DB

Tables, fields

- A database **table** is composed of records and fields that hold data. Tables are also called datasheets. Each table in a database holds data about a different, but related, subject.
- Data is stored in records. A record is composed of fields and contains all the data about one particular item (person, company etc.) in a database. Records appear as rows in the database table.
- A field is part of a record and contains a single piece of data for the subject of the record. Fields appear as columns in a database table.

CRUD - Create, Read, Update, Delete

- When we are building APIs, we want our models to provide four basic types of functionality. The model must be able to Create, Read, Update, and Delete resources. Computer scientists often refer to these functions by the acronym CRUD. A model should have the ability to perform at most these four functions in order to be complete. If an action cannot be described by one of these four operations, then it should potentially be a model of its own.
- In a REST environment, CRUD often corresponds to the HTTP methods POST, GET, PUT, and DELETE, respectively. These are the fundamental elements of a persistent storage system.

Spring CrudRepository

CrudRepository provides generic CRUD operation on a repository for a specific type. CrudRepository is a Spring data interface and to use it we need to create our interface by extending CrudRepository for a specific type. Spring provides CrudRepository implementation class automatically at runtime. It contains methods such as save, findById, delete, count etc. If we want to add extra methods, we need to declare it in our interface. All the methods of CrudRepository are annotated with @Transactional in implementation class by default at runtime.

- Recommended standards and response codes in a RESTful environment:

- CREATE - To create resources in a REST environment, we most commonly use the HTTP POST method.

  POST creates a new resource of the specified resource type. Upon successful creation, the server should return a header with a link to the newly-created resource, along with a HTTP response code of 201 (CREATED).

- To read resources in a REST environment, we use the GET method. Reading a resource should never change any information. Response: Status Code - 200 (OK). If there is an error, it most often will return a 404 (NOT FOUND) response code.

- PUT is the HTTP method used for the CRUD operation Update. The response includes a Status Code of 200 (OK) to signify that the operation was successful, but it need not return a response body.
- The CRUD operation Delete corresponds to the HTTP method DELETE. It is used to remove a resource from the system. Response: Status Code - 204 (NO CONTENT) Body - None

ORM

- Object-relational mapping (ORM) in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to construct their own ORM tools.
- In object-oriented programming, data-management tasks act on **objects** that are almost always **non-scalar** values. For example, an address book entry that represents a single person along with zero or more phone numbers and zero or more addresses. This could be modeled in an object-oriented implementation by a "Person object" with attributes/fields to hold each data item that the entry comprises: the person's name, a list of phone numbers, and a list of addresses. The list of phone numbers would itself contain "PhoneNumber objects" and so on. The address-book entry is treated as a single object by the programming language (it can be referenced by a single variable containing a pointer to the object, for instance). Various methods can be associated with the object, such as a method to return the preferred phone number, the home address, and so on.
- However, **many popular database** products such as SQL database management systems (DBMS) **can only store and manipulate scalar values** such as integers and strings organized within tables. The programmer must either convert the object values into groups of simpler values for storage in the database (and convert them back upon retrieval), or only use simple scalar values within the program. Object-relational mapping implements the first approach.
- The heart of the problem involves translating the logical representation of the objects into an atomized form that is capable of being stored in the database while preserving the properties of the objects and their relationships so that they can be reloaded as objects when needed. If this storage and retrieval functionality is implemented, the objects are said to be persistent.

- Compared to traditional techniques of exchange between an object-oriented language and a relational database, ORM often reduces the amount of code that needs to be written.
- Disadvantages of ORM tools generally stem from the high level of abstraction obscuring what is actually happening in the implementation code. Also, heavy reliance on ORM software has been cited as a major factor in producing poorly designed databases.

- **Alternative:** Another approach is to use an object-oriented database management system (OODBMS) or that provide more flexibility in data modeling. OODBMSs are databases designed specifically for working with object-oriented values. Using an OODBMS eliminates the need for converting data to and from its SQL form, as the data is stored in its original object representation and relationships are directly represented, rather than requiring join tables/operations. Object-oriented databases tend to be used in **complex, niche applications**.

SQL syntax

- Most of the actions you need to perform on a database are done with so-called SQL statements. Many database system, incl. mysql, require that statements be separated by semicolons
- Statements consist of keywords and references to tables (and possibly fields within) in the database, and all statements start with a keyword
- SQL keywords are NOT case sensitive, but it IS general practice to write them in ALLCAPS (whereas in MySQL, table names ARE case sensitive)
- there are slight differences in the exact syntax between the various implementations (e.g. MySQL, SQLite...)

Basic commands in command line

- To start up mysql:

$ mysql -u username -p which will ask you to enter the password for the username in question

- Switch to a specific database:

USE database_name;

- CREATE a database:

CREATE DATABASE database_name;

- DELETE a database:

DROP DATABASE database_name;

- CREATE a table:

CREATE TABLE table_name (column1 datatype, column2 datatype ..., PRIMARY KEY(one or more columns));

- DELETE a table:

DROP TABLE table_name;

- Rename a table:

ALTER TABLE table_name RENAME TO new_table_name;

- Modify a table:

ALTER TABLE table_name {ADD | DROP | MODIFY} column_name {data_type};

- View table structure:

DESC table_name;

- UPDATE data:

UPDATE table_name
SET column1 = value1, column2 = value2
[WHERE CONDITION];

- DELETE records:

DELETE FROM table_name

WHERE {condition};

- Querying / viewing records:

SELECT column1, column2 ... FROM table_name;

- View unique records only:

SELECT DISTINCT column1, column2 ...
FROM table_name;

(DISTINCT is applied to the result of the select statement, and displays all unique COMBINATIONS of the select items, i.e. it's not only column1 that DISTINCT applies to in the above example)

- View records that satisfy a certain condition:

SELECT column1, column2...
FROM table_name
WHERE CONDITION

- View records that satisfy one or more of a set of conditions:

SELECT column1, column2...
FROM table_name
WHERE CONDITION1 {AND | OR} CONDITION2;

- View records where a given field has one of a specified set of values (basically a shorthand for a chain of OR statements):

SELECT column1, column2...
FROM table_name
WHERE column_name IN (value1, value2, value3...);

- View records where a given field has a value between the provided thresholds:

SELECT column1, column2...
FROM table_name
WHERE column_name BETWEEN value1 AND value2;

- View records where the value in a given field matches the provided pattern:

SELECT column1, column2...
FROM table_name
WHERE column_name LIKE { PATTERN };

There are two wildcards often used in conjunction with the LIKE operator:
% - The percent sign represents zero, one, or multiple characters
_ - The underscore represents a single character

- Order the results of the select by a given field:

SELECT column1, column2...
FROM table_name
WHERE CONDITION
ORDER BY column_name {ASC | DESC};

- Group the result set by one or more columns:

SELECT column1, column2...

FROM table_name
WHERE CONDITION
GROUP BY column_name;

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG).
e.g.
SELECT COUNT(column_name1), column_name2
FROM table_name
GROUP BY column_name2
ORDER BY COUNT(column_name1) DESC;

- Applying conditions to results of aggregate functions:

SELECT COUNT(column_name1), column_name2
FROM table_name
GROUP BY column_name2
HAVING COUNT(column_name1) > 5
ORDER BY COUNT(column_name1) DESC;

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

JOIN, joined models

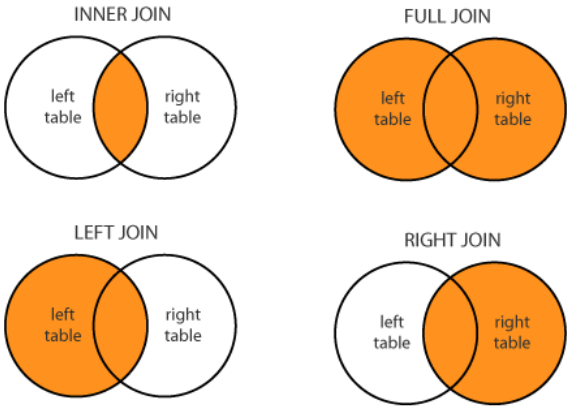
A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Example:
SELECT table1.column1, table2.column1, tabl1.column2
FROM table1
INNER JOIN table2 ON table1_primarykey=table2_foreignkey;

The SELECT statement will be applied to the already combined dataset (from table1 and table2).

There are multiple types of JOIN, not all of which are implemented in all SQL variants.

- (INNER) JOIN (usually the default type) - returns records that have matching values in both tables
- LEFT (OUTER) JOIN - Return all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN - Return all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN - Return all records when there is a match in either left or right table



FOREIGN KEY

- A FOREIGN KEY is a key used to link two tables together.
- A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.
- The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.
- FOREIGN KEY CONSTRAINT (declaring a field as foreign key at table creation or when ALTERing it)

  The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
  The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

EXAMPLES

- **New specific query from database**

There are 3 ways to create queries in JPA:

1. Query generation from Method Names
2. @Query annotation
3. with Named Queries

Method Names

The query generation from the method name is a query generation strategy where the invoked query is derived from the name of the query method.

We can create query methods that use this strategy by following these rules:

- The name of our query method must start with one of the following prefixes: find...By, read...By, query...By, count...By, and get...By.
- If we want to limit the number of returned query results, we can add the First or the Top keyword before the first By word. If we want to get more than one result, we have to append the optional numeric value to the First and the Top keywords. For example, findTopBy, findTop1By, findFirstBy, and findFirst1By all return the first entity that matches with the specified search criteria.
- If we want to select unique results, we have to add the Distinct keyword before the first By word. For example, findTitleDistinctBy or findDistinctTitleBy means that we want to select all unique titles that are found from the database.
- We must add the search criteria of our query method after the first By word. We can specify the search criteria by combining property expressions with the supported keywords.
- If our query method specifies X search conditions, we must add X method parameters to it. In other words, the number of method parameters must be equal to the number of search conditions. Also, the method parameters must be given in the same order as the search conditions.

Example 1: If we want to create a query method that returns the todo entry whose id is given as a method parameter, we have to add one of the following query methods to our repository interface:

```
import org.springframework.data.repository.Repository;

import java.util.Optional;

interface TodoRepository extends Repository<Todo, Long> {
```

```
    /**
     * Returns the found todo entry by using its id as search
     * criteria. If no todo entry is found, this method
     * returns null.
     */
    public Todo findById(Long id);

    /**
     * Returns an Optional which contains the found todo
     * entry by using its id as search criteria. If no to entry
     * is found, this method returns an empty Optional.
     */
    public Optional<Todo> findById(Long id);
}
```

Example 2: If we want to create a query method that returns todo entries whose title or description is given as a method parameter, we have to add the following query method to our repository interface:

```
...
    /**
     * Returns the found todo entry whose title or description is given
     * as a method parameter. If no todo entry is found, this method
     * returns an empty list.
     */
    public List<Todo> findByTitleOrDescription(String title, String
description);
}
```

Example 3: If we want to create a query method that returns the number of todo entries whose title is given as a method parameter, we have to add the following query method to our repository interface:

```
...
    /**
     * Returns the number of todo entry whose title is given
```

```
     * as a method parameter.
     */
    public long countByTitle(String title);
}
```

Example 4: If we want to return the distinct todo entries whose title is given as a method parameter, we have to add the following query method to our repository interface:

```
...
    /**
     * Returns the distinct todo entries whose title is given
     * as a method parameter. If no todo entries is found, this
     * method returns an empty list.
     */
    public List<Todo> findDistinctByTitle(String title);
}
```

Example 5: If we want to to return the first 3 todo entries whose title is given as a method parameter, we have to add one of the following query methods to our repository interface:

```
...
    /**
     * Returns the first three todo entries whose title is given
     * as a method parameter. If no todo entries is found, this
     * method returns an empty list.
     */
    public List<Todo> findFirst3ByTitleOrderByTitleAsc(String title);

    /**
     * Returns the first three todo entries whose title is given
     * as a method parameter. If no todo entries is found, this
     * method returns an empty list.
     */
    public List<Todo> findTop3ByTitleOrderByTitleAsc(String title);
```

```
}
```

This query generation strategy has the following benefits:

- Creating simple queries is fast.
- The method name of our query method describes the selected value(s) and the used search condition(s).

This query generation strategy has the following weaknesses:

- The features of the method name parser determine what kind of queries we can create. If the method name parser doesn't support the required keyword, we cannot use this strategy.
- The method names of complex query methods are long and ugly.
- There is no support for dynamic queries.

@Query Annotation

We can configure the invoked database query by annotating the query method with the @Query annotation. It supports both JPQL and SQL queries, and the query that is specified by using the @Query annotation precedes all other query generation strategies.

In other words, if we create a query method called findbyId() and annotate it with the @Query annotation, Spring Data JPA won't (necessarily) find the entity whose id property is equal than the given method parameter. It invokes the query that is configured by using the @Query annotation.

Example:

```
...
interface TodoRepository extends Repository<Todo, Long> {


    @Query("SELECT t FROM Todo t WHERE t.title = 'title'")
    public List<Todo> findById();
}
```

Even though the findById() method follows the naming convention that is used to create database queries from the method name of the query method, the findById() method returns todo entries whose title is 'title', because that is the query which is specified by using the @Query annotation.

Creating SQL Queries

- Add a query method to our repository interface.
- Annotate the query method with the @Query annotation, and specify the invoked query by setting it as the value of the @Query annotation's value attribute.
- Set the value of the @Query annotation's nativeQuery attribute to true.

The source code of our repository interface looks as follows:

```
...
interface TodoRepository extends Repository<Todo, Long> {


    @Query(value = "SELECT * FROM todos t WHERE t.title = 'title'",
            nativeQuery=true
    )
    public List<Todo> findByTitle();
}
```

The @Query annotation has the following benefits:

- It supports both JPQL and SQL.
- The invoked query is found above the query method. In other words, it is easy to find out what the query method does.
- There is no naming convention for query method names.

The @Query annotation has the following drawbacks:

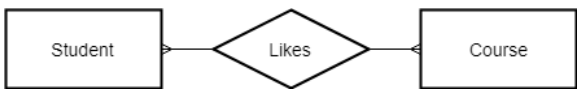
- There is no support for dynamic queries.
- If we use SQL queries, we cannot change the used database without testing that our SQL queries work as expected.

With Named Queries

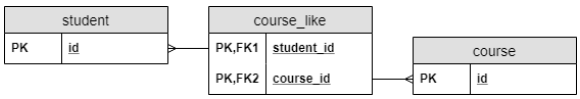
https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-with-named-queries/

- **Creating many-to-many connections**

Modeling a Many-To-Many Relationship
A relationship is a connection between two types of entities. In case of a many-to-many relationship, both sides can relate to multiple instances of the other side. For example, when the students mark the courses they like: a student can like many courses, and many students can like the same course:



As we know, in RDBMSes we can create relationships with foreign keys. Since both sides should be able to reference the other, we need to create a separate table to hold the foreign keys:



Such a table is called a join table. Note, that in a join table, the combination of the foreign keys will be its composite primary key.

Implementation in JPA
Modeling a many-to-many relationship with POJOs is easy. We should include a Collection in both classes, which contains the elements of the others.
After that, we need to mark the class with @Entity, and the primary key with @Id to make them proper JPA entities.
Also, we should configure the relationship type. Hence we mark the collections with @ManyToMany annotations:

```
@Entity
class Student {

    @Id
    Long id;

    @ManyToMany
    Set<Course> likedCourses;

    // additional properties
    // standard constructors, getters, and setters
}

@Entity
class Course {

    @Id
    Long id;

    @ManyToMany
```

```
    Set<Student> likes;

    // additional properties
    // standard constructors, getters, and setters
}
```

Additionally, we have to configure how to model the relationship in the RDBMS.

The owner side is where we configure the relationship, which for this example we'll pick the Student class.

We can do this with the @JoinTable annotation in the Student class. We provide the name of the join table (course_like), and the foreign keys with the @JoinColumn annotations. The joinColumn attribute will connect to the owner side of the relationship, and the inverseJoinColumn to the other side:

```
@ManyToMany
@JoinTable(
  name = "course_like",
  joinColumns = @JoinColumn(name = "student_id"),
  inverseJoinColumns = @JoinColumn(name = "course_id"))
Set<Course> likedCourses;
```

Note, that using @JoinTable, or even @JoinColumn **isn't required**: JPA will generate the table and column names for us. However, the strategy JPA uses won't always match the naming conventions we use. Hence the possibility to configure table and column names.

On the target side, we only have to provide the name of the field, which maps the relationship. Therefore, we set the mappedBy attribute of the @ManyToMany annotation in the Course class:

```
@ManyToMany(mappedBy = "likedCourses")
Set<Student> likes;
```

Note, that since a many-to-many relationship doesn't have an owner side in the database, we could configure the join table in the Course class and reference it from the Student class.

Actually, there isn't any many-to-many relationship in an RDBMS - those are all pairs of Many-to-One relationships. We call the structures we create with join tables many-to-many relationships because that's what we model. Besides, it's more clear if we talk about many-to-many relationships, because that's our intention. Meanwhile, a join table is just an implementation detail; we don't really care about it.

- **Basic SQL commands in terminal**

  see above: SQL syntax

- **Cascading deletion for connected records**

Marking a reference field with CascadeType.REMOVE (or CascadeType.ALL, which includes REMOVE) indicates that remove operations should be cascaded automatically to entity objects that are referenced by that field (multiple entity objects can be referenced by a collection field):

```
@Entity
class Employee {
     :
    @OneToOne(cascade=CascadeType.REMOVE)
    private Address address;
     :
}
```

The Employee entity class contains an address field that references an instance of Address, which is another entity class. Due to the CascadeType.REMOVE setting, when an Employee instance is removed the operation is automatically cascaded to the referenced Address instance, which is then automatically removed as well. Cascading may continue recursively when applicable (e.g. to entity objects that the Address object references, if any).

Orphan Removal
JPA 2 supports an additional and more aggressive remove cascading mode which can be specified using the orphanRemoval element of the @OneToOne and @OneToMany annotations:

```
@Entity
class Employee {
     :
    @OneToOne(orphanRemoval=true)
```

```
    private Address address;
     :
}
```

When an Employee entity object is removed the remove operation is cascaded to the referenced Address entity object. In this regard, orphanRemoval=true and cascade=CascadeType.REMOVE are identical, and if orphanRemoval=true is specified, CascadeType.REMOVE is redundant.

The difference between the two settings is in the response to disconnecting a relationship. For example, such as when setting the address field to null or to another Address object.

If orphanRemoval=true is specified the disconnected Address instance is automatically removed. This is useful for cleaning up dependent objects (e.g. Address) that should not exist without a reference from an owner object (e.g. Employee).
If only cascade=CascadeType.REMOVE is specified no automatic action is taken since disconnecting a relationship is not a remove operation.
To avoid dangling references as a result of orphan removal this feature should only be enabled for fields that hold private non shared dependent objects.

Deleting a Todo connected to an Assignee
(Experience from Orientation phase) There is a ManyToOne relationship between Todo and Assignee (the Assignee table has a Foreign Key Todo_ID). When trying to delete a Todo that is already assigned to an Assignee, JPA won't let you do it unless you first "break" the connection, by e.g. setting the corresponding Todo_ID to null in the Assignee table.