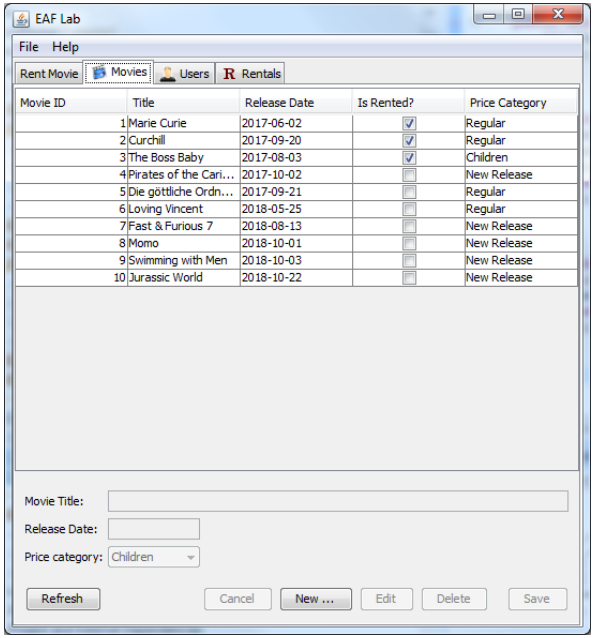
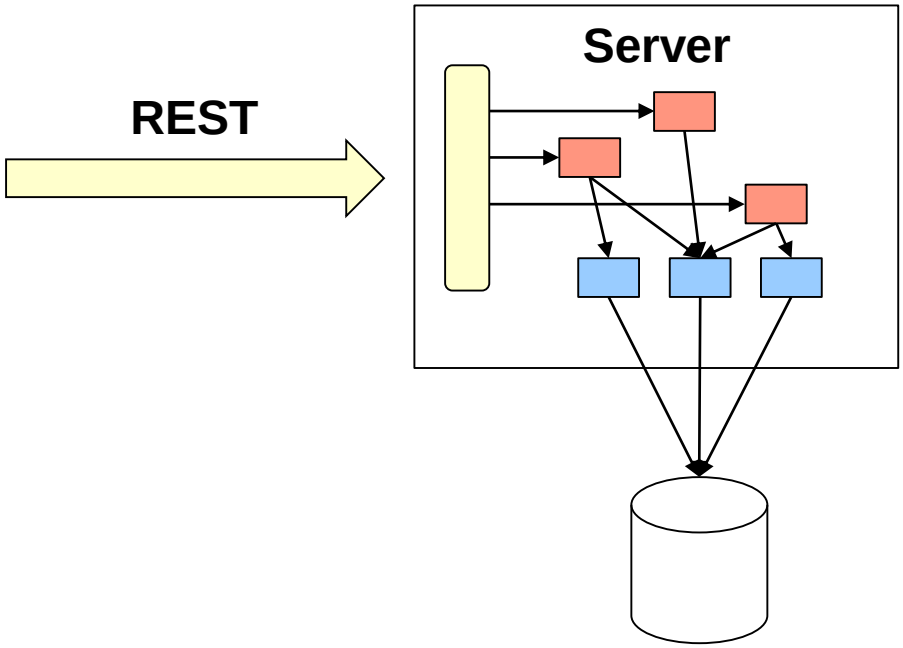
Java Persistence API

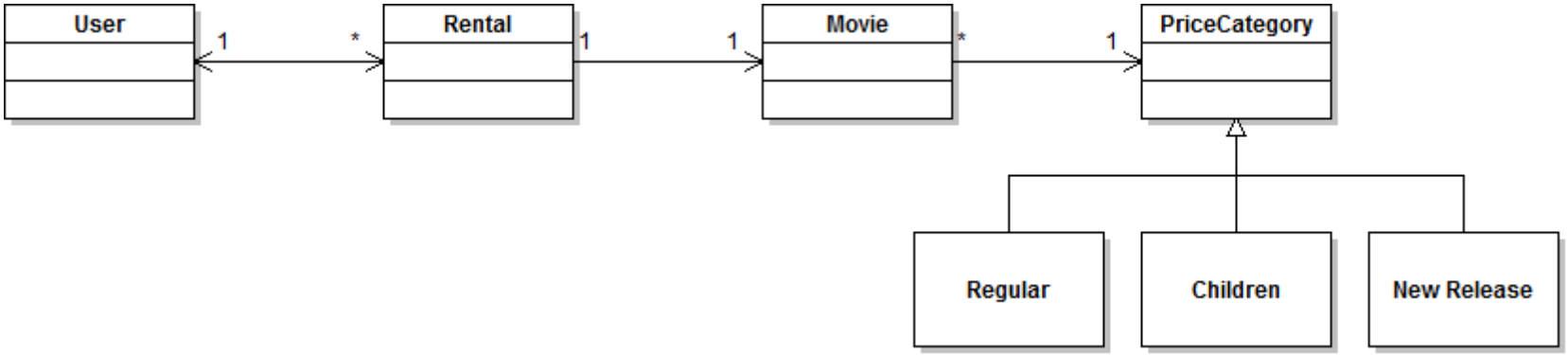
- **Persistence Design Patterns**
  - Data Access Object Pattern
  - Service Façade





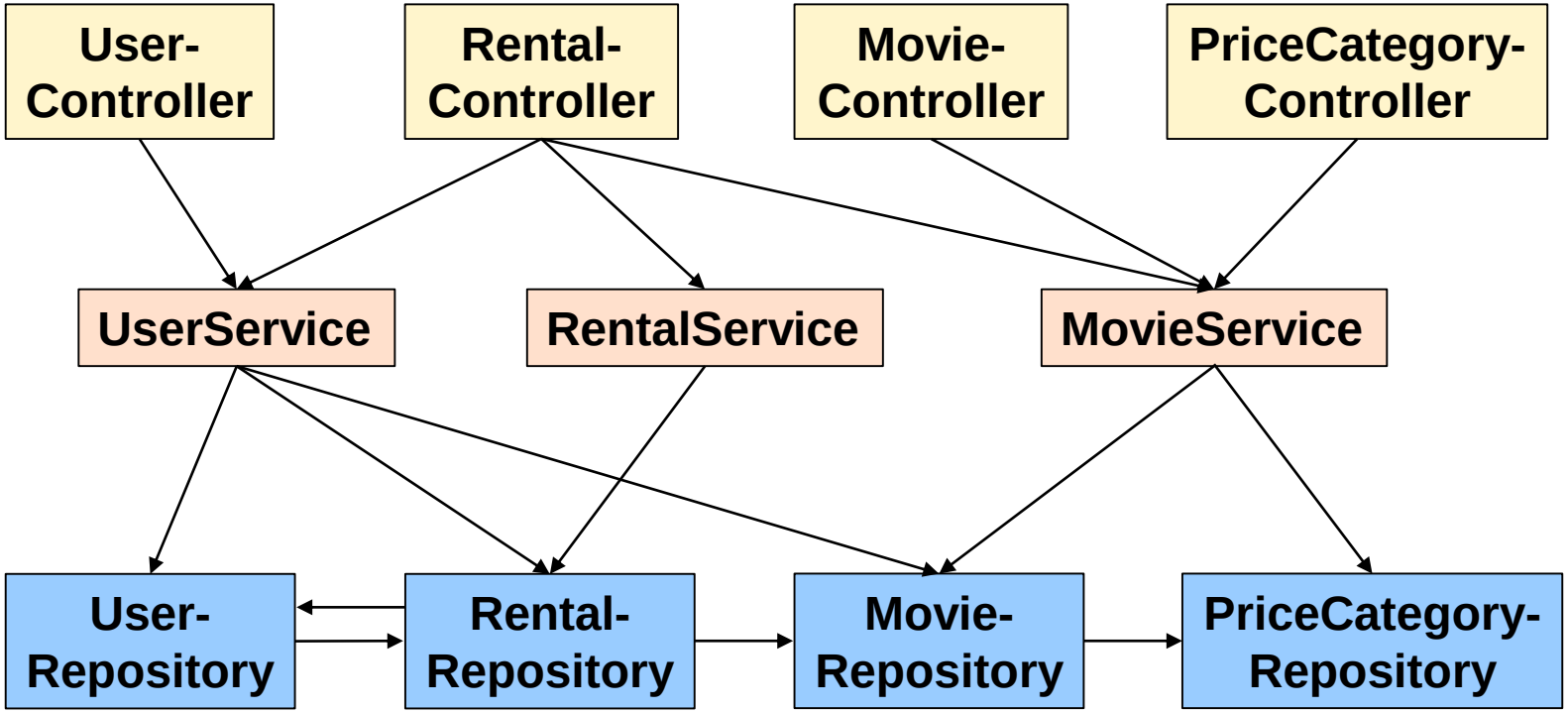
Movie Rental Application

- **Model Classes**



Movie Rental Application

- **Component Wiring**



Service Layer

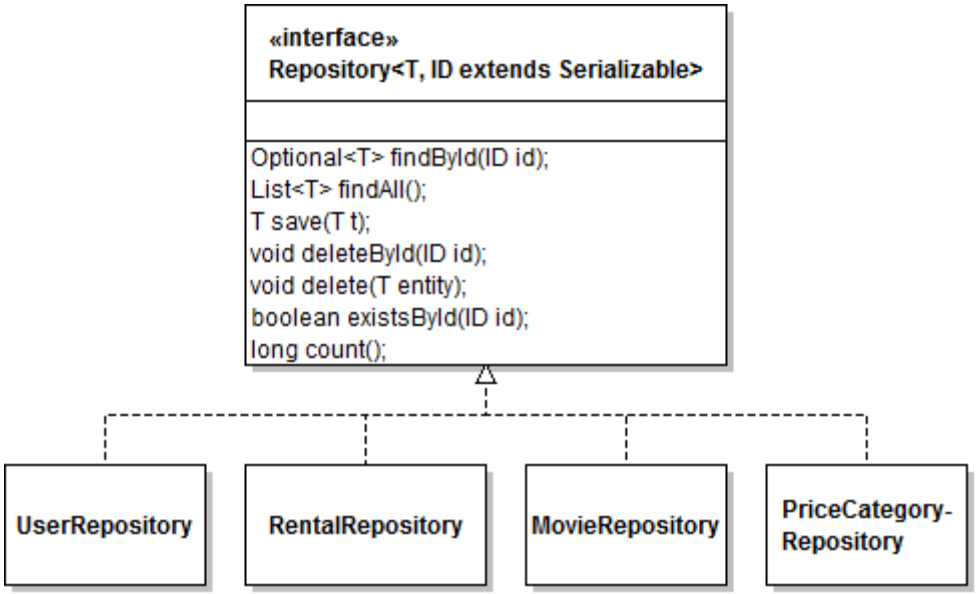
- **Tasks of the Service Layer**
  - Core API through which other layers of the application will interface => Façade Pattern
  - Defines the core business logic, usually calling on one or more DAO methods to achieve this goal
    - Examples:
      - a movie which is returned may be reserved
      - a user which is deleted can have open rentals
    - Some service methods may simply call the corresponding DAO method
  - Combines methods defined in the DAOs and assembles them to cohesive business methods that define an atomic unit of work
    - Typically transactional semantics are applied to service methods
    - Spring AOP is used to define transactions

DAO Pattern

- **Data Access Object Pattern (Repository Pattern)**
  - All database access in the system is made through a DAO to achieve encapsulation
    - DAO is also called Repository
  - Each DAO instance is responsible for one primary domain object or entity
  - The DAO is responsible for creations, reads (by primary key), updates, and deletions (CRUD) on the domain object
  - The DAO may allow queries based on criteria other than the primary key (typically *finder methods*) returning collections of the domain object
  - The DAO is *not* responsible for handling transactions, sessions, or connections - these are handled outside the DAO to achieve flexibility

Movie Rental Application

- **Repository Interfaces**



DAO Pattern

- **Generic DAO / Repository Interface**

```
public interface Repository<T, ID extends Serializable> {
   Optional<T> findById(ID id);
   List<T>     findAll();

   T           save(T t);

   void        deleteById(ID id);
   void        delete(T entity);

   boolean     existsById(ID id);
   long        count();
}
```

  - Save is used to create and update entries
    - Result is the saved/updated instance

DAO Pattern

- **Separation of Concerns**
  - Business & workflow logic
  - Persistence issues

Business logic should *not* depend on chosen persistence technology

- **Strategy Pattern for Data Access**
  - Not 100% possible
  - DAO type influences its use
    - JDBC-based repositories contains explicit update operations, whereas managed persistence repositories provides object lifecycles, as e.g. JPA, i.e. objects are automatically saved
    - Managed persistence repositories typically allow to define cascade operations on dependent objects, JDBC-based repositories typically not.

DAO Pattern

- **Motivation for the DAO Pattern**
  - Encapsulation
  - Testability
    - DAO is easier to mock for testing than a hibernate entity manager
  - Vendor independence
    - DAO abstracts from different implementations (Hibernate / OpenJPA) of the same DAO Type (JDBC vs. managed)
  - Code generation
    - Repository could be created (e.g. using reflection) by frameworks
- **When won't you use the DAO Pattern**
  - When the application's business logic consists of data access operations and not much else, it makes sense to include the data access code directly in the business operations.
  - JPA Entity Manager or Spring Repositories (Spring Data) **are** DAOs

Java Persistence API

- Persistence Design Patterns
- JPA Overview
- JPA Entity Annotations
- JPA Entity Manager
- Spring/SpringBoot & JPA

Java Persistence API

- **Management of Persistence and Object/Relational Mapping**
  - Incorporates contributions from different technologies (communities)
    - Hibernate / TopLink (Oracle) / JDO

- **Standard**
  - JPA 1.0: JSR 220 EJB 3.0 11.05.06 Final Release
  - JPA 2.0: JSR 317 Java Persistence 2.0 10.12.09 Final Release
  - JPA 2.1: JSR 338 Java Persistence 2.1 22.04.13 Final Release
  - JPA 2.2: JSR 338 Java Persistence 2.2 04.08.17 Maintenance Rel.
  - Jakarta Persistence 2.2 Sept 2019 (no changes)
    https://github.com/eclipse-ee4j/jpa-api
  - Jakarta Persistence 3.0 Sept 2020
    https://jakarta.ee/specifications/persistence/3.0/jakarta-persistence-spec-3.0.pdf
    - `javax.persistence` => `jakarta.persistence`

JPA Implementations

- **EclipseLink JPA 3.0.2 [JPA 3.0]**
  - http://www.eclipse.org/eclipselink/
  - Reference Implementation
  - Main Sponsor: Oracle
- **Hibernate 5.5.7 [JPA 3.0]**
  - http://www.hibernate.org/
  - Main Sponsor: Red Hat
- **Apache OpenJPA 3.1.2 [JPA 2.2]**
  - http://openjpa.apache.org/
  - Main Sponsor: IBM

We will work in the module eaf with JPA 2.2, i.e. we use classes from the package `javax.persistence`.

JPA Features

- **Features of JPA**
  - JPA allows the developer to work directly with objects rather than with SQL statements
  - JPA keeps entities in a cache (Persistence Context)
    - Changes on entities are automatically persisted when the transaction is committed
    - No problem with cyclic dependencies
    - No need to implement `equals` and `hashCode`
  - JPA supports inheritance relationships and supports polymorphic queries
  - JPA manages relations
    - Supports automatic loading of dependent objects
    - Supports cascaded delete
    - Supports bidirectional associations

JPA Ingredients

- **Entity Definitions**
  - Metadata configuration
    - Annotations (or XML files)
  - Configuration by Exception

- **Entity Operations**
  - Entity-Manager provides access to the objects *(similar to a Repository)*
    - Query API: `find / persist / update / remove`
    - JPQL: `SELECT m FROM Movie m WHERE m.title = :title`
  - Persistence context & controlled lifecycle

JPA Entity

```
@Entity
public class Movie {
   @Id @GeneratedValue
   private Long id;

   private String title;
   private boolean rented;
   private LocalDate releaseDate;

   protected Movie() { }
   public Movie(String title, LocalDate releaseDate) { ... }

   public String getTitle() { return title; }

   public boolean isRented() { return rented; }
   public void setRented(boolean rented) { this.rented = rented; }
   ...
}
```

MOVIE

ID	TITLE	RENTED	RELEASEDATE

JPA Entity Manager

```
@Service
@Transactional
public class MovieService {

   @PersistenceContext
   private EntityManager em;

   public Long saveNewMovie(String title, LocalDate date) {
      Movie m = new Movie(title, date);
      em.persist(m);
      return m.getId();
   }

   public void rentMovie(Long id) {
      Movie m = em.find(Movie.class, id);
      m.setRented(true); // will be persisted at the end of the TX
   }
}
```

Created by an `Entity-ManagerFactory` or injected

Java Persistence API

- **Persistence Design Patterns**
- **JPA Overview**
- **JPA Entity Annotations**
- **JPA Entity Manager**
- **Spring/SpringBoot & JPA**

Entity Class: Requirements

- **Entity Class**
  - Entity class must be annotated with the **`@Entity`** annotation

- **Constraints**
  - Entity class must have a public or protected parameterless constructor
    - Additional constructors may be declared
  - Entity class must not be final
  - Persistent instance fields must be non-final
  - Class must be a top-level class, i.e.
    - not a non-static inner class
    - not an interface
    - not an `enum` class
    - not a record

Persistent Fields: Supported Types

- **Persistent Fields: Supported Types**
  - Primitive types
    - `char`, `short`, `int`, `long`, `byte`, `float`, `double`, `boolean`
  - Serializable Types
    - Strings: *`java.lang.String`*
    - Primitive Wrappers: *`Integer`*, *`Short`*, *`Long`*, *`Boolean`*, *`Double, ...`*
    - Big Numericals: *`java.math.BigInteger`*, *`java.math.BigDecimal`*
    - Java temporal types: *`java.util.Date`*, *`java.util.Calendar`*
    - JDBC temporal types: *`java.sql.Date`*, *`Time`*, *`TimeStamp`*
    - Java8 temporal types: *`java.time.LocalDate`*, *`LocalTime`*, *`LocalDateTime`*
    - Byte- and character arrays: *`byte[], char[], Byte[], Character[]`*
    - User-defined serializable types (=> DB content is serialized object)
  - Enums
  - Entity types & Collections of entity types
    - Collection, Set, List, Map

@Entity

- **Specifies that the class is an entity**

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Entity {
   String name() default "";
}
```

  - *name* annotation element
    - This name is used to refer to the entity in queries
    - Must not be a reserved literal in JP-QL (e.g. `ORDER`)
    - Default: *unqualified name of the entity class*

@Table

- **Specification of primary database table used**

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Table {
   String name() default "";
   String catalog() default "";
   String schema() default "";
   UniqueConstraint[] uniqueConstaints() default {};
}
```

- *name* name of the table *unqualified name of the entity class*
- *catalog* catalog of the table *standard catalog*
- *schema* schema of the table *standard schema*
- *uniqueConstraints* constraints applied to generated DDL tables

```
@Table(name = "DEMO_USER", uniqueConstraints = {
   @UniqueConstraint(columnNames = { "PHONENR", "AREA-CODE" })
})
```

@Column

- **Specification of mapped column for a persistent property**

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
   String name() default "";                // name of column
   boolean unique() default false;          // if DB column is unique
   boolean nullable() default true;         // if DB column is nullable
   boolean insertable() default true;       // if manipulation with a
                                            // sql insert is allowed
   boolean updatable() default true;
   String columnDefinition() default "";    // e.g. CLOB / BLOB
   String table() default "";               // table in which field
                                            // is stored (sec. table)
   int length() default 255;                // size for strings
   int precision() default 0;               // decimal precision
   int scale() default 0;                   // decimal scale
}
```

@Enumerated

- **Specification of mapping of enumerations**

```
@Target({METHOD,FIELD}) @Retention(RUNTIME)
public @interface Enumerated {
   EnumType value() default ORDINAL;
}
```

  - Value field
    - `EnumType.ORDINAL`
      - Value is stored as an integer
    - `EnumType.STRING`
      - Value is stored as a string

@Id

- **Specification of primary key field**

```
@Target({METHOD,FIELD}) @Retention(RUNTIME)
public @interface Id {
}
```

  - Possible types for primary keys
    - Primitive types: *byte, int, short, long, char*
    - Wrapper types: *Byte, Integer, Short, Long, Character*
    - Arrays of primitives or wrappers
    - Strings: *java.lang.String*
    - Large numerics: *java.math.BigInteger*
    - Temporal types: *java.util.Date, java.sql.Date, java.time.LocalDate*
    - UUID: *java.util.UUID* (supported by Hibernate)

Primary Keys: Generation

- **Assigned**
  - Primary keys may be assigned by the application, i.e. no key generation is necessary
    - E.g. language table: primary key is the ISO country code
    - UUID (global unique identifier)
- **Identity**
  - Auto increment supported by some DBs
- **Sequence**
  - Some DBs support sequences which generate unique values (e.g. Oracle, PostgreSQL)
- **Table**
  - Primary keys are stored in a separate PK table

@GeneratedValue Annotation

- **Specification of primary key generation method**

```
@Target({METHOD,FIELD}) @Retention(RUNTIME)
public @interface GeneratedValue {
   GenerationType strategy() default AUTO;
   String generator() default "";
}

public enum GenerationType {TABLE, SEQUENCE, IDENTITY, AUTO};
```

  - *strategy* primary key generation strategy
  - *generator* name of the generator as specified in SequenceGenerator or TableGenerator annotations

Annotations Example

```
@Entity
@Table(name="EMP")
public class Employee {
   public enum Type {FULL, PART_TIME};

   protected Employee() { }
   public Employee(String name, Type type) {
      this.name = name; this.type = type;
   }

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

   @Enumerated(EnumType.STRING)
   @Column(name = "EMP_TYPE", nullable = false)
   Type type;

   String name;
}
```

Annotations Example

```
CREATE SEQUENCE PUBLIC.SYSTEM_SEQUENCE_6AD8FFC7
  START WITH 1
  INCREMENT BY 1
  MINVALUE 1;

CREATE MEMORY TABLE PUBLIC.EMP(
    ID BIGINT NOT NULL
       DEFAULT NEXT VALUE FOR PUBLIC.SYSTEM_SEQUENCE_6AD8FFC7
       NULL_TO_DEFAULT SEQUENCE PUBLIC.SYSTEM_SEQUENCE_6AD8FFC7,
    NAME VARCHAR(255),
    EMP_TYPE VARCHAR(255) NOT NULL,
    PRIMARY KEY (ID)
)
```

- IDENTITY is modelled with a sequence which is updated on each insert

Associations

- **Relationship modeling annotations**
  - `@OneToOne`
  - `@OneToMany`
  - `@ManyToOne`
  - `@ManyToMany`

  - Annotations must be applied to the corresponding property

```
@OneToOne
Address address;
@OneToMany
List<Rental> rentals;
```

  - Associations may be **unidirectional** or **bidirectional**
    - For bidirectional associations, one side is marked with "`mappedBy`" *(inverse side)*, the other side is called the *owning side*

ManyToOne: User - Rental: bidirectional

```
@Entity
public class Rental {
   @Id @GeneratedValue
   private int id;

   @ManyToOne            // Rental is the owner of the relationship
   @JoinColumn(name = "USER_FK") // optional
   private User user;

   public Rental() { }

   public User getUser() { return user; }
   public void setUser (User user) { this.user = user; }

   public int getId() { return id; }
   public void setId(int id) { this.id = id; }
}
```

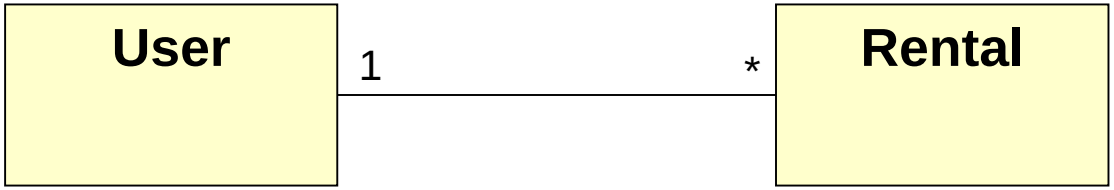
ManyToOne: User - Rental: bidirectional

```
@Entity
public class User {
   ...
   @OneToMany(mappedBy = "user")
   private List<Rental> rentals;
           // this is the inverse side of the relationship

   public Collection<Rental> getRentals() {
      return rentals;
   }
   public void setRentals(Collection<Rental> rentals) {
      this.rentals = rentals;
   }
}
```

ManyToOne: User - Rental: bidirectional

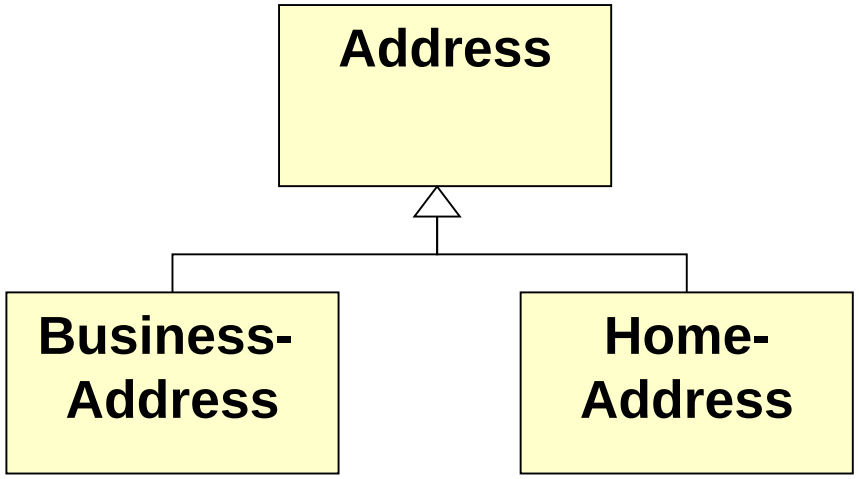
- **Representation in Database**




```
create table user (
    ID              integer,
    FIRSTNAME       varchar(32),
    LASTNAME        varchar(32),
);
```

```
create table rental (
    ID              integer,
    USER_FK         integer,
    RENTED_DAYS     integer,
    RENTED_DATE     date
);
```

Inheritance



- **Entities**
  - All classes have to be declared to be an `@Entity` classes
- **Representation**
  - All classes are (by default) stored in a SINGLE TABLE
- **Annotation**
  - `@DiscriminatorColumn(name="PRICECATEGORY_TYPE")`
    - defines the name of the column where dynamic type is stored
    - to be defined on the base class
  - `@DiscriminatorValue("Children")`
    - to be defined on the concrete base- and subclasses

Java Persistence API

- Persistence Design Patterns
- JPA Overview
- JAP Entity Annotations
- JPA Entity Manager
- Spring/SpringBoot & JPA

Entity Manager

- **EntityManager = API to access the database**
  - Entity manager stores the entities in a persistence context
  - Entity manager manages the lifecycle of entity instances
    - An entity contained in a persistence context is managed (attached)
    - An entity not associated with a persistence context is unmanaged (detached)
- **Persistence Context = set of managed objects**
  - Only one entity instance with the same persistent identity may exist in a Persistence Context
  - Entity manager automatically synchronizes the content of the persistence context with the database
  - Changes made on managed objects are automatically flushed
- **Declaration**

```
@PersistenceContext
private EntityManager em;
```

Entity Manager

```
public interface EntityManager {
   public void persist(Object entity);
   public <T> T merge(T entity);
   public void remove(Object entity);
   public <T> T find(Class<T> entityClass, Object primaryKey);

   public void flush();
   public void setFlushMode(FlushModeType flushMode);
   public FlushModeType getFlushMode();

   public void refresh(Object entity);
   public void clear();
   public boolean contains(Object entity);
   public void detach(Object entity);

   public Query createQuery(String qlString);
   public Query createNamedQuery(String name);
   ...
}
```

Entity Manager

- **persist**
  - Makes an instance managed and persistent
- **remove**
  - Removes the entity instance from the database
- **find**
  - Finds an entity by its primary key
- **merge**
  - Merges the state of the given entity into the current persistence context
    - Returns new (unique) instance
    - May be used for insert operations as well (managed instance is returned)
- **refresh**
  - Refresh the state of the (managed) instance from the database, discarding changes made on the entity, if any

Entity Manager

- **flush**
  - Synchronize the persistence context to the underlying database
- **contains**
  - Check if the instance belongs to the current persistence context (but not whether the entity is stored on the database)
- **clear**
  - Clear the persistence context, causing all managed entities to become detached
  - Changes made to entities that have not been flushed to the database will not be persisted
- **detach**
  - Removes the given entity from the persistence context

Queries

- **createQuery**
  - Creates an instance of Query for executing an JPQL statement
  - Returns a `Query` / `TypedQuery<T>` object
    - Parameters:
      - `q.setParameter(String, Object)` sets named parameters
      - `q.setParameter(int, Object)` sets positional parameters
    - Results:
      - `q.getResultList()`
      - `q.getSingleResult()`

```
TypedQuery<Movie> q = em.createQuery(
                  "SELECT m FROM Movie m WHERE m.title = :title",
                 Movie.class);
q.setParameter("title", title);
List<Movie> movies = q.getResultList();
```

Transactions

- **Transactions**
  - A transaction is needed for all EntityManager operations that change anything in the DB
    - No transaction is required for SELECT queries (without locking)

```
javax.persistence.TransactionRequiredException:
No EntityManager with actual transaction available for current
thread - cannot reliably process 'merge' call
```

- **Declare your Services as @Transactional**
  - Probably tests have to be declared as Transactional as well otherwise entities are detached

Transactions and Transaction strategies will be covered later

Java Persistence API

- **Persistence Design Patterns**
- **JPA Overview**
- **JPA Entity Annotations**
- **JPA Entity Manager**
- **Spring/SpringBoot & JPA**

Spring Boot

- **Dependencies**

```
dependencies {
  implementation("org.springframework.boot:spring-boot-starter-data-jpa")
  runtime("com.h2database:h2")
```

Category/License	Group / Artifact	Version	Updates
EDL 1.0 EPL 2.0	jakarta.persistence » jakarta.persistence-api	2.2.3	3.0.0
EPL 2.0	jakarta.transaction » jakarta.transaction-api	1.3.3	2.0.0
O/R Mapping LGPL 2.1	org.hibernate » hibernate-core	5.4.32.Final	6.0.0.Alpha9
AOP Apache 2.0	org.springframework » spring-aspects	5.3.10	✓
AOP Apache 2.0	org.springframework.boot » spring-boot-starter-aop	2.5.5	✓
Apache 2.0	org.springframework.boot » spring-boot-starter-jdbc	2.5.5	✓
Apache 2.0	org.springframework.data » spring-data-jpa	2.5.5	✓

Spring Boot Properties

- **spring.jpa**

```
org.springframework.boot.autoconfigure.orm.jpa.JpaProperties
```

```
JpaProperties
    JpaProperties()
    properties : Map<String, String>
    mappingResources : List<String>
    databasePlatform : String
    database : Database
    generateDdl : boolean
    showSql : boolean
    openInView : Boolean
```

- **spring.jpa.hibernate**

```
org.springframework.boot.autoconfigure.orm.jpa.HibernateProperties
```

```
HibernateProperties
    HibernateProperties()
    naming : Naming
    ddlAuto : String
    useNewIdGeneratorMappings : Boolean
```

Spring Boot Properties

- **spring.datasource**

```
org.springframework.boot.autoconfigure.jdbc.DataSourceProperties
```

DataSourceProperties
- DataSourceProperties()
- classLoader : ClassLoader
- generateUniqueName : boolean
- name : String
- type : Class<? extends DataSource>
- driverClassName : String
- url : String
- username : String
- password : String
- jndiName : String
- schema : List<String>
- embeddedDatabaseConnection : EmbeddedDatabaseConnection
- xa : Xa
- uniqueName : String

- **spring.h2.console**

```
org.springframework.boot.autoconfigure.h2.H2ConsoleProperties
```

H2ConsoleProperties
- H2ConsoleProperties()
- path : String
- enabled : boolean
- settings : Settings

Spring Boot Properties

- **spring.sql.init**

```
org.springframework.boot.autoconfigure.sql.init.
                                        SqlInitializationProperties
```

```
SqlInitializationProperties
    SqlInitializationProperties()
    schemaLocations : List<String>
    dataLocations : List<String>
    platform : String
    username : String
    password : String
    continueOnError : boolean
    separator : String
    encoding : Charset
    mode : DatabaseInitializationMode
```

IDEs provide auto completion in `application.properties`

- **Documentation**
  - https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

Spring Boot Entity Scanning

- **EntityScan**
  - Spring Boot uses "Entity Scanning" to look for Entities
    - The package containing the class annotated with @EnableAutoConfiguration is used as base (sub packages are scanned as well)
    - May be overwritten with `@EntityScan`

```
@Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME)
public @interface EntityScan {

   // list of base packages to scan for entities
   String[] basePackages() default {};
   String[] value() default {}; // alias for basePackages

   // type safe variant to specify base packages to scan for
   // entities
   Class<?>[] basePackageClasses() default {};
}
```

Spring Boot Data-Source Properties

- **Datasource Properties**

```
# JDBC URL of the database
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.url=jdbc:h2:file:./movierental-db
spring.datasource.url=jdbc:h2:./movierental-db
spring.datasource.url=jdbc:h2://localhost:8888/test
```

  – If no URL is specified, Spring Boot auto generates the database name for in-memory databases (H2, HSQL and Derby), name is shown in the log, e.g. `jdbc:h2:mem:94b35e1f-228d-4ab1-99da-bd2879b3605d`
  – This behavior can be disabled using

```
spring.datasource.generate-unique-name=false
```

  – Data source name is then `testdb` for embedded DBs, null otherwise, can be changed with

```
spring.datasource.name=movie-db
```

Spring Boot Data-Source Properties

- **Login username and password of the database**

```
# username & password (sa/<empty> for H2 in-memory DB)
spring.datasource.username=movieadmin
spring.datasource.password=changeme
```

- **Name of the JDBC driver**

```
# Autodetected based on the URL by default
spring.datasource.driver-class-name=org.h2.Driver
```

- **Using a different connection pool**

```
spring.datasource.type=org.apache.tomcat.jdbc.pool.DataSource
```

  - HakariCP (default)
  - Tomcat Connection Pooling
  - Commons DBCP2
  - Oracle UCP

Spring Boot Schema Creation

- **Creating Schemas**
  - By default, JPA databases will be created automatically only if an embedded database (H2, HSQL or Derby) is used
  - Default behavior can be overridden
    - Hibernate property

```
hibernate.hbm2ddl.auto
```

```
spring.jpa.hibernate.ddl-auto=create-drop
```

      - `none`
      - `validate` validate the schema, makes no changes to the database
      - `update` update the schema
      - `create` creates the schema, destroying previous data
      - `create-drop` create and then drop the schema at the end of the session
    - Defaults
      - For an embedded DB the default is `create-drop`
      - For a non-embedded DB the default is `none`

Database Initialization using Hibernate

- **import.sql**
  - A file named `import.sql` in the root of the classpath is executed on startup if `dll-auto=create` or `create-drop`.
  - This is a Hibernate-feature (and has nothing to do with Spring)
  - Name of the file can be configured:

```
# configuration for Hibernate (import.sql by default)
javax.persistence.hibernate.hbm2ddl.import_files=d1.sql,d2.sql
```

Database Initialization using Spring JDBC

- **Database Initialization using SQL-Scripts (schema.sql / data.sql)**
  - Configuration properties
    - `spring.sql.init.mode` (always/embedded/never)
      - Defines whether scripts are executed
    - `spring.sql.init.schema-locations`/`spring.sql.init.data-locations`
      - path to schema and data files; can be absolute, relative or classpath based
    - `spring.sql.init.continue-on-error`
      - Indicates whether program should stop if an error occurs while initializing the database
  - **Default Parameters**

```
spring.sql.init.mode=embedded
spring.sql.init.schema-locations=classpath:/schema.sql
spring.sql.init.data-location=classpath:/data.sql
spring.sql.init.continue-on-error=false
```

Database Initialization using Spring JDBC

- **Initialization Order**
  - Script-based DataSource initialization is performed, by default, *before* any JPA `EntityManagerFactory` beans are created, i.e. before schema is created by JPA
    - Typically either script-based or JPA-based initialization is used
    - JPA overrides schemas defined by scripts
  - If scripts should be executed *after* the schema was created by Hibernate, then set the following property:

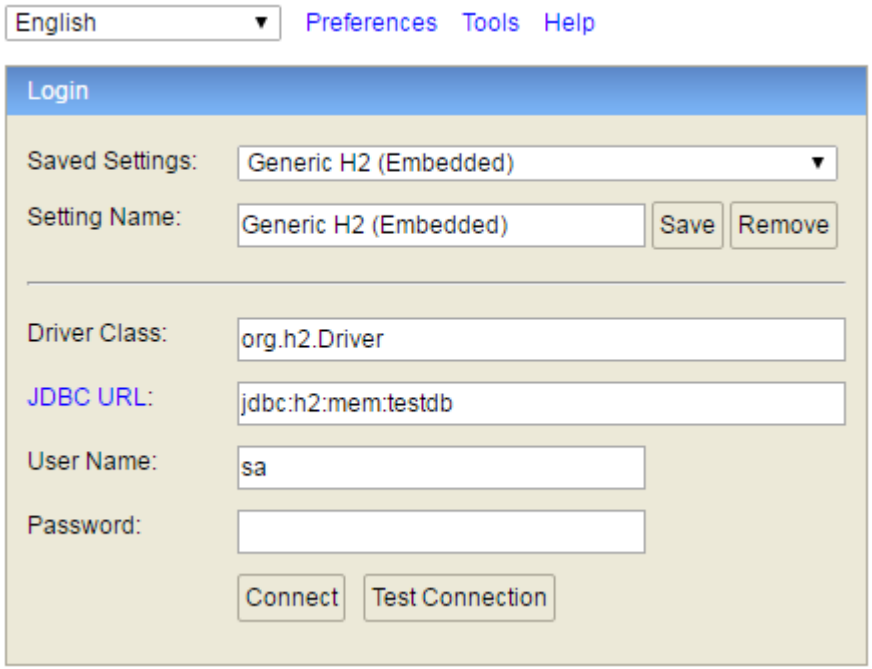
```
spring.jpa.defer-datasource-initialization=true
```

    - `schema.sql` can then be used to make additions to any created schema
    - `data.sql` can be used to populate the created schemas

Spring Boot H2 Properties

- **H2's Browser Based Web Console**

```
spring.h2.console.enabled=true   # default: false
spring.h2.console.path=/h2       # default: /h2-console, must start
                                 # with a "/"
```



Spring Boot Additional Properties

- **JPA Logging**

```
# Enable logging of SQL statements, default is false
spring.jpa.show-sql=true
```

- **Hibernate Properties**
  - Additional native properties to be set on a JPA provider (as e.g. Hibernate) start with `spring.jpa.properties.*` (this prefix is dropped before passing them)
  - https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#configurations

```
# JDBC connection transaction isolation level
spring.jpa.properties.hibernate.connection.isolation=8
# Allows Hibernate to generate SQL optimized for a particular DB
# Value is chosen based on the metadata returned by JDBC-driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```