Veebirakenduse projekt (ITI0302)

Loeng 4

Databases: PostgreSQL, JPA, Liquibase · Siim Rebane

The problem

Assignment 1 keeps its data in a list inside the running program. Restart the container and the list is empty:

```
$ curl -X POST http://<ip>/api/teams -H 'Content-Type: application/json' -d '{"name":"A","memberCount":3}'
{"id":1,"name":"A","memberCount":3}
$ docker compose restart
$ curl http://<ip>/api/teams
[]
```

Every deploy restarts the container, so every push to `main` deletes every record. The project needs data that outlives the program, is shared by every copy of the program, and is guarded by rules the program cannot forget to check.

What a database server gives

A **database server** is a separate program that stores data on disk and answers queries from other programs over the network.

Need	What the database server does
data outlives the application	it writes to disk; the application can restart, crash or be replaced
several copies of the application	they all connect to the same server and see the same data
rules on the data	the server refuses a row that breaks a constraint, whichever program sends it
finding data	it answers queries (SQL) and uses indexes to do it fast

The application stops owning the data. It becomes a client that asks for it.

PostgreSQL

PostgreSQL ("Postgres") is an open-source relational database server. The course uses version **18**.

- **Relational**: data is stored in tables of rows and columns; tables refer to each other by keys
- **Client–server**: the server runs as its own process and listens on TCP port **5432**; clients connect with a host, port, database name, user and password
- One server holds several **databases**; each database holds its own tables
- `psql` is the command-line client that comes with it

Your Spring application is one client. `psql`, the IDE's database window and any other tool are others, all talking to the same server.

Tables, recapped

What the databases course covered, and what this course relies on:

- A **table** has named columns, each with a type: `bigint`, `text`, `boolean`, `date`, `numeric`, …
- A **row** is one record
- A **primary key** is a column (or columns) whose value is unique for every row and never null. Every table has one; usually `id`
- A **foreign key** is a column whose value must exist as a primary key in another table. The server refuses a row that points to nothing
- `NOT NULL` and `UNIQUE` are **constraints**: rules the server checks on every write

Naming, the same across the whole project: lower case, words joined with `_` (**snake_case**): `employee`, `department_id`, `start_date`. Table names in singular.

SQL: the four statements

```sql
SELECT id, name FROM employee WHERE department_id = 1;
INSERT INTO employee (name, department_id) VALUES ('Mari Maasikas', 1);
UPDATE employee SET email = 'mari@example.com' WHERE id = 1;
DELETE FROM employee WHERE id = 1;
```

And one join, because a foreign key is only useful if you can follow it:

```sql
SELECT e.name, d.name AS department
FROM employee e
JOIN department d ON d.id = e.department_id;
```

The application in this lecture writes none of these by hand. They are still what reaches the database, and the log can show them.

The example in this lecture

A staff register with two tables:

department		employee	
`id`	bigint, primary key	`id`	bigint, primary key
`name`	text, not null, unique	`name`	text, not null
		`email`	text
		`department_id`	bigint, not null, foreign key → `department(id)`

Every employee belongs to exactly one department; a department has any number of employees. Endpoints: `/api/departments` (list, create) and `/api/employees` (list, filter by name, get, create, delete).

Postgres on your laptop, in Docker

Installing Postgres on every laptop means different versions, different passwords and a server that runs all the time. A container gives everyone the same server, started and stopped with one command. `compose.yaml` in the project root:

```yaml
services:
  postgres:
    image: postgres:18
    container_name: postgres
    environment:
      POSTGRES_DB: staff
      POSTGRES_USER: staff
      POSTGRES_PASSWORD: staff
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql

volumes:
  pgdata:
```

The Compose file, line by line

Line	Does
`image: postgres:18`	the official Postgres image, major version 18
`POSTGRES_DB`	the database created on first start
`POSTGRES_USER`, `POSTGRES_PASSWORD`	the user created on first start, and its password
`ports: "5432:5432"`	port 5432 on your laptop leads to port 5432 in the container, so the application in the IDE can connect to `localhost:5432`
`volumes: pgdata:/var/lib/postgresql`	the data directory lives in a **named volume**, not in the container
`volumes: pgdata:` at the end	declares the volume; Docker creates it on first `up`

The three `POSTGRES_…` variables are read **only when the data directory is empty**, the very first time. Changing the password in the file later changes nothing in an existing database; the old password stays until the volume is deleted.

Volumes: where the data actually is

A container's own filesystem disappears with the container. A **volume** is storage that Docker keeps outside every container and attaches to one.

```
$ docker compose down
 Container postgres  Removed
$ docker compose up -d
 Container postgres  Started
$ docker exec postgres psql -U staff -d staff -c 'select count(*) from employee'
 count
-------
     2
```

The container was removed and made again; the rows are still there, because they were in the volume. `docker volume ls` lists it as `<project>_pgdata`.

`docker compose down -v` removes the volumes too. That is the command for starting over with an empty database, and the one to never run on a server by accident.

Postgres 18 moved its data directory

Older examples mount the volume at `/var/lib/postgresql/data`:

```
    volumes:
      - ./postgres-data:/var/lib/postgresql/data     # Postgres 17 and older
```

From version 18 the image keeps its data in a version-named directory under `/var/lib/postgresql`, and the volume goes on `/var/lib/postgresql`. With the old path, a Postgres 18 container stops at once:

```
Error: in 18+, these Docker images are configured to store database data in a
       format which is compatible with "pg_ctlcluster" (specifically, using
       major-version-specific directory names).
       …
       Counter to that, there appears to be PostgreSQL data in:
         /var/lib/postgresql/data (unused mount/volume)
```

`docker compose ps` shows the container as `Exited (1)`. The fix: mount the volume at `/var/lib/postgresql`.

Looking inside: `psql`

`psql` is already inside the container, so nothing needs installing. Run it there:

```
docker exec -it postgres psql -U staff -d staff
```

Part	Does
`docker exec`	runs a command inside a container that is already running
`-i`	keeps the input open, so what you type reaches the command
`-t`	gives the command a terminal, so it can show a prompt and format its output
`postgres`	the container, by its `container_name` from the Compose file
`psql`	the command to run inside it: the Postgres command-line client
`-U staff`	log in as the database user `staff` (`POSTGRES_USER`)
`-d staff`	connect to the database `staff` (`POSTGRES_DB`)

No password is asked: inside the container, Postgres trusts local connections.

psql: the commands you need

At the `staff=#` prompt:

Command	Shows
`\dt`	the tables of this database
`\d employee`	columns, types, keys and constraints of one table
`select * from employee;`	any SQL; ends with `;`
`\q`	quit

One statement without opening the prompt: `-c` instead of `-it`, the statement in quotes:

```
docker exec postgres psql -U staff -d staff -c 'select count(*) from employee'
```

The same connection from the IDE's database window (IntelliJ: Database → + → Data Source → PostgreSQL): host `localhost`, port `5432`, database, user and password from the Compose file.

From Java to the database: the layers

A Java program reaches Postgres through four pieces, each on top of the previous one:

Piece	What it is
JDBC	Java's standard API for sending SQL to a database and reading rows back
JDBC driver	the library that speaks one database's network protocol; for Postgres: `org.postgresql:postgresql`
connection pool	keeps a few connections open and lends them out, because opening a connection is slow. Spring Boot uses HikariCP
JPA / Hibernate	maps Java objects to rows, so the application works with objects and SQL is generated

You write code only against the top layer. The lower three show up in the startup log and in error messages.

JPA, Hibernate, Spring Data JPA

Rows are flat; objects have references to other objects. Translating between the two by hand is the same code in every class. An **ORM** (object-relational mapper) does it once.

- **JPA** (Jakarta Persistence API): the Java standard for ORM. Annotations (`@Entity`, `@Id`, ...) and interfaces, no implementation
- **Hibernate**: the implementation of JPA that Spring Boot uses. It generates the SQL
- **Spring Data JPA**: Spring's layer on top. You declare an interface; Spring writes its implementation at startup

One class, the **entity**, describes one table. One interface, the **repository**, gives the operations on it.

Dependencies

Spring Initializr: add **Spring Data JPA**, **PostgreSQL Driver**, **Liquibase Migration**, **Lombok**. The `pom.xml` gets:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-liquibase</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Spring Boot 4 has a starter for Liquibase. Older guides add `liquibase-core` alone; in Boot 4 the starter is what switches on the code that runs Liquibase at startup.

Dependencies with Gradle

`build.gradle`, as Initializr writes it:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-liquibase'
    implementation 'org.springframework.boot:spring-boot-starter-webmvc'
    compileOnly 'org.projectlombok:lombok'
    runtimeOnly 'org.postgresql:postgresql'
    annotationProcessor 'org.projectlombok:lombok'
}
```

`runtimeOnly` / `<scope>runtime</scope>` on the driver: your code never names a driver class; it is needed only when the program runs.

Telling Spring where the database is

`src/main/resources/application.properties`:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/staff
spring.datasource.username=staff
spring.datasource.password=staff

spring.jpa.hibernate.ddl-auto=validate
spring.jpa.open-in-view=false

spring.liquibase.change-log=classpath:db/changelog/changelog-master.xml
```

Six lines: where the database is, who logs in, what Hibernate may do to the schema, whether the database may be reached outside the service layer, and where Liquibase finds the schema.

The properties, line by line

Property	Means
`datasource.url`	`jdbc:postgresql://` + host + `:` port + `/` database name
`username`, `password`	the user from the Compose file
`ddl-auto=validate`	Hibernate checks that tables match the entities, and never changes them
`open-in-view=false`	the database is not reached while the response is being written; the default is `true`
`liquibase.change-log`	where the schema definition starts

A property can come from the environment

Every property can be overridden by an environment variable: upper case, `.` and `-` become `_`.

Property	Environment variable
`spring.datasource.url`	`SPRING_DATASOURCE_URL`
`spring.datasource.password`	`SPRING_DATASOURCE_PASSWORD`

The environment wins over `application.properties`. So the file in git holds the laptop's values, and the server sets its own in the container's environment, with the real password nowhere in git. The same mechanism carried `APP_COMMIT` in lecture 3.

The entity

```java
@Entity
@Getter
@Setter
@NoArgsConstructor
public class Employee {

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

    private String name;

    private String email;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;
}
```

One class, one table: the class `Employee` is the table `employee`, each field is a column.

The entity, annotation by annotation

Annotation	Means
`@Entity`	this class is mapped to a table; the table name comes from the class name
`@Id`	this field is the primary key
`@GeneratedValue(strategy = IDENTITY)`	the database assigns the value on insert; the field is `null` until then
`@ManyToOne`	many employees point to one department: the Java side of the foreign key
`fetch = FetchType.LAZY`	the department is loaded from the database only when something reads it
`@JoinColumn(name = "department_id")`	the column that holds the foreign key
`@Getter`, `@Setter`, `@NoArgsConstructor`	Lombok: the getters, setters and the no-argument constructor are generated at compile time

Field names map to column names with camelCase turned into snake_case: a field `startDate` is the column `start_date`.

What an entity must be

- A **class with a no-argument constructor**: Hibernate creates the object first and fills the fields after
- **Mutable**: Hibernate sets fields after loading and tracks changes to them
- **Not a record**: records have no no-argument constructor and cannot be changed after creation; lecture 1 recommended records for DTOs, and this is why they are not used for entities
- Its id is a **wrapper type** (`Long`, not `long`): `null` means "not saved yet"

The entity describes the table. It is not what the API sends and receives; that is the DTO's job.

The repository

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    List<Employee> findByNameContainingIgnoreCase(String name);
}
```

`JpaRepository<Employee, Long>`: a repository for the entity `Employee`, whose id is a `Long`. There is no class that implements this interface in the project. At startup Spring Data creates one:

```
Bootstrapping Spring Data JPA repositories in DEFAULT mode.
Finished Spring Data repository scanning in 44 ms. Found 2 JPA repository interfaces.
```

What a repository has without writing anything

Inherited from `JpaRepository`:

Method	SQL it becomes
`findAll()`	`select … from employee`
`findById(id)`	`select … where id = ?`, returns `Optional<Employee>`
`existsById(id)`	`select count(*) … where id = ?`
`save(employee)`	`insert` if the id is `null`, `update` otherwise; returns the saved entity with its id
`deleteById(id)`	`select … where id = ?`, then `delete … where id = ?`
`count()`	`select count(*) from employee`

`save` returns an object. Use the returned one: it has the id the database assigned.

Query methods from the method name

Spring Data reads the method name and writes the query:

```
List<Employee> findByNameContainingIgnoreCase(String name);
```

The SQL Hibernate sent for `?name=mari`:

```
select e1_0.id,e1_0.department_id,e1_0.email,e1_0.name from employee e1_0
    where upper(e1_0.name) like upper(?) escape '\'
```

Part of the name	Becomes
`findBy`	`select … where`
`Name`	the field `name`
`Containing`	`like '%…%'`
`IgnoreCase`	`upper(…)` on both sides

A name that does not match a field fails at startup, not at the first request.

Seeing the SQL

```
spring.jpa.show-sql=true
```

Every statement Hibernate sends is printed. `GET /api/employees` with two employees in two departments:

```
Hibernate: select e1_0.id,e1_0.department_id,e1_0.email,e1_0.name from employee e1_0
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
```

One query for the employees, then one per department, because each department is loaded lazily when its name is read. Turn it on while developing, and read what your endpoints actually send. Off in production: it goes to standard output, not through the logger.

The session: Hibernate's working area

A **session** (in JPA terms, the **persistence context**) is Hibernate's working area for one piece of work. It holds a connection borrowed from the pool, and every entity loaded through it.

- An entity loaded in an open session is **managed**: Hibernate knows it and keeps a copy of how it looked when it was loaded
- Within one session, loading the same row twice gives the same Java object
- A lazy relation (`fetch = LAZY`) is loaded through the session of the entity that holds it. After that session has closed, reading the relation fails:

```
Could not initialize proxy [ee.taltech.staff.department.Department#1] – no session
```

Sessions are opened and closed by transactions, and, unless it is switched off, around the whole web request.

`@Transactional`: one transaction, one session

A **transaction** is a group of database operations that succeed together or fail together. `@Transactional` on a method runs the whole method in one:

- a session is open for the whole method, so lazy relations can be read inside it
- if the method throws, every change it made is rolled back
- `@Transactional(readOnly = true)` on the class covers every method that only reads; methods that write carry their own `@Transactional`

On the service, because the service is where one piece of work begins and ends.

open-in-view: a session for the whole request

"View" is what happens after the controller returns: turning the result into JSON. With `spring.jpa.open-in-view=true`, Spring opens a session when the request arrives and closes it only after the response is written. That is the default, and Spring Boot says so at startup:

```
WARN … JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default.
    Therefore, database queries may be performed during view rendering.
    Explicitly configure spring.jpa.open-in-view to disable this warning
```

Set in `application.properties`. This course sets it to `false`:

```
spring.jpa.open-in-view=false
```

open-in-view on and off, compared

A controller that returns entities straight from the repository, `GET /api/raw-employees`, with the default `true`:

```
Hibernate: select … from employee e1_0
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
Hibernate: select … from employee e1_0 where e1_0.department_id=?
HTTP/1.1 500
Could not write JSON: Document nesting depth (501) exceeds the maximum allowed (500 …)
```

The JSON writer read `department`, which loaded it, then its `employees`, then each employee's `department` again, until it gave up. Two of the three queries ran while the response was being written, outside any service method.

What `open-in-view=false` changes

	`open-in-view=true` (default)	`open-in-view=false`
lazy relations readable	anywhere during the request, JSON writing included	only inside `@Transactional` methods
a lazy read in the wrong place	an extra query nobody sees in the code	fails at once: `no session`
where DTOs are built	anywhere	in the service, where the data is loaded

Three terms: bean, IoC container, dependency injection

- A **bean** is an object that Spring creates and manages: every `@RestController`, every `@Service`, every repository
- The **IoC container** (inversion of control) is the part of Spring that creates the beans at startup and hands each one the beans its constructor asks for. The class does not create what it needs; it is given it
- That handing over is **dependency injection**

If a constructor asks for something that is not a bean, the application does not start, and the error names the missing type.

Beans and dependency injection

The service needs a repository. It does not create one; it asks for one in its constructor:

```
@Service
public class EmployeeService {

    private final EmployeeRepository employeeRepository;

    public EmployeeService(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }
}
```

One constructor, and Spring uses it: it creates the repository bean first and passes it in. The field is `final`: set once, never replaced.

The same with Lombok

```
@Service
@RequiredArgsConstructor
public class EmployeeService {

    private final EmployeeRepository employeeRepository;
    private final DepartmentRepository departmentRepository;
    private final EmployeeMapper employeeMapper;
}
```

`@RequiredArgsConstructor` generates the constructor with one parameter per `final` field. Adding a dependency is adding one line.

`@Autowired` on a field also injects, and is the style in older examples. It hides the dependencies, allows fields that are not `final`, and a test cannot create the class without Spring. Use constructor injection.

Why the controller does not return entities

A controller that returns the entity straight from the repository:

```
@GetMapping("/api/raw-employees")
public List<Employee> all() {
    return employeeRepository.findAll();
}
```

```
HTTP/1.1 500
HttpMessageNotWritableException: Could not write JSON:
    Could not initialize proxy [ee.taltech.staff.department.Department#1] - no session
```

The JSON writer tries to read `department`, which was never loaded, after the database session has closed. And if it worked, the response would contain every field of every related entity: the table structure becomes the API, and every column added to the table appears in the API.

DTOs: one per purpose

A **DTO** (data transfer object) is the shape the API sends or receives. Records, as in lecture 1:

```
public record EmployeeDto(Long id, String name, String email, String departmentName) {
}

public record EmployeeCreateDto(String name, String email, Long departmentId) {
}
```

- `EmployeeDto` is what the API returns: the department's name, not the department object
- `EmployeeCreateDto` is what `POST` accepts: no `id`, because the server assigns it; a `departmentId`, because that is what the client knows
- A DTO has no field you never fill and no field you never read

Mapping entity to DTO by hand

```
private EmployeeDto toDto(Employee employee) {
    return new EmployeeDto(
            employee.getId(),
            employee.getName(),
            employee.getEmail(),
            employee.getDepartment().getName());
}
```

Every entity needs this, often in both directions, and every new field means editing it. Forgetting one field compiles fine and sends `null`.

Lombok

An **annotation processor** is a program the compiler runs while compiling, which reads annotations and writes new code. **Lombok** is one: it generates boilerplate methods from annotations:

Annotation	Generates
`@Getter`, `@Setter`	a getter / setter for every field
`@NoArgsConstructor`	a constructor with no parameters
`@AllArgsConstructor`	a constructor with one parameter per field
`@RequiredArgsConstructor`	a constructor with one parameter per `final` field
`@Data`	getters, setters, `equals`, `hashCode`, `toString`, required-args constructor
`@Slf4j`	a `log` field for logging

The generated methods are in the compiled class, not in your source. The IDE needs the Lombok plugin (built into IntelliJ) and annotation processing enabled to see them.

`@Data` on entities: no

`@Data` generates `equals`, `hashCode` and `toString` from **all** fields. On an entity that means:

- `toString` reads `department`, which loads it from the database, or throws outside a session
- two entities that refer to each other call each other's `toString` without end: `StackOverflowError`
- `hashCode` changes when the id is assigned on save, so a saved entity is lost from a `HashSet` it was in

On entities: `@Getter`, `@Setter`, `@NoArgsConstructor`. On DTOs: records, which need no Lombok at all.

MapStruct

MapStruct generates the mapping code at compile time from an interface you declare:

```java
@Mapper(componentModel = "spring")
public interface EmployeeMapper {

    @Mapping(source = "department.name", target = "departmentName")
    EmployeeDto toDto(Employee employee);

    List<EmployeeDto> toDtoList(List<Employee> employees);
}
```

- `componentModel = "spring"` : the generated class is a bean, injected like a repository
- Fields with the same name are mapped automatically
- `@Mapping(source, target)` for the ones whose names differ; `department.name` follows the reference
- A list method uses the single-object method for each element

What MapStruct generates

`target/generated-sources/annotations/…/EmployeeMapperImpl.java`, written during `mvn compile`:

```
public EmployeeDto toDto(Employee employee) {
    if ( employee == null ) {
        return null;
    }
    …
    departmentName = employeeDepartmentName( employee );
    id = employee.getId();
    name = employee.getName();
    email = employee.getEmail();

    EmployeeDto employeeDto = new EmployeeDto( id, name, email, departmentName );
    return employeeDto;
}
```

Plain Java, no reflection, readable. When a mapping is wrong, open this file.

MapStruct dependencies

```
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>1.6.3</version>
</dependency>
```

And in `maven-compiler-plugin`, inside `<annotationProcessorPaths>`, in this order:

```
<path><groupId>org.projectlombok</groupId><artifactId>lombok</artifactId></path>
<path><groupId>org.projectlombok</groupId><artifactId>lombok-mapstruct-binding</artifactId><version>0.2.0</version></path>
<path><groupId>org.mapstruct</groupId><artifactId>mapstruct-processor</artifactId><version>1.6.3</version></path>
```

Gradle: `implementation 'org.mapstruct:mapstruct:1.6.3'` and three `annotationProcessor` lines in the same order.

Lombok and MapStruct: the order of the processors

Lombok and MapStruct are both annotation processors: Lombok writes getters into your classes, MapStruct writes mapper classes that call those getters.

MapStruct must see the getters Lombok made. With `mapstruct-processor` listed before `lombok` and no binding:

```
[ERROR] EmployeeMapper.java:[11,23] No property named "department.name" exists
        in source parameter(s). Type "Employee" has no properties.
```

"Has no properties" means: no getters found. `lombok-mapstruct-binding` makes MapStruct wait for Lombok, and Lombok first keeps it simple.

Three layers

```
HTTP request → Controller → Service → Repository → Postgres
```

Layer	Annotation	Knows about	Does not know about
controller	`@RestController`	HTTP: paths, methods, status codes, JSON, DTOs	the database
service	`@Service`	the rules of the application, DTOs and entities	HTTP
repository	interface extending `JpaRepository`	entities and queries	HTTP, DTOs

Each layer calls only the one below it. A controller never calls a repository.

Controller

```
@RestController
@RequestMapping("/api/employees")
@RequiredArgsConstructor
public class EmployeeController {

    private final EmployeeService employeeService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public EmployeeDto create(@RequestBody EmployeeCreateDto dto) {
        return employeeService.create(dto);
    }
}
```

- Receives the request, calls one service method, returns its result
- Takes and returns DTOs only
- No business logic: no `if` about the data, no loops over results

Service

```
@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class EmployeeService {
    …
    @Transactional
    public EmployeeDto create(EmployeeCreateDto dto) {
        Department department = departmentRepository.findById(dto.departmentId())
                .orElseThrow(() -> new ResponseStatusException(HttpStatus.BAD_REQUEST, "no such department"));
        Employee employee = new Employee();
        employee.setName(dto.name());
        employee.setEmail(dto.email());
        employee.setDepartment(department);
        return employeeMapper.toDto(employeeRepository.save(employee));
    }
}
```

- All business logic: here, the department must exist
- May call several repositories
- No controller annotations: no `@GetMapping`, `@RequestParam`, `@PathVariable`

Repository

- An interface that extends `JpaRepository<Entity, IdType>`
- Responsible for reading and writing one entity
- One query per method; no business logic
- The service decides which repository methods to call and in which order

A repository that does not extend `JpaRepository` (a class you write yourself) needs `@Repository` on it. One that extends it needs nothing.

Why the update needs no `save`

An entity loaded inside a transaction is **managed**: Hibernate keeps a copy of how it looked when it was loaded. When the transaction ends, Hibernate compares each managed entity with its copy and sends an `update` for every one that changed. This is called **dirty checking**.

- Load, change with setters, return: that is the whole update
- The `update` is sent when the method returns, not when the setter is called
- Outside `@Transactional` the entity is not managed, and the change goes nowhere unless `save` is called
- Calling `save` on a managed entity does no harm; it is not needed

The same `PUT` with an id that does not exist answers `404`, from the `orElseThrow`.

Updating a row

`PUT /api/employees/{id}` changes an employee's name and email. The service:

```
@Transactional
public EmployeeDto update(Long id, EmployeeCreateDto dto) {
    Employee employee = employeeRepository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
    employee.setName(dto.name());
    employee.setEmail(dto.email());
    return employeeMapper.toDto(employee);
}
```

There is no `save` call. The SQL it sent: the employee, its department for the DTO's `departmentName`, then the `update` when the method returned:

```
Hibernate: select e1_0.id,e1_0.department_id,e1_0.email,e1_0.name from employee e1_0 where e1_0.id=?
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
Hibernate: update employee set department_id=?,email=?,name=? where id=?
```

The schema: who creates the tables?

The tables must exist before Hibernate can use them, and they change during the project: a new column, a new table, a new constraint. Every copy of the database, on every teammate's laptop and on the server, must go through the same changes in the same order.

Two approaches:

	`ddl-auto=update`	a migration tool (Liquibase)
who changes the schema	Hibernate, guessing from the entities	you, in files in git
renaming a column	adds a new empty column, the old one stays	a changeset that renames it
same on every database	depends on each database's history	yes: the same files, applied in order
visible in code review	no	yes

This course uses **Liquibase**, with `ddl-auto=validate`: Liquibase changes the schema, Hibernate only checks it.

Liquibase: changelog and changesets

- A **changeset** is one change to the schema: create a table, add a column, add a constraint. It has an `id` and an `author`
- A **changelog** is a file listing changesets in order
- The **master changelog** includes the others, in the order they are applied

```
<databaseChangeLog …>
    <include file="changes/001-department.xml" relativeToChangelogFile="true"/>
    <include file="changes/002-employee.xml" relativeToChangelogFile="true"/>
</databaseChangeLog>
```

In `src/main/resources/db/changelog/` : `changelog-master.xml` and a `changes/` directory, one file per change, numbered so the order is visible in the file list.

A changeset: create a table

`changes/001-department.xml`, inside the same `<databaseChangeLog …>` root:

```xml
<changeSet id="001-department" author="siim">
    <createTable tableName="department">
        <column name="id" type="bigint" autoIncrement="true">
            <constraints primaryKey="true" nullable="false"/>
        </column>
        <column name="name" type="text">
            <constraints nullable="false" unique="true"/>
        </column>
    </createTable>
</changeSet>
```

`autoIncrement="true"` on Postgres becomes `generated by default as identity`, which is what `GenerationType.IDENTITY` in the entity expects.

A changeset with a foreign key

`changes/002-employee.xml`:

```
<changeSet id="002-employee" author="siim">
    <createTable tableName="employee">
        <column name="id" type="bigint" autoIncrement="true">
            <constraints primaryKey="true" nullable="false"/>
        </column>
        <column name="name" type="text">
            <constraints nullable="false"/>
        </column>
        <column name="email" type="text"/>
        <column name="department_id" type="bigint">
            <constraints nullable="false"
                         foreignKeyName="fk_employee_department"
                         references="department(id)"/>
        </column>
    </createTable>
</changeSet>
```

`references` makes the foreign key; `foreignKeyName` is its name in `\d` and in errors. `department` comes first: the key must point at a table that exists.

What the changesets built

The table in the database, `\d employee` in psql or the table's structure in the IDE:

```
                                Table "public.employee"
    Column     |  Type  | Collation | Nullable |              Default
---------------+--------+-----------+----------+-----------------------------------
 id            | bigint |           | not null | generated by default as identity
 name          | text   |           | not null |
 email         | text   |           |          |
 department_id | bigint |           | not null |
Indexes:
    "employee_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk_employee_department" FOREIGN KEY (department_id) REFERENCES department(id)
```

- `generated by default as identity`: the server assigns `id` on insert
- `employee_pkey`: the primary key, with the index Postgres builds for it
- `fk_employee_department`: the foreign key, by the `foreignKeyName` in the changeset

A foreign key at work

The server checks the rule on every insert, whichever client sends it:

```
staff=# insert into employee (name, department_id) values ('Nobody', 42);
ERROR:  insert or update on table "employee" violates foreign key constraint "fk_employee_department"
DETAIL:  Key (department_id)=(42) is not present in table "department".
```

There is no department 42, so there can be no employee in it. Without the constraint the row would be stored, and every query that joins employees to departments would silently lose it.

Every relation between tables in your project gets a foreign key.

What Liquibase does at startup

Before Hibernate starts, Liquibase compares the changelog with what this database has already run.

First start on an empty database:

```
Run:                          2
Previously run:               0
Total change sets:            2
Liquibase: Update has been successful.
```

Every start after that:

```
Run:                          0
Previously run:               2
```

A changeset runs **once per database**. The laptop, a teammate's laptop and the server each run it the first time the application starts against them.

What the startup log says when it works

```
HikariPool-1 - Starting...
Liquibase: Update has been successful. Rows affected: 0
Database JDBC URL [jdbc:postgresql://localhost:5432/staff]
Database version: 18.6
Initialized JPA EntityManagerFactory for persistence unit 'default'
Tomcat started on port 8080 (http) with context path '/'
Started StaffApplication in 5.029 seconds
```

In order: the pool opens connections, Liquibase brings the schema up to date, Hibernate checks the schema against the entities, the web server starts. If any step fails, the application stops and the web server never starts.

How Liquibase remembers

Two tables of its own, in your database:

```
staff=# select id, author, filename, orderexecuted, md5sum from databasechangelog;
       id       | author |                 filename                  | orderexecuted |               md5sum
----------------+--------+-------------------------------------------+---------------+-----------------------------------
 001-department | siim   | db/changelog/changes/001-department.xml |             1 | 9:98ab0480ca36a3b4a47750045cbf7c55
 002-employee   | siim   | db/changelog/changes/002-employee.xml   |             2 | 9:8de725eacb3f7cced66549f8dd8a11ad
```

- `databasechangelog`: one row per changeset that has run, with a **checksum** (`md5sum`) of its content
- `databasechangeloglock`: a lock, so that two copies of the application starting at once do not both apply the same change

Never edit a changeset that has run

Change `email` from `text` to `varchar(200)` inside `002-employee`, which has already run, and restart:

```
ERROR: Exception Primary Reason:  Validation Failed:
     1 changesets check sum
          db/changelog/changes/002-employee.xml::002-employee::siim
          was: 9:8de725eacb3f7cced66549f8dd8a11ad but is now: 9:09bc0489fd5efcafd3c91d4c0a60ebda
```

The application does not start. The checksum in `databasechangelog` no longer matches the file, and Liquibase refuses to guess: the database was built from the old version.

A changeset that has run anywhere, on a teammate's laptop or on the server, is frozen. Every change is a **new** changeset.

Changing the schema: a new changeset

`changes/003-employee-phone.xml`, and one more `<include>` line at the end of the master changelog:

```
<changeSet id="003-employee-phone" author="siim">
    <addColumn tableName="employee">
        <column name="phone" type="text"/>
        <column name="start_date" type="date"/>
    </addColumn>
</changeSet>
```

Next start:

```
Run:                          1
Previously run:               2
```

Then the two fields in the entity: `private String phone;` and `private LocalDate startDate;`. The changeset and the entity change go into the same commit.

`ddl-auto=validate` catches a missing column

The entity gets a field, the changeset is forgotten:

```
private String phone;
```

```
Schema validation: missing column [phone] in table [employee]
```

The application does not start. Without `validate`, it would start and fail on the first request that reads employees. With it, the mismatch between entity and schema is found before any request, and on the server before the new version replaces the old one's traffic.

Relations: three kinds

Kind	Example	In the tables
many-to-one	many employees, one department each	a foreign key column in `employee`: `department_id`
one-to-many	one department, many employees	**the same** foreign key, seen from the department's side
many-to-many	an employee works on many projects, a project has many employees	a third table, the **join table**, with two foreign keys

Many-to-one and one-to-many are one relation seen from its two ends: one column in the database, and a field on either or both entities in Java.

`@OneToMany` : the other end of the same foreign key

```java
@Entity
@Getter
@Setter
@NoArgsConstructor
public class Department {
    …
    @OneToMany(mappedBy = "department")
    private List<Employee> employees = new ArrayList<>();
}
```

- `mappedBy = "department"` : this list is described by the field `department` in `Employee` . No new column, no new changeset
- The list is loaded lazily, when something reads it, with the SQL:

```
Hibernate: select e1_0.department_id,e1_0.id,e1_0.email,e1_0.name from employee e1_0 where e1_0.department_id=?
```

- The side with `@JoinColumn` , `Employee.department` , is the **owning side**: changing it changes the database. Adding an employee to `department.getEmployees()` alone changes nothing

Many-to-many: the join table

`changes/004-project.xml`: the `project` table, and a table that holds pairs:

```
<createTable tableName="employee_project">
    <column name="employee_id" type="bigint">
        <constraints nullable="false" foreignKeyName="fk_employee_project_employee"
                     references="employee(id)"/>
    </column>
    <column name="project_id" type="bigint">
        <constraints nullable="false" foreignKeyName="fk_employee_project_project"
                     references="project(id)"/>
    </column>
</createTable>
<addPrimaryKey tableName="employee_project" columnNames="employee_id, project_id"
               constraintName="pk_employee_project"/>
```

One row per employee–project pair. The primary key is the pair: the same employee cannot be on the same project twice.

`@ManyToMany` in the entities

The owning side, `Employee`, names the join table and its two columns:

```
@ManyToMany
@JoinTable(name = "employee_project",
        joinColumns = @JoinColumn(name = "employee_id"),
        inverseJoinColumns = @JoinColumn(name = "project_id"))
private Set<Project> projects = new HashSet<>();
```

The other side, `Project`, points back to it:

```
@ManyToMany(mappedBy = "projects")
private Set<Employee> members = new HashSet<>();
```

- `Set`, not `List`: a pair exists once
- The join table has no entity of its own. When the pair needs data of its own, a role or a start date, the join table becomes an entity with two `@ManyToOne` fields

Adding an employee to a project

`PUT /api/projects/{id}/members/{employeeId}`:

```
@Transactional
public ProjectDto addMember(Long projectId, Long employeeId) {
    Project project = find(projectId);
    Employee employee = employeeRepository.findById(employeeId)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
    employee.getProjects().add(project);
    project.getMembers().add(employee);
    return toDto(project);
}
```

The last statement Hibernate sent:

```
Hibernate: insert into employee_project (employee_id,project_id) values (?,?)
```

Both sides are updated: the owning side for the database, the other side so that the objects in memory agree with it.

Only the owning side is written

The same method without `employee.getProjects().add(project);`:

```
$ curl -X PUT localhost:8080/api/projects/1/members/2
{"id":1,"name":"Intranet","members":["Jaan Tamm","Kati Karu","Mari Mustikas"]}
$ curl localhost:8080/api/projects/1
{"id":1,"name":"Intranet","members":["Kati Karu","Mari Mustikas"]}
```

The answer lists Jaan, because the `Set` in memory has him. The database does not: no `insert` was sent, because `Project.members` is the `mappedBy` side. The next request reads the database, and Jaan is gone.

The rule for every bidirectional relation: change the owning side, the one with `@JoinColumn` or `@JoinTable`, always. Change the other side too, to keep the objects consistent.

Reading by relation

`GET /api/departments/{id}/employees` : all employees of one department. A query method that walks the relation:

```
List<Employee> findByDepartmentId(Long departmentId);
List<Employee> findByDepartmentName(String departmentName);
```

```
findByDepartmentId:
  select … from employee e1_0 where e1_0.department_id=?
findByDepartmentName:
  select … from employee e1_0 left join department d1_0 on d1_0.id=e1_0.department_id where d1_0.name=?
```

- `DepartmentId` : the field `department` , then its `id` . The foreign key column holds that value, so no join is needed
- `DepartmentName` : the field `department` , then its `name` . That is in the other table, so Spring Data joins it
- The URL nests the resource under its parent: employees **of** department 5 is `/api/departments/5/employees`

Two ways to the same list

	`findByDepartmentId(id)`	`department.getEmployees()`
needs	a repository method	a `@OneToMany` field on `Department`
queries	one: employees where `department_id = ?`	two: the department, then its employees
filtering, sorting, paging	in the query	in Java, after loading every employee

For an endpoint that returns the list, the query method. The collection is for code that already has the department in hand and needs all of its employees.

Errors the database sends back

A department name must be unique. `POST /api/departments` with a name that exists:

```
HTTP/1.1 500
ERROR: duplicate key value violates unique constraint "department_name_key"
```

The constraint worked: the duplicate was not stored. The client got `500 Internal Server Error`, which says "the server is broken", not "the name is taken". The service can check first and return a `4xx`, as `create` does for a department that does not exist:

```
POST /api/employees  {"name":"X","departmentId":42}
HTTP/1.1 400
```

The generated test needs the database

Spring Initializr adds one test, `contextLoads`: it starts the whole application. With JPA and Liquibase, starting means connecting to the database. In the pipeline from lecture 3 there is no database, and the build job goes red:

```
Caused by: org.postgresql.util.PSQLException: Connection to localhost:5432 refused.
[ERROR] Tests run: 1, Failures: 0, Errors: 1, Skipped: 0 <<< FAILURE! -- in ee.taltech.staff.StaffApplicationTests
Job failed: exit code 1
```

The same happens on a laptop where Postgres is not running: `./mvnw package` runs the tests.

A database for the pipeline: a service

A job can start extra containers next to itself: `services`. The build job gets its own empty Postgres:

```yaml
build:
  stage: build
  image: maven:3.9-eclipse-temurin-25
  services:
    - postgres:18
  variables:
    MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"
    POSTGRES_DB: staff
    POSTGRES_USER: staff
    POSTGRES_PASSWORD: staff
    SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/staff
```

The service is reachable by its image name, `postgres`. The `POSTGRES_…` variables reach the service, `SPRING_DATASOURCE_URL` reaches Maven.

The pipeline's database, in the job log

```
Starting service postgres:18...
Waiting for services to be up and running (timeout 30 seconds)...
liquibase.util : Run:                          2
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
Job succeeded
```

The database exists only for the job and is thrown away with it. Liquibase builds the schema from zero on every run, which also tests the changelog.

The database password: a CI/CD variable

The server's database needs a password, and it must not be in git. It goes where the Docker Hub token went in lecture 3:

1. Generate one, once, on your laptop:

```
openssl rand -hex 12
```

2. GitLab → Settings → CI/CD → Variables → Add variable:

Key	Value	Flags
`DB_PASSWORD`	the generated value	**Mask variable** on

Never change it afterwards. Postgres reads the password once, when its volume is created; a new value in GitLab would no longer match the database.

The deploy job writes it into `.env`

Lecture 3's deploy job writes `.env` with one line, `DOCKER_IMAGE=…`, using `>`, which replaces the file. The password needs a second line, appended with `>>`. The `script` line of the deploy job, broken over lines for reading; YAML joins them into one command:

```
    - ssh -o StrictHostKeyChecking=no ubuntu@$SERVER_IP
        "cd /opt/app
        && echo DOCKER_IMAGE=$DOCKER_IMAGE > .env
        && echo DB_PASSWORD=$DB_PASSWORD >> .env
        && docker compose pull && docker compose up -d"
```

After a deploy, `/opt/app/.env` on the server:

```
DOCKER_IMAGE=<username>/iti0302-2026
DB_PASSWORD=<the generated value>
```

The file is written again on every deploy, so nothing in it is edited by hand. Compose reads it and fills in every `${DOCKER_IMAGE}` and `${DB_PASSWORD}` in `compose.yaml`.

The database on the server: the `app` service

On the server, Postgres runs next to the application, in the same `compose.yaml`. The application's half:

```
services:
  app:
    image: ${DOCKER_IMAGE}:latest
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/staff
      SPRING_DATASOURCE_USERNAME: staff
      SPRING_DATASOURCE_PASSWORD: ${DB_PASSWORD}
    ports:
      - "8080:8080"
    depends_on:
      postgres:
        condition: service_healthy
    restart: unless-stopped
```

The database on the server: the `postgres` service

The second half of the same file:

```
  postgres:
    image: postgres:18
    environment:
      POSTGRES_DB: staff
      POSTGRES_USER: staff
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U staff -d staff"]
      interval: 5s
      retries: 10
    restart: unless-stopped

volumes:
  pgdata:
```

The server's Compose file, line by line

Line	Does
`postgres:5432` in the URL	inside a Compose project, a service's name is its host name; the app reaches the database as `postgres`, not `localhost`
`ports: "5432:5432"` on `postgres`	the database answers on the server's port 5432, so the IDE can connect to it; the lab network lets in only 22, 80 and 443, so IntelliJ connects through an SSH tunnel (Data Source → SSH/SSL)
`${DB_PASSWORD}`	from `.env`; the same password goes to the app's datasource and to Postgres
`depends_on … service_healthy`	the app starts only after the database answers
`healthcheck` with `pg_isready`	how Docker knows the database answers: `pg_isready` exits 0 when it accepts connections
`pgdata` volume	the data survives every deploy, because a deploy replaces containers, not volumes
`restart: unless-stopped`	both start again after the server reboots

Measured on a course server (1 vCPU, 2 GB): the application starts in 16 s; the app uses 209 MiB of memory, Postgres 45 MiB.

Lab 4

- Assignment 1: questions and help. Due 5th week Sunday 23:59
  - Check your deployed application yourself: iti0302.digivader.com/a1-check/
  - Assignment 1 stays in-memory, as the assignment says
- Teams: register your team in Moodle; milestone 1 is submitted by a team
- Milestone 1: start the team's repository, its pipeline and its first entities and changesets; everything in this lecture applies



Questions