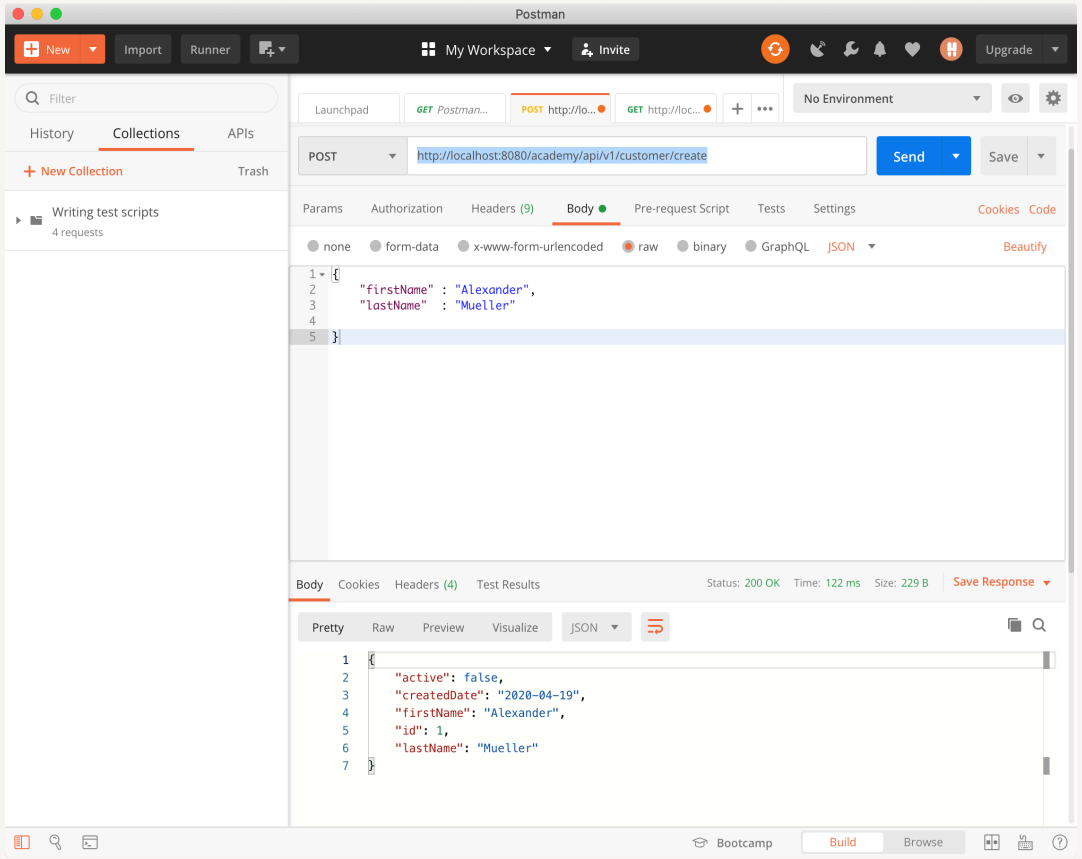

```
public List<Customer> findAll() {
    return entityManager
    .createQuery("Select c from Customer c",  Customer.class)
    .getResultList();
}
```

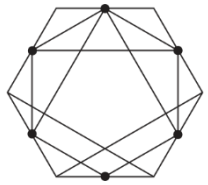
REST API FIND ALL

```
@Path("all")
@GET
public List<Customer> findAll(){
    return customerService.findAll();
}
```



APPLICATION TEST MIT POSTMAN





BEAN VALIDATION

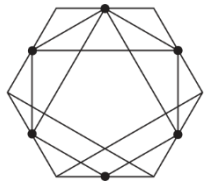
Annotation	Bedeutung
@NotEmpty(message = “msg”)	Field darf nicht leer sein
@NotNull(message = “msg”)	Field darf nicht null Wert haben
@Size(min=int, max=int, message = “msg”)	Min, Max Character
@Pattern(regexp = “regexp”, message = “msg”)	Erlaubte Pattern
@Pattern(regexp = "^[A-Za-z0-9+_.-]+@(.+)$")	Beispiel für Email
@FutureOrPresent(message = “no past date”)	Datum darf nicht in Vergangenheit liegen
@JsonbDateFormat(value = "yyyy-MM-dd")	Datum Format erzwingen

VALIDATION BEISPIEL I

```
@NotEmpty(message = "last name should not be empty!")
@Size(min =3, max = 20, message = ”Min 3, Max 20 ")
private String lastName;
```

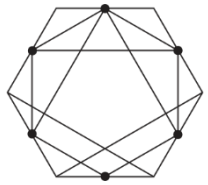
VALIDATION BEISPIEL II

```
@NotEmpty
@Email(message = "Email in Format name@domain.xxx")
@Pattern(regexp = "^[A-Za-z0-9+_.-]+@(.+)$")
private String email;
```



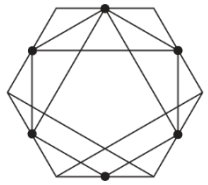
VALIDATION BEISPIEL III

```
@NotNull
@FutureOrPresent(message = "Date should not be in the past")
@JsonbDateFormat(value = "yyyy-MM-dd")
private LocalDate createdDate;
```



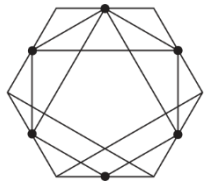
INVALID EXCEPTION MAPPER

```
@Provider
public class ConstraintViolationExceptionMapper implements
ExceptionMapper<ConstraintViolationException> {


    @Override
    public Response toResponse(ConstraintViolationException exception) {
        return Response.status(Response.Status.BAD_REQUEST)
                .header("X-Validation-Error", exception.getMessage())
                .entity(exception.getMessage())
                .build();
    }
```



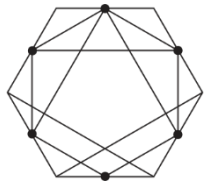
EXCEPTION IN EIGENE KLASSE VERPACKEN

```
public class CustomerCreationException extends RuntimeException{
      public CustomerCreationException(String message) {
      super(message);
}
```



HILFSKLASSE FÜR BESSERE AUSGABE

```
public class RestError {
    private UUID ref;
    private Integer status;
    private String code;
    private String msg;
```

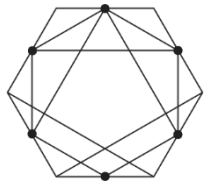


MAPPER MIT EIGENEN KLASSEN

```
@Provider
public class CustomerCreationMapper implements
ExceptionMapper<CustomerCreationException> {
@Override
public Response toResponse(CustomerCreationException e) {
      RestError error = new RestError();
       error.setRef(UUID.randomUUID());
        error.setStatus(405);
        error.setCode("entity.costumer");
        error.setMsg("Invalid Customer Input!");
        returnResponse.status(Response.Status.BAD_REQUEST)
       .entity(error).build();
}
```

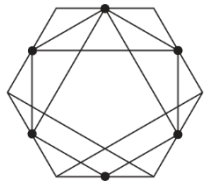
JPQL

- NAMED Query
- Native Query
- Dynamische Query



NAMED QUERY IN ENTITY

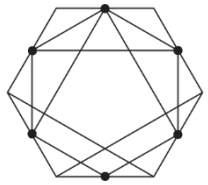
```
@NamedQuery (name = “Name of Query”, query = “JPQL Query Syntax”)
```



NAMED QUERY IN ENTITY (BEISPIEL 1)

```
@NamedQuery(name = Customer.ALL, query = "select c from Customer c order by c.name ")
```

```
@Entity
public class Customer {

    public static final String ALL= "ALL_CUSTOMERS";
```



NAMED QUERY IN QUERY SERVICE (BEISPIEL 1)

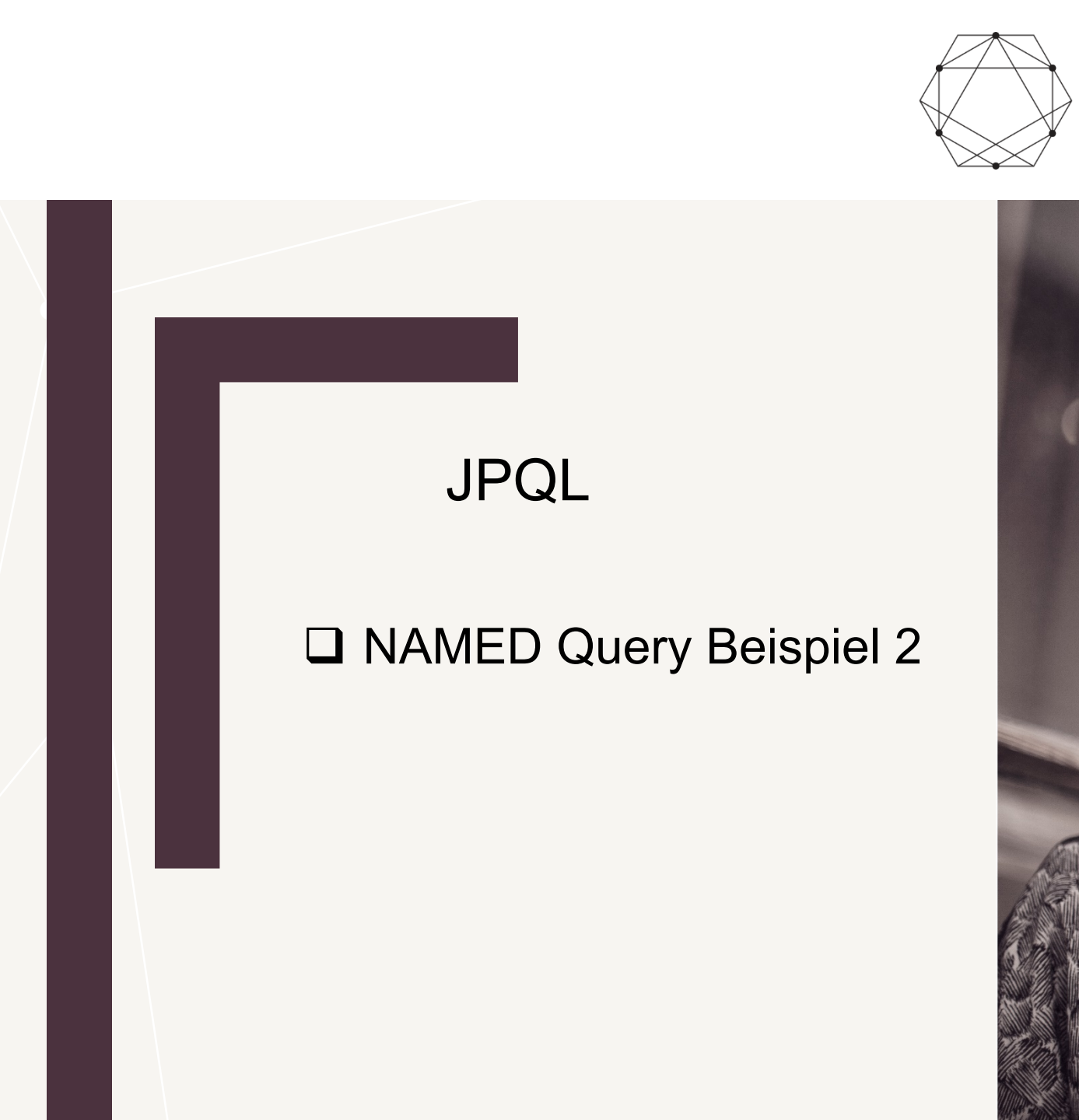
```
@Stateless
public class QueryService {

  @PersistenceContext
  EntityManager entityManager;
```


```
public List<Customer> findAllCustomer(){
return entityManager.createNamedQuery(Customer.ALL, Customer.class)
.getResultList();
}
```

NAMED QUERY IN REST API (BEISPIEL 1)

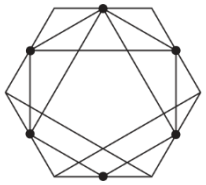
```
public class CustomerRest {
@Inject
QueryService queryService;
```

```
 @GET
    @Path("list")
    public Response listAllTodoUsers() {
    return Response.ok(queryService.findAllCustomer()).build();
    }
```



JPQL

- NAMED Query Beispiel 2

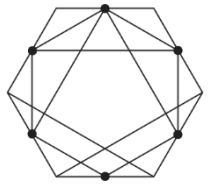


NAMED QUERY IN ENTITY (BEISPIEL 2)


```
@NamedQuery(name = Customer.BY_PATTERN,
query = "select c from Customer c where c.name like :name ")
```

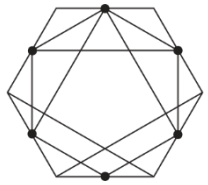
```
@Entity
public class Customer {
```

```
    public static final String BY_PATTERN= ”find.ByPattern";
```



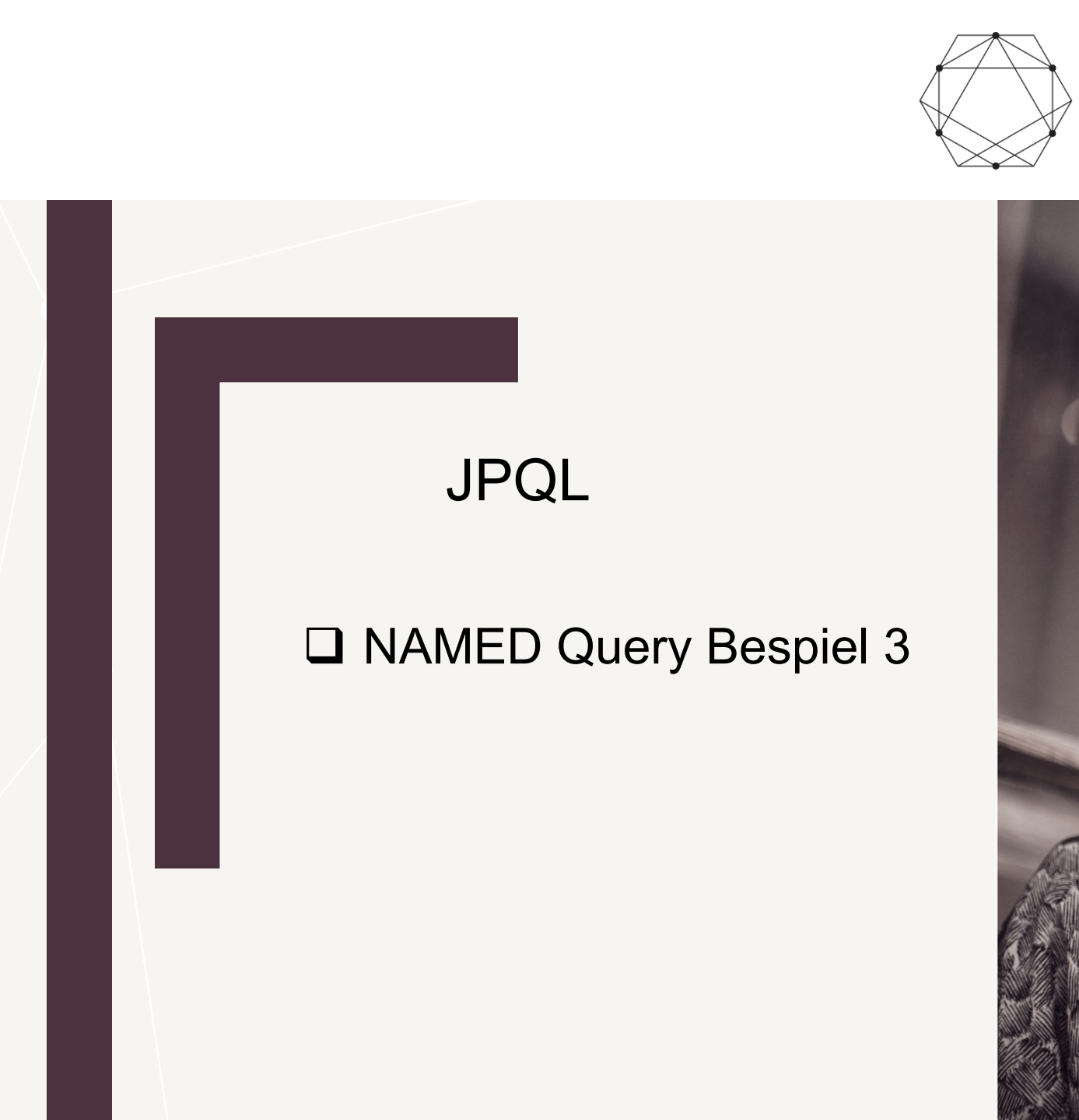
NAMED QUERY IN QUERY SERVICE (BEISPIEL 2)


```
@Stateless
public class QueryService {
  @PersistenceContext
  EntityManager entityManager;


public Collection<Customer> findCustomerByPattern(String name) {
return entityManager
.createNamedQuery(Customer.BY_PATTERN, Customer.class)
.setParameter("name", "%" +name+ "%")
.getResultList();
}
```



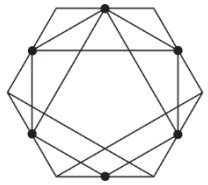
NAMED QUERY IN REST API (BEISPIEL 2)

```
public class CustomerRest {
@Inject
QueryService queryService;


@Path(”pattern")
@GET
public Response findCustomerByPattern(@QueryParam("name") String name) {
Collection<TodoUser> result =  queryService.findCustomerByPattern(name);
return Response.ok(result).build();
}
```



JPQL

- NAMED Query Bespiel 3

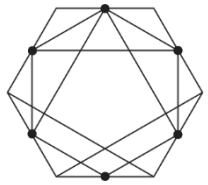


NAMED QUERY IN ENTITY (BEISPIEL 3)


```
@NamedQuery(name = Customer.BY_PATTERN,
query = "select c from Customer c where c.email = :email ")
```

```
@Entity
public class Customer {
```

```
public static final String BY_EMAIL= ”find.ByEmail";
```



NAMED QUERY IN QUERY SERVICE (BEISPIEL 3)


```
@Stateless
public class QueryService {

public TodoUser findCustomerByEmail(String email) {
try {
// NoResultException, NonUniqueResultException, IllegalStateException by delete or update
return entityManager
.createNamedQuery(Customer.BY_EMAIL, Customer.class)
                .setParameter("email", email)
                .getSingleResult();
} catch(NonUniqueResultException | NoResultException exception) {
return null;
} }
```

NAMED QUERY IN REST API (BEISPIEL 3)

```
public class CustomerRest {
@Inject
QueryService queryService;


@Path("find/{email}")  //  Dynamic Segment: url/find/email
@GET
// public TodoUser findTodoUserByEmail(@NotNull @PathParam("email") @DefaultValue("")
String email) {
public Customer findCustomerByEmail(@PathParam("email") @DefaultValue("") String email) {
return queryService.findCustomerByEmail(email);
}
```

PROBLEM

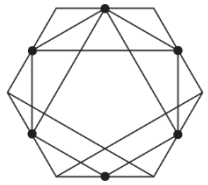
- DUPLIKATE !!

CUSTOMER-REST CRUD: CREATE

- ❑ Unser Lösung erlaubt Duplikate!
- ❑ Lösung: Email eindeutig!
- ❑ Anzahl gespeicherte Emails!

JPQL

- Dynamic Query
- Native Query

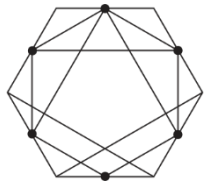


NATIVE QUERY IN QUERY SERVICE (BEISPIEL 4)


```
@Stateless
public class QueryService {


public List countCustomerByEmail(String email) {
        List resultList =
        entityManager
       .createNativeQuery("select count (id) from CustomerTable
              where exists (select id from CustomerTable where email = ?)")
                .setParameter(1, email).getResultList();
        return resultList;
    }
```

NAMED QUERY IN REST API (BEISPIEL 4)

```
public class CustomerRest {
@Inject
QueryService queryService;


 @GET
 @Path("count")
  public Response countTodoUserByEmail(@QueryParam("email”)String email) {
    return Response.ok(queryService.countTodoUserByEmail(email)).build();
  }
```

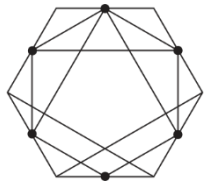


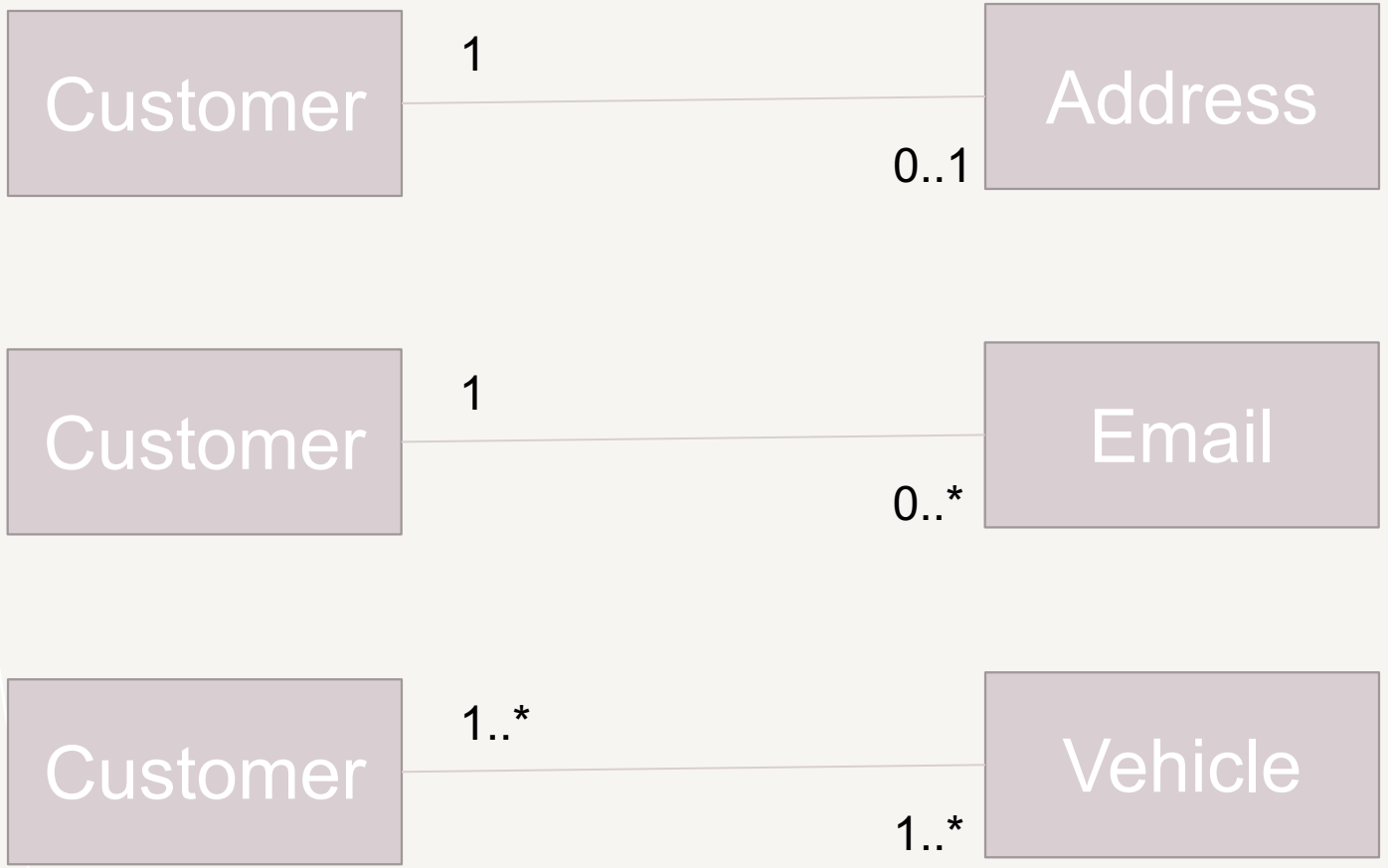
PERSISTENCE SERVICE ANPASSEN!

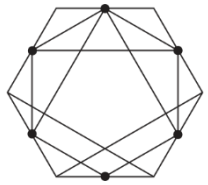
```
public Customer create(Customer customer) {


List list = queryService.countCustomerrByEmail(customer.getEmail());
Integer count = (Integer) list.get(0);


if (customer.getId() == null && count == 0) {
      entityManager.persist(customer);
}
return todoUser;
}
```

JPL BEZIEHUNGEN

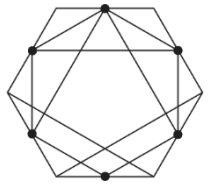
- OneToOne
- OneToMany
- ManyToOne
- ManyToMany





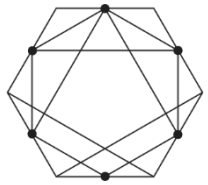


```
public class Customer {

        private Address address              // one-to-one

        private Set<Email> emails;           // one-to-many


Customer customer = new Customer();
customer.getAddress().getCity();             // navigieren
```

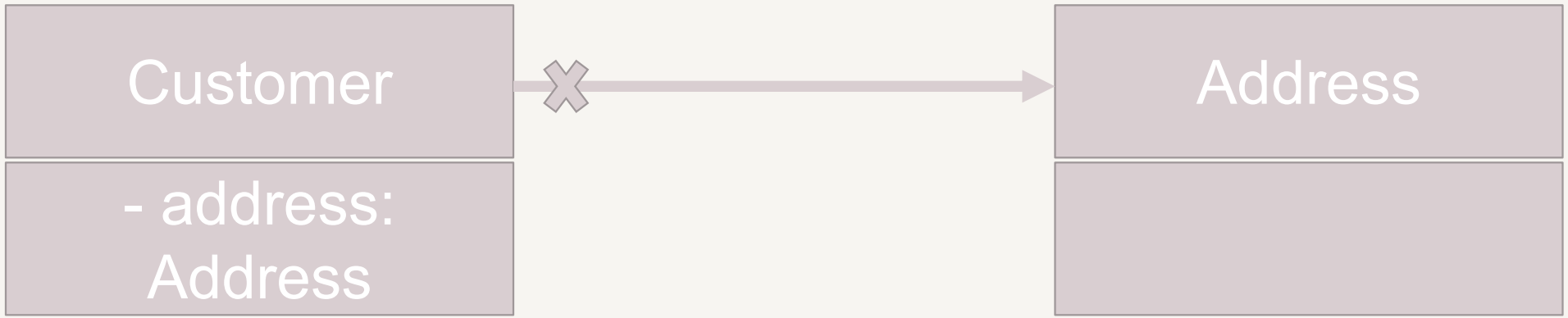


ERSTER VERSUCH

```
public class CalculatorTest1 {

    public static void main(String[] args) {

      Calculator calculator = new Calculator();              // setup

      double actualValue = calculator.add(10.5, 5.5);        // action
      double expectedValue = 16;

      if(expectedValue != actualValue)                       // verify
          System.out.println("Expected: " + expectedValue
            + " but found is: " + actualValue);

    }
}
```

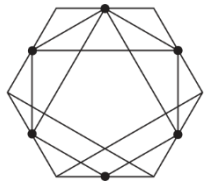


UNIDIREKTIONAL BEZIEHUNG

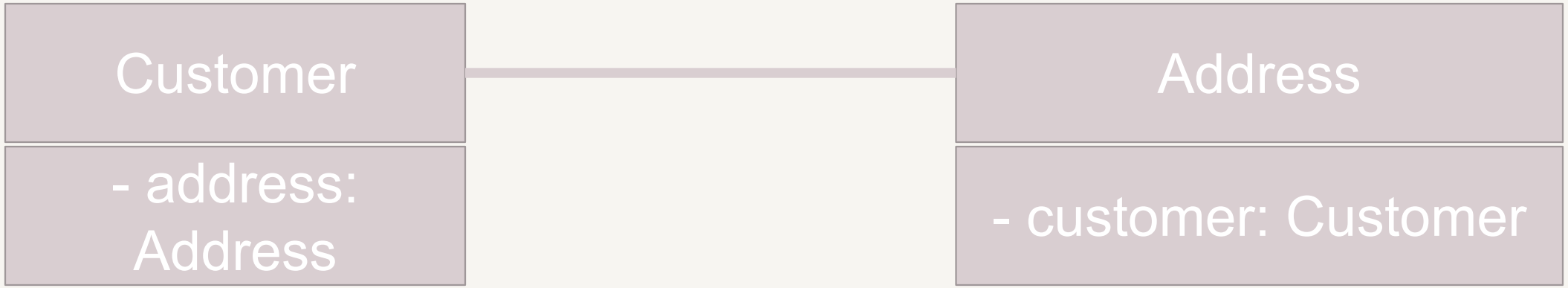


In Java: von links nach rechts navigierbar: Customer => Address

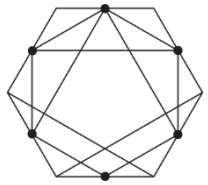
In Datenbank: von beiden Seiten navigierbar: Person ⇔ Address



BIDIREKTIONAL BEZIEHUNG



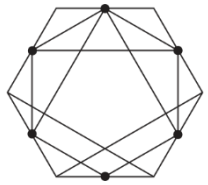
In JAVA und in DB in beiden Seiten navigierbar!



MAPPING VON 1:1 BEZIEHUNG


```
@Entity
public class Customer {
  ..
@OneToOne
(mappedBy = "customer",
cascade=CascadeType.ALL)
private Address address;
```

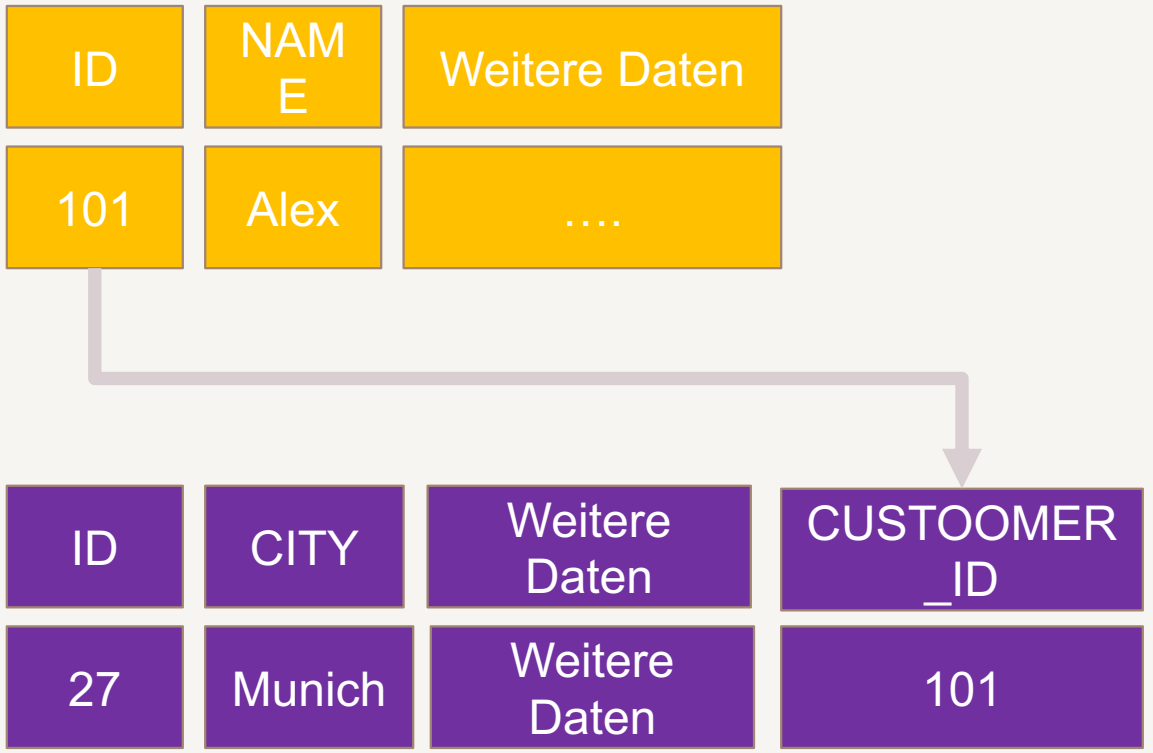
```
@Entity
public class Address {
..

@OneToOne
private Customer customer;
```

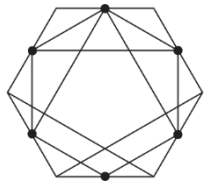


MAPPING VON 1:1 BEZIEHUNG TABELLEN

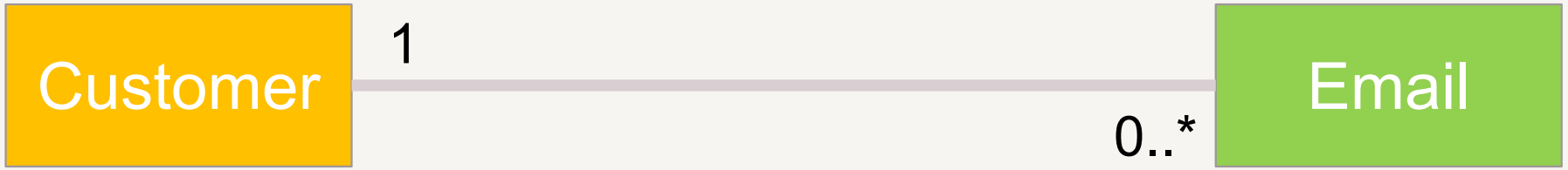
```
public class Customer {

@OneToOne(mappedBy = ”customer")
private Address address;
```

```
@Entity
public class Address {

@OneToOne
private Customer customer;
```



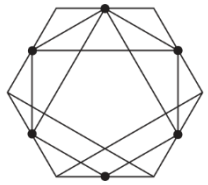


MAPPING BIDIREKTIONALE VON 1:N BEZIEHUNG




```
@Entity
public class Customer {
@OneToMany(mappedBy="customer")
private Set<Email> emails;
```

```
@Entity
public class Email {
@ManyToOne
private Customer customer;
```



1-N BIDIREKTIONALES MAPPING TABELLEN

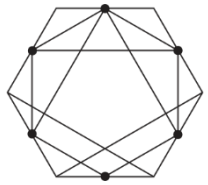
```
@Entity
public class Customer {
@OneToMany
private Set<Email> emails;
```

@ManyToOne verfügt über kein mappedBy

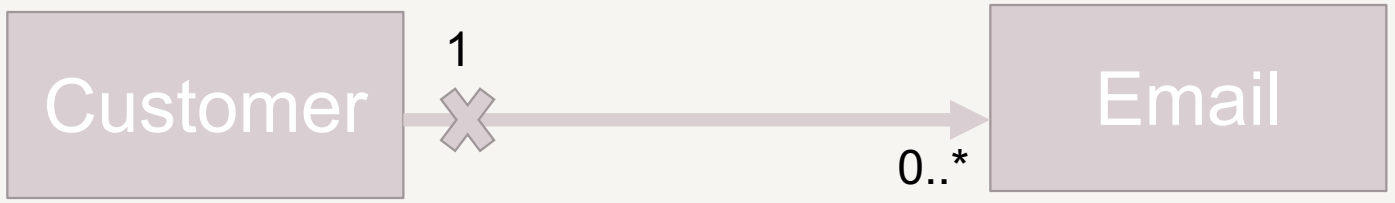
```
public class Email {
@ManyToOne
private Customer customer;
}
```

ID	NAME	Weitere Daten
101	Alex	....

ID	EMAIL	CUSTOMER_ID
47	alex@m.de	101
57	*a.x*@m.de	101



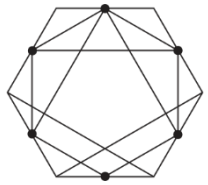
UNIDIREKTIONALES MAPPING VON 1:N BEZIEHUNG



1-N Beziehung

```
@Entity
public class Customer {

@OneToMany
private Set<Email> emails;
```

Gleiche Attribute wie bei @OneToOne

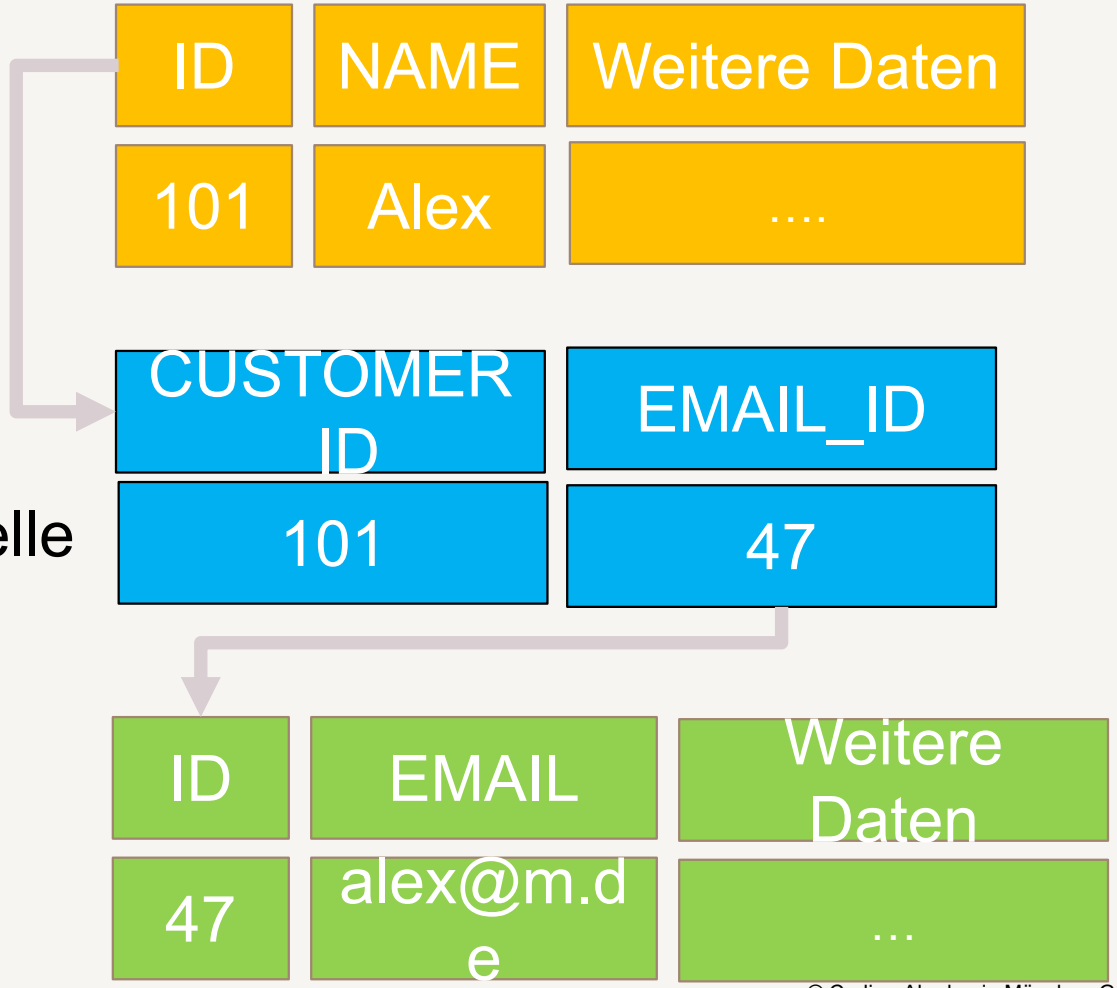


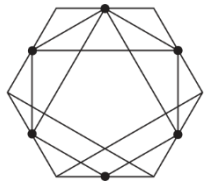
1-N UNIDIREKTIONALES MAPPING TABELLEN

```
@Entity
public class Customer {

@OneToMany
private Set<Email> emails;
```

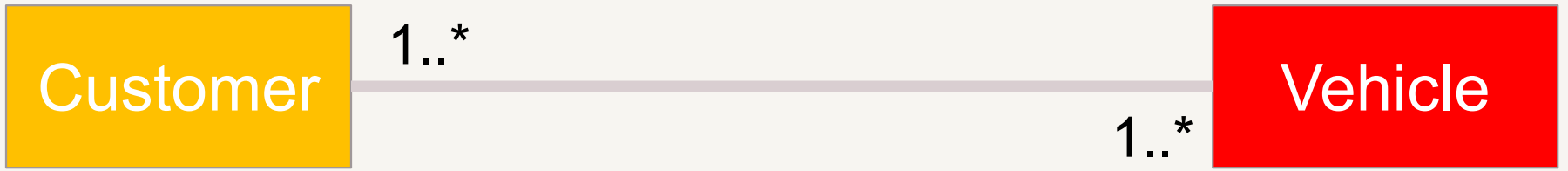
```
public class Email {

        private String email;
}
```

JOINED Tabelle



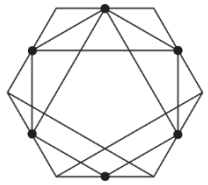


MAPPING BIDIREKTIONALES N:M BEZIEHUNG




```
@Entity
public class Customer {
@ManyToMany
private Set<Vehicle> vehicles;
```

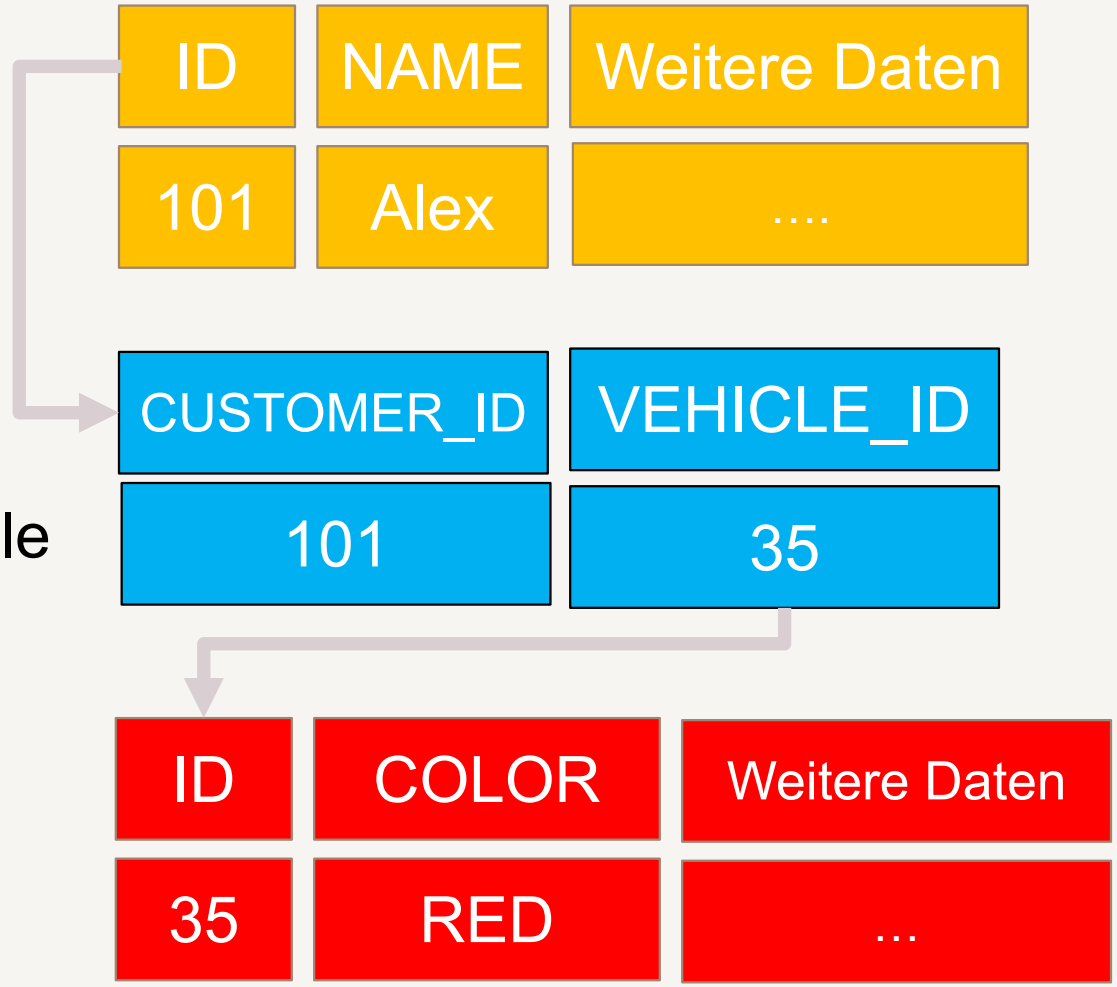
```
@Entity
public class Vehicle {
@ManyToMany(mappedBy=”vehicles")
Private Set<Customer> customers;
```

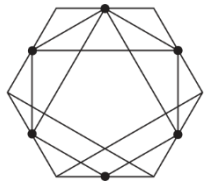


N-M MAPPING TABELLEN

```
@Entity
public class Customer {

@ManyToMany
private Set<Vehicle> vehicles;
```

JOINED Tabelle

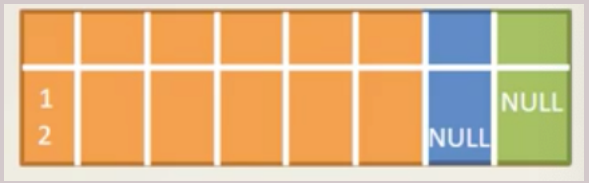




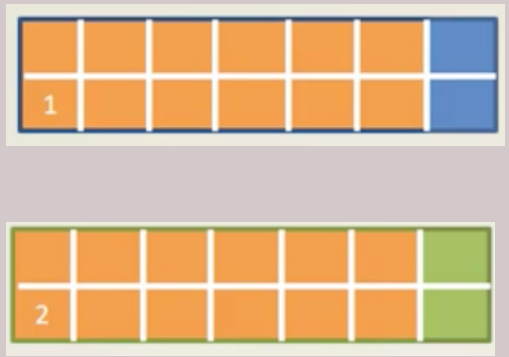
VEREBUNGSHIERARCHIE IN JPA

```
Customer extends Person
Employee extends Person
```

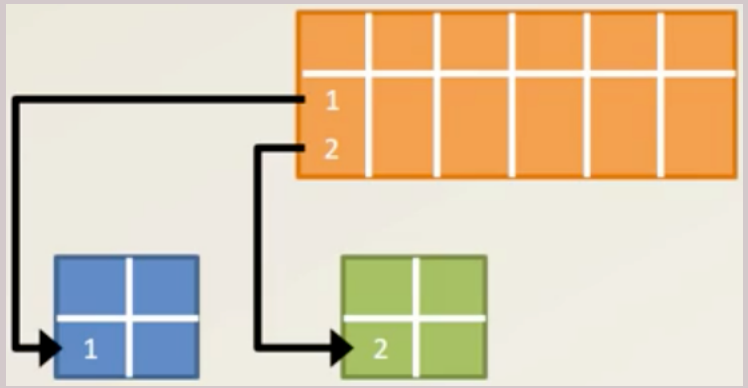
SINGLE_TABLE



TABLE_PER_CLASS



JOINED

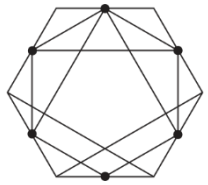


AUFGABE

- class Address
- class
- ManyToOne
- ManyToMany

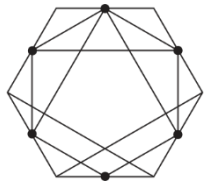
PROBLEM

- ❑ Für Logged-in User
- ❑ Seine Vehicle suchen!
- ❑ Lösung: Session



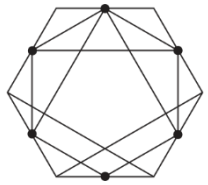
MERKE DEN CUSTOMER DURCH SESSION

```
@SessionScoped
public class UserSession implements Serializable{
private static final long serialVersionUID = 1L;
private String email;

public String getEmail() {
return email;
}

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



DIE KLASSE VEHICLE

```
@Entity
@NamedQuery(name = Vehicle.ALL_BY_OWNER_EMAIL,
query = "select v from Vehicle v where v.customer.email = :email")
public class Vehicle {


public static final String  ALL_BY_OWNER_EMAIL = "Find.CustomerVehicle";

@ManyToOne
@JoinColumn(name = ”Customer_Id")
private Customer customer;

}
```



IN QUERY SERVICE


```
@Stateless
public class QueryService {

@Inject
CustomerSession customerSession;


public Collection<Vehicle> findUserVehicles(){
return entityManager
.createNamedQuery(Vehicle.ALL_BY_OWNER_EMAIL, Vehicle.class )
.setParameter("email",  customerSession.getEmail())
.getResultList();
}
```

IN REST API

```
@GET
@Path(”vehicles")
public Response listCustomerVehicles() {
return Response.ok(queryService.findUserVehicles()).build();
}
```

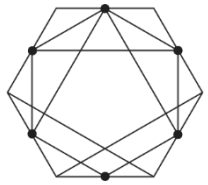
https://www.sqlite.org/download.html

SQLITE : DATENBANK ERZEUGEN

```
Terminal => Navierge zu SQLTE Ordner

sql> sqlite3 databasename.db
sql> sqlite3 caro.db
sql> .schema
sql> .exit
```

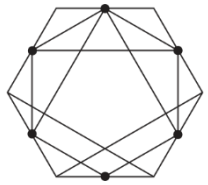
POM XML FÜR SQLITE ANPASSEN

```
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.30.1</version>
</dependency>
```



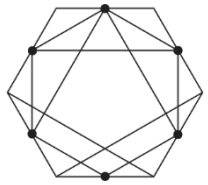
DATASOURCE DEFINITION

```
@DataSourceDefinition(
    name = "java:app/caro/datasource",
    className = "org.sqlite.SQLiteDataSource",
    url = "jdbc:sqlite:C:/Users/Laith/BMW/sql/todo.db”,  // Windows
    url = "jdbc:sqlite:/Users/laithraed/sql/todo.db”      //Mac
        )


@Stateless
public class PersistenceService {
```



DATASOURCE NAME IN PERSISTENCE.XML

```
<persistence-unit name="myPU" transaction-type="JTA">
        <jta-data-source>java:app/caro/datasource</jta-data-source>
```



WAS NOCH FEHLT?

- ❑ GUI
- ❑ SELENIUM
- ❑ SECURITY

EINE BITTE AN EUCH

- ❑ Sind diese Live Coding sinssvoll?
- ❑ Würden Ihr das für weitere Gruppen empfehlen?
- ❑ Soll ein Teil des Kurses so aufgebaut werden?
- ❑ Praktische Folien: Gemeinsam Coden, Übungen

- ❑ Eure Verbesserungsvorschläge bitte an Oliver
- ❑ Eure Verbesserungsvorschläge an laith@gmx.de

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT.