# Veebirakenduse projekt (ITI0302)

## Loeng 6

The frontend on the server · relations, fetching and queries · Siim Rebane

# The problem

The frontend from lecture 5 runs on a laptop: `npm run dev` starts Vite, the browser opens `localhost:5173`, and Vite forwards `/api` to the backend on `localhost:8080`.

Milestone 1 asks for something else:

| Requirement | Means |
|---|---|
| `http://<server>/` opens the application | the frontend is served from the team's server, on port 80 |
| its requests to `/api` reach the backend | the backend runs on the same server |
| a push to `main` changes what runs | the frontend has its own build and deploy jobs |

Nothing of that is done by hand on the server after the first setup. The laptop, Node and Vite are not part of it.

# What `npm run build` makes

```
$ npm run build
> vite build
vite v8.3.1 building client environment for production...
✓ 93 modules transformed.
dist/index.html                   0.42 kB │ gzip:  0.28 kB
dist/assets/index-D3qsKolI.css    3.05 kB │ gzip:  1.07 kB
dist/assets/index-BI5xrAm3.js   152.32 kB │ gzip: 57.59 kB
✓ built in 413ms
```

| File | Is |
|---|---|
| `dist/index.html` | one page that loads the other two |
| `dist/assets/index-….js` | every component, Vue itself, the router and Axios, in one file |
| `dist/assets/index-….css` | every style |

Three static files. Serving them needs a program that answers `GET` with a file's content — a **web server**. Node and Vite are needed to build, not to serve.

# The web server is already on the server

Lab 2 installed nginx on the team's server and put it in front of the backend. What it does now, and what the frontend needs from it:

| | Now, since lab 2 | For the frontend |
|---|---|---|
| port 80 | nginx listens | the same |
| `/api/…` | forwarded to the backend on `localhost:8080` | the same |
| every other path | a file from `/var/www/html`: nginx's own welcome page | the three built files, put there by the pipeline |
| a path that is not a file | 404 | `index.html`, so that the router's pages work |

Two changes: the files arrive in `/var/www/html`, and one line in nginx's site file.

# Two kinds of requests, one address

The browser sends every request to the same host, `http://<server>/`:

| Request | Must be answered by |
|---|---|
| `GET /` | nginx, with `index.html` |
| `GET /assets/index-….js` | nginx, with the file |
| `GET /api/employees` | the backend |

nginx does both: it serves the files it has, and **forwards** requests for `/api/…` to the backend and passes the answer back. A server that forwards requests to another server on the caller's behalf is a **reverse proxy** — the role nginx has had since lab 2, and the role Vite's `proxy` setting played on the laptop.

The browser sees one origin, `http://<server>`, for the page and for the data: no CORS.

# The site file

`/etc/nginx/sites-enabled/default` on the server, as lab 2 left it, with one line changed:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    root /var/www/html;
    index index.html index.htm index.nginx-debian.html;
    server_name _;

    location /api {
        proxy_pass http://localhost:8080;
    }

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

| Line | Means |
|---|---|
| `root /var/www/html` | where the files are |
| `index index.html …` | what `/` means: the first of these that exists |
| `location /api { proxy_pass http://localhost:8080; }` | every path starting with `/api` goes to the backend, path unchanged — lab 2 |
| `location / { try_files $uri $uri/ /index.html; }` | every other path: the file with that name; else a directory; else `index.html` |

# The one line: why `try_files` ends with `index.html`

The router from lecture 5 puts the current page into the address: `http://<server>/employees`. In the browser that is a change of the address without a request. Typed into the address bar, or reloaded, it **is** a request: `GET /employees`.

There is no file named `employees` in `/var/www/html`. Lab 2's line ends with `=404`, and nginx answers 404. With `/index.html` at the end, nginx sends `index.html`, the application starts, the router reads `/employees` from the address and shows that page.

```
$ sudo nano /etc/nginx/sites-enabled/default      # =404  →  /index.html
$ sudo nginx -t
nginx: configuration file /etc/nginx/nginx.conf test is successful
$ sudo systemctl reload nginx
```

On the server, before and after the reload, with the built files in place:

```
$ curl -s -o /dev/null -w "%{http_code}\n" localhost/employees
404
$ curl -s -o /dev/null -w "%{http_code}\n" localhost/employees
200
```

# Getting the files there: `rsync`

`rsync` copies a directory to another machine over SSH and sends only what changed. The deploy job runs:

```
rsync -e "ssh -o StrictHostKeyChecking=no" -avz --delete --chmod=D755,F644 frontend/dist/ ubuntu@$SERVER_IP:/var/www/html/
```

| Part | Means |
|---|---|
| `-e "ssh -o StrictHostKeyChecking=no"` | go over SSH, and accept the server's host key, as the deploy job of lecture 3 does |
| `-a` | copy the directory tree as it is |
| `-v` , `-z` | list the files; compress on the way |
| `--delete` | remove files on the server that are no longer in `dist/` : the old `index-….js` of the previous build |
| `--chmod=D755,F644` | directories and files readable by nginx, whoever created them |
| `frontend/dist/` with the trailing `/` | the **content** of `dist/` , not a `dist` directory inside the target |
| `ubuntu@$SERVER_IP:/var/www/html/` | nginx's `root` |

`/var/www/html` belongs to `root` . Once, by hand, like `/opt/app` in lab 3:

```
sudo chown -R ubuntu:ubuntu /var/www/html
```

# What the copy prints

```
Transfer starting: 6 files
deleting index.nginx-debian.html
./
favicon.ico
index.html
assets/
assets/index-BI5xrAm3.js
assets/index-D3qsKolI.css

sent 59410 bytes  received 120 bytes  1484538 bytes/sec
total size is 160092  speedup is 2.69
```

`deleting index.nginx-debian.html`: nginx's welcome page, gone because of `--delete`. On the next deploy with a changed component, the list holds the new `index-….js`, the deletion of the old one, and `index.html`; the unchanged `favicon.ico` is not sent.

# One repository or two

A team's backend and frontend can live:

**in one repository** (a monorepo)

```
iti0302-2026-staff/
├── backend/
│   ├── pom.xml
│   └── src/
├── frontend/
│   ├── package.json
│   └── src/
├── compose.yaml
├── .gitlab-ci.yml
└── README.md
```

**in two repositories**

```
iti0302-2026-staff-backend/
├── pom.xml
├── src/
├── compose.yaml
└── .gitlab-ci.yml

iti0302-2026-staff-frontend/
├── package.json
├── src/
└── .gitlab-ci.yml
```

Milestone 1 allows both. Either way the repository names start with `iti0302-2026`.

# What each one means

| | One repository | Two repositories |
|---|---|---|
| a feature that changes the API and the page | one branch, one merge request, reviewed together | two merge requests, merged in the right order |
| issues, wiki, README | one place | two places, or one chosen for both |
| pipeline | one `.gitlab-ci.yml` with jobs for both parts; `rules: changes:` runs a part's jobs only when its folder changed | one simple pipeline per repository |
| history | one history for the whole application | each part's history separately |
| permissions | the same people on both parts | can differ per part |
| deploying to the team's server | both parts from one pipeline | two pipelines deploy to the same server: Compose in `/opt/app` from one, `/var/www/html` from the other |

The rest of this lecture uses one repository.

# The pipeline: the same three stages, five jobs

The stages are lecture 3's. Build exists once per part, dockerize for the backend only, deploy once per part.

| Stage | Job | Does |
|---|---|---|
| build | `build-backend` | Maven: tests with a Postgres service, `app.jar` |
| build | `build-frontend` | npm: `dist/` |
| dockerize | `dockerize` | the backend's image, pushed |
| deploy | `deploy-backend` | `compose.yaml` and `.env` to the server, `docker compose pull && up -d` |
| deploy | `deploy-frontend` | `dist/` to `/var/www/html` with `rsync` |

The frontend has no image: three static files need no container, and the web server that serves them is already on the server. Jobs of one stage run at the same time when the runners allow it; on the TalTech runner they ran one after the other ( `build-backend` 51 s, then `build-frontend` 20 s). Dockerize and deploy run on `main` only, as in lecture 3.

# build-frontend

```yaml
build-frontend:
  stage: build
  image: node:22-alpine
  cache:
    key: npm
    paths:
      - frontend/node_modules
  script:
    - cd frontend
    - npm ci
    - npm run build
  artifacts:
    paths:
      - frontend/dist
```

| Line | Means |
| --- | --- |
| `image: node:22-alpine` | Node 22 and npm, nothing else |
| `cache: … frontend/node_modules` | the installed packages are kept for the next run, as `.m2/repository` is for Maven |
| `npm ci` | install exactly what `package-lock.json` says; `npm install` may change the lock file, `ci` refuses to |
| `npm run build` | the `build` script of `package.json`: `vite build` |
| `artifacts: paths: frontend/dist` | the built files are kept and given to the later stages |

# The job log

```
$ npm ci
added 171 packages, and audited 172 packages in 3s
$ npm run build
> vite build
vite v8.3.1 building client environment for production...
dist/index.html                   0.42 kB │ gzip:  0.28 kB
dist/assets/index-D3qsKolI.css    3.05 kB │ gzip:  1.07 kB
dist/assets/index-BI5xrAm3.js   152.32 kB │ gzip: 57.59 kB
✓ built in 534ms
Saving cache for successful job
frontend/node_modules: found 4123 matching artifact files and directories
Uploading artifacts for successful job
Job succeeded
```

The same three files as on the laptop. 20 seconds for the job; most of it is the runner starting the container and uploading the artifact.

# `build-backend`: two lines differ from lecture 4

```yaml
build-backend:
  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
  cache:
    key: maven
    paths:
      - .m2/repository
  script:
    - cd backend
    - ./mvnw -B clean package
    - cp target/*.jar app.jar
  artifacts:
    paths:
      - backend/app.jar
    reports:
      junit: backend/target/surefire-reports/TEST-*.xml
```

`cd backend` first, and every path in `artifacts` starts with `backend/`: the job's working directory is the repository's root, where there is no `pom.xml`. The `dockerize` job likewise starts with `cd backend`.

# `deploy-frontend`

```yaml
deploy-frontend:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --no-cache openssh-client rsync
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh && chmod 700 ~/.ssh
  script:
    - rsync -e "ssh -o StrictHostKeyChecking=no" -avz --delete --chmod=D755,F644 frontend/dist/ ubuntu@$SERVER_IP:/var/www/html/
  only:
    - main
```

| Line | Means |
|---|---|
| `before_script` | lecture 3's deploy job, with `rsync` installed next to the SSH client |
| `frontend/dist/` | the artifact of `build-frontend`, unpacked into the job before `script` runs |
| the same `$SSH_PRIVATE_KEY`, the same `$SERVER_IP` | one deploy key and one server for both parts |

`deploy-backend` is lecture 4's deploy job, unchanged: `compose.yaml` is at the repository's root, `.env` gets `DOCKER_IMAGE` and `DB_PASSWORD`.

# The variables of the project

Settings → CI/CD → Variables: nothing new. The five variables of lectures 3 and 4 serve both parts:

| Key | Holds | Used by |
|---|---|---|
| `DOCKER_USERNAME`, `DOCKER_PASSWORD` | Docker Hub | `dockerize` |
| `DOCKER_IMAGE` | `<user>/iti0302-2026-staff` | `dockerize`, `deploy-backend` |
| `SERVER_IP` | the team's server | both deploy jobs |
| `SSH_PRIVATE_KEY` | the deploy key | both deploy jobs |
| `DB_PASSWORD` | the database password, masked | `deploy-backend` |

The frontend's deploy reaches the server with the same key as the backend's, and writes to a different place on it.

# After the deploy

```
$ curl -s -o /dev/null -w "%{http_code} %{content_type}\n" http://<server>/
200 text/html
$ curl -s -o /dev/null -w "%{http_code} %{content_type}\n" http://<server>/employees
200 text/html
$ curl -s http://<server>/api/version
{"commit":"01597ac9"}
$ curl -s http://<server>/api/employees
[{"id":1,"name":"John Smith","email":"john.smith@example.com","departmentName":"Sales"}]
```

`/api/version` answers through nginx, as it has since lab 2; `/employees` answers with the page because of the one changed line. In the browser: `http://<server>/employees` shows the page from lecture 5, with the data from the server's database.

```
$ curl -s -o /dev/null -m 5 -w "%{http_code}\n" http://<server>:8080/api/version
000
```

Port 8080 is not open in the server's firewall (lecture 2: 22, 80, 443): the backend is reachable from the internet through nginx only.

# What the browser does now

| Step | Request | Answered by |
|---|---|---|
| 1 | `GET /` | nginx: `index.html` |
| 2 | `GET /assets/index-….js`, `GET /assets/index-….css` | nginx: the files |
| 3 | the script runs; Vue mounts the application; the router reads the address | the browser |
| 4 | `GET /api/employees` from `onMounted` | nginx forwards to `localhost:8080`; the backend's JSON comes back |
| 5 | the list is rendered | the browser |

Steps 1 and 2 happen once per page load; step 4 happens for every call the components make. The address of every call is relative, `/api/…`: the same code works behind Vite on the laptop and behind nginx on the server.

# Development and production, side by side

| | On the laptop | On the server |
|---|---|---|
| serves the page | Vite's dev server, `localhost:5173` | nginx, port 80 |
| the page's files | built in memory on every change | three files built once, in the pipeline, in `/var/www/html` |
| `/api` goes to | `localhost:8080`, by `vite.config.js` | `localhost:8080`, by the site file |
| the backend runs | `./mvnw spring-boot:run` or the IDE | a container from the pipeline's image |
| the database | a container on the laptop | a container on the server, with a volume |
| a change becomes visible | on save | about two minutes after the push to `main` |

`localhost:8080` appears in `vite.config.js` only. A component that calls `http://localhost:8080/api/…` works on the laptop and fails on the server.

# With a name and HTTPS

Lab 2's optional step, `sudo certbot --nginx -d <name>`, edits this same site file: it adds a second `server` block for port 443 with the certificate, and turns the port 80 block into a redirect. The `root`, the `location /api` and the `location /` with `try_files` stay in the block that serves the site — certbot moves them, it does not remove them.

| Then | |
|---|---|
| `https://<name>/` | the page |
| `https://<name>/api/…` | the backend, through the same nginx |
| `http://<name>/…` | redirected to `https://` |
| the deploy jobs | unchanged: the files still go to `/var/www/html`, the backend still listens on 8080 |

One nginx on the server does the files, the `/api` forwarding and TLS.

# Relations, fetching and queries

How the backend loads related rows, what it costs, and how to say what you need in one query

# Where the department name comes from

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

```
@Entity
public class Employee {
    …
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;
}
```

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

The mapper reads `employee.getDepartment().getName()`. The `department` field was loaded **lazily**: what does that mean, and what does it cost when 500 employees are listed?

# Lazy loading

`FetchType.LAZY` on a relation: when Hibernate loads an `Employee`, it does **not** load the `Department`. The field holds a stand-in object, a **proxy**, that knows only the department's id, the value of the `department_id` column.

| Access | Happens |
|---|---|
| `employee.getDepartment().getId()` | no query: the id is already there |
| `employee.getDepartment().getName()` | the proxy runs `select … from department where id=?` and fills itself, once |
| the same access outside a transaction | no session to run the query in: `LazyInitializationException` — lecture 4's 500 "no session" |

`FetchType.EAGER`: the relation is loaded together with the entity.

| Annotation | Default |
|---|---|
| `@ManyToOne`, `@OneToOne` | EAGER |
| `@OneToMany`, `@ManyToMany` | LAZY |

The sample sets LAZY on `@ManyToOne` on purpose: the default would load a department every time any employee is loaded, needed or not.

**Rule:** write `fetch = FetchType.LAZY` on every `@ManyToOne` and `@OneToOne`, and ask for the data each query needs in the query itself. EAGER decides for every load path at once and still does not fix lists: Vlad Mihalcea, EAGER fetching is a code smell.

# The list, and its queries

Six employees in six departments. `GET /api/employees`, with `spring.jpa.show-sql=true`:

```
Hibernate: select e1_0.id,e1_0.department_id,e1_0.email,e1_0.name,e1_0.phone,e1_0.start_date 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=?
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=?
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=?
```

Seven queries for one request: one for the employees, then one per employee for its department, each run when the mapper reads `getName()`.

This is the **N+1 problem**: one query for a list of N rows, and N more for a relation of each row. The number of queries grows with the data, not with the code; 500 employees are 501 queries, each a round trip to the database.

# EAGER does not fix the list

The same request with `@ManyToOne(fetch = FetchType.EAGER)`:

```
GET /api/employees/3                                              1 query
Hibernate: select e1_0.id,d1_0.id,d1_0.name,e1_0.email,… from employee e1_0
           left join department d1_0 on d1_0.id=e1_0.department_id where e1_0.id=?

GET /api/employees                                                7 queries
Hibernate: select e1_0.id,e1_0.department_id,… from employee e1_0
Hibernate: select d1_0.id,d1_0.name from department d1_0 where d1_0.id=?
… (six times)
```

| Load | With EAGER |
|---|---|
| one entity by id, `findById` | Hibernate joins the relation into the one query |
| a list from a query, `findAll` or a query method | the query loads the employees; the relation is then loaded per row, as before, only earlier: at load instead of at first access |

EAGER decides *when*, not *how many*. The fix is in the query.

# The fix: fetch in the query

A query that says it wants the department with each employee:

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

    @Query("select e from Employee e join fetch e.department")
    List<Employee> findAllWithDepartment();
}
```

```
GET /api/employees                                                  1 query
Hibernate: select e1_0.id,d1_0.id,d1_0.name,e1_0.email,e1_0.name,e1_0.phone,e1_0.start_date
           from employee e1_0 join department d1_0 on d1_0.id=e1_0.department_id
```

| Part | Means |
|---|---|
| `@Query("…")` | the query is written out instead of derived from the method name |
| `select e from Employee e` | entities and their fields, not tables and columns: **JPQL** |
| `join fetch e.department` | join the relation **and load it** into each `Employee` |

Six departments arrive in the same result rows as the employees. The mapper's `getName()` finds them loaded; no further query.

# JPQL: the query language of entities

JPQL looks like SQL and is translated into SQL by Hibernate. It names what the Java code knows:

| | SQL | JPQL |
|---|---|---|
| from | a table: `employee` | an entity: `Employee` |
| columns | `e1_0.department_id` | fields: `e.department`, `e.department.name` |
| join | `join department d on d.id = e.department_id` | `join e.department d` — the condition is in the mapping |
| load the joined side | not a concept | `join fetch` |
| parameter | `?` | `:name`, filled from a method argument |
| result | rows | entities, or values, or new objects |

The method names of lecture 4, `findByDepartmentName`, are JPQL that Spring Data writes for you. `@Query` is for what a name cannot say: a fetch, a condition over several relations, an aggregate.

# The same with `@EntityGraph`

The second way to say "load this relation with the entity", without writing the query: an annotation on a derived or inherited method.

```
@EntityGraph(attributePaths = "department")
Page<Employee> findAll(Pageable pageable);
```

```
Hibernate: select e1_0.id,d1_0.id,d1_0.name,e1_0.email,… from employee e1_0
           left join department d1_0 on d1_0.id=e1_0.department_id
           order by e1_0.name offset ? rows fetch first ? rows only
```

`attributePaths` takes the **field names of the entity**, as written in the Java class, not column names: `"department"` is `Employee.department`. Several in braces, `{"department", "projects"}`; nested with a dot, `"department.manager"`. A name that is not a field fails at startup.

| | `join fetch` in `@Query` | `@EntityGraph` on a method |
|---|---|---|
| the query itself | written by you | derived from the name, or inherited like `findAll` |
| the join | inner: an employee without a department is left out | left: it is kept, with `department` null |
| combines with paging | with care | yes, as here |

# The collection side has the same problem

Departments with the number of their employees. First version: `department.getEmployees().size()` in the mapper.

```
GET /api/departments                                                    7 queries
Hibernate: select d1_0.id,d1_0.name from department d1_0
Hibernate: select e1_0.department_id,e1_0.id,e1_0.email,… from employee e1_0 where e1_0.department_id=?
… (six times)
```

One query for the departments, then one per department that loads **all its employees** to count them in Java. The lazy `@OneToMany` list is filled on `size()`.

The result wanted is six numbers, not six lists of entities.

# A query that returns what is needed: a projection

```
public interface DepartmentRepository extends JpaRepository<Department, Long> {

    @Query("""
            select new ee.taltech.staff.department.DepartmentDto(d.id, d.name, count(e))
            from Department d left join d.employees e
            group by d.id, d.name
            order by d.name""")
    List<DepartmentDto> findAllWithCounts();
}
```

```
GET /api/departments                                                1 query
Hibernate: select d1_0.id,d1_0.name,count(e1_0.id) from department d1_0
           left join employee e1_0 on d1_0.id=e1_0.department_id group by d1_0.id,d1_0.name order by d1_0.name
```

```
[{"id":2,"name":"Engineering","employeeCount":3},{"id":4,"name":"Finance","employeeCount":1},…
```

| Part | Means |
|---|---|
| `select new …DepartmentDto(d.id, d.name, count(e))` | build the DTO in the query; no entity is loaded |
| `left join d.employees e` | a department without employees stays in the result, with count 0 |
| `group by d.id, d.name` | one row per department; `count(e)` counts its employees |

The service returns the list as it is; no mapper runs.

# Four ways to load, and when

| Need | Use |
| --- | --- |
| a list of entities with one relation each | `join fetch` in `@Query`, or `@EntityGraph` |
| a list for display only: some fields, a count, a sum | a projection: `select new Dto(…)` |
| a filter by one or two fields | a derived method name: `findByDepartmentId` |
| one entity by id, then its relations | `findById`, and lazy loading inside the service's transaction |

The question before writing an endpoint that returns a list: how many queries does one request make, and does that number depend on the number of rows?

# `@Query` with parameters

```java
@Query("select e from Employee e join fetch e.department where lower(e.name) like lower(concat('%', :part, '%'))")
List<Employee> searchByName(@Param("part") String part);
```

```
GET /api/employees?name=an                                        1 query
Hibernate: select e1_0.id,d1_0.id,d1_0.name,e1_0.email,… from employee e1_0
           join department d1_0 on d1_0.id=e1_0.department_id
           where lower(e1_0.name) like lower(('%'||?||'%')) escape ''
[{"id":2,"name":"Jane Doe","email":"jane.doe@example.com","departmentName":"Engineering"}]
```

| Part | Means |
|---|---|
| `:part` | a named parameter; `@Param("part")` ties it to the argument |
| `concat('%', :part, '%')` | the `%` around the text are added in the query, not by the caller |
| `?` in the SQL | the value travels separately from the query text: no SQL injection through `name=` |

Lecture 4's `findByNameContainingIgnoreCase(name)` produces the same `where`; this version adds the `join fetch`.

# Native queries

```
@Query(value = "select * from employee where start_date > :since", nativeQuery = true)
List<Employee> startedAfter(@Param("since") LocalDate since);
```

`nativeQuery = true`: the text is SQL, sent to the database as it is.

| | JPQL | native SQL |
|---|---|---|
| names | entities and fields | tables and columns |
| checked against the mapping at startup | yes | no: a renamed column is found at run time |
| database-specific functions and syntax | the common subset | everything the database has |
| `join fetch` | yes | no |
| result mapped to an entity | always | when the columns match the entity |

For a report that needs a PostgreSQL function JPQL does not have. For everything else, JPQL, so that the mapping stays the one place where tables and columns are named.

# Paging

A list endpoint that returns every row returns 50 000 rows when the table has them. The database can return a slice, and Spring Data asks for one with a `Pageable`:

```
@GetMapping("/paged")
public PageDto<EmployeeDto> findPage(Pageable pageable) {
    return employeeService.findPage(pageable);
}
```

```
@EntityGraph(attributePaths = "department")
Page<Employee> findAll(Pageable pageable);
```

Paging limits rows; the department still comes from the entity graph, as on the list. Spring Data attaches the graph to the page query only, not to the count query.

```
GET /api/employees/paged?page=0&size=3&sort=name                2 queries
Hibernate: select … from employee e1_0 left join department d1_0 on … order by e1_0.name
           offset ? rows fetch first ? rows only
Hibernate: select count(e1_0.id) from employee e1_0
```

| Part | Means |
|---|---|
| `Pageable` as a controller argument | Spring fills it from `page`, `size` and `sort` in the address |
| `offset ? rows fetch first ? rows only` | the database returns three rows, starting at row 0 |
| the second query | the total, so that the client knows how many pages there are |

# What a page answers

```
public record PageDto<T>(List<T> items, int page, int totalPages, long totalElements) {}
```

```
{"items":[{"id":3,"name":"Alex Brown","email":"alex.brown@example.com","departmentName":"Support"},
          {"id":7,"name":"Chris Evans","email":"chris.evans@example.com","departmentName":"Engineering"},
          {"id":4,"name":"Emily Clark","email":"emily.clark@example.com","departmentName":"Finance"}],
 "page":0,"totalPages":3,"totalElements":8}
```

```
public PageDto<EmployeeDto> findPage(Pageable pageable) {
    Page<Employee> page = employeeRepository.findAll(pageable);
    return new PageDto<>(employeeMapper.toDtoList(page.getContent()),
            page.getNumber(), page.getTotalPages(), page.getTotalElements());
}
```

`Page<Employee>` is what the repository returns: the slice plus the counts. The service maps its content to DTOs and puts the four values the frontend needs into a record of its own; `Page` itself is not returned, because its JSON shape is Spring's and changes between versions.

Sorting and paging belong in the query. `findAll()` followed by `subList` in Java has already loaded every row.

# Seeing the queries of your own application

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

Every statement Hibernate sends is printed, with `Hibernate:` in front, as on these slides. The question to ask for every list endpoint: send one request, count the lines.

| Setting | Does |
|---|---|
| `spring.jpa.show-sql=true` | prints the SQL to standard output |
| `logging.level.org.hibernate.SQL=DEBUG` | the same through the logging system, with timestamps |
| `spring.jpa.properties.hibernate.format_sql=true` | one clause per line |

On the laptop, while developing. Not on the server: every statement of every request in the log.

# Why the mapping ran inside a transaction

The N+1 queries ran while the mapper read `getName()`. That worked because the sample has `spring.jpa.open-in-view=false` and the service is `@Transactional(readOnly = true)`: the mapping happens inside the service method, and the transaction, and its session, are still open.

| Where the lazy field is read | Result |
|---|---|
| in the service, inside `@Transactional` | a query, as on these slides |
| in the controller, after the service returned | `LazyInitializationException`: the session closed with the transaction |
| anywhere, with the relation fetched by the query | no query: the value is there |

A fetched relation is the only one of the three that does not depend on where the code runs.

# Cascade

An operation on an entity does not touch its relations unless the mapping says so. `cascade` on a relation says which operations follow it. The sample's new relation: notes of an employee, rows that exist only as part of that employee:

```
@OneToMany(mappedBy = "employee", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Note> notes = new ArrayList<>();
```

| Setting | Means |
|---|---|
| `CascadeType.PERSIST` | saving the employee saves its new notes |
| `CascadeType.REMOVE` | deleting the employee deletes its notes |
| `CascadeType.ALL` | every operation, the two above included |
| `orphanRemoval = true` | a note taken out of `employee.getNotes()` is deleted from the table, without a `delete` call |
| none (the default) | the notes are saved and deleted by themselves, through their own repository |

Cascade fits a relation whose other side exists only as part of this entity: notes of an employee, lines of an order. It does not fit `Department.employees`: deleting a department would delete people; without cascade the database's foreign key refuses the delete instead, as in lecture 4.

# Cascade: PERSIST and orphanRemoval, each run once

Each block is one `@Transactional` service method, with `spring.jpa.show-sql=true`:

```
PERSIST   e = new Employee(); e.getNotes().add(n1); e.getNotes().add(n2); employeeRepository.save(e)
Hibernate: insert into employee (department_id,email,name,phone,start_date) values (?,?,?,?,?)
Hibernate: insert into note (employee_id,text) values (?,?)
Hibernate: insert into note (employee_id,text) values (?,?)
```

```
orphanRemoval   e = employeeRepository.findById(id).orElseThrow(); e.getNotes().remove(0)
Hibernate: select e1_0.id,… from employee e1_0 where e1_0.id=?
Hibernate: select n1_0.employee_id,n1_0.id,n1_0.text from note n1_0 where n1_0.employee_id=?
Hibernate: delete from note where id=?
```

| Case | What happened |
|---|---|
| PERSIST | one `save`, three inserts; the notes were never saved by name |
| orphanRemoval | no `delete` call anywhere; the note that left the list is gone at commit |

# Cascade: REMOVE, and none

```
REMOVE   employeeRepository.deleteById(id)
Hibernate: select e1_0.id,… from employee e1_0 where e1_0.id=?
Hibernate: select n1_0.employee_id,n1_0.id,n1_0.text from note n1_0 where n1_0.employee_id=?
Hibernate: delete from employee_project where employee_id=?
Hibernate: delete from note where id=?
Hibernate: delete from employee where id=?
```

```
none   d = departmentRepository.findAll().get(0); d.getEmployees().add(new Employee(…)); departmentRepository.save(d)
Hibernate: select d1_0.id,d1_0.name from department d1_0
```

| Case | What happened |
|---|---|
| REMOVE | the notes and the join-table rows go first, then the employee; the foreign keys are satisfied in that order |
| none | **nothing**, and no error: `Department.employees` is the `mappedBy` side, Hibernate ignores what is added to it. The new employee is lost |

The last one is the mistake to know: adding to a `mappedBy` list writes nothing. Set the owning side, `employee.setDepartment(d)`, and save the employee.

# Entity states

Every entity object is in one of four states, and the state decides whether Hibernate writes it:

| State | The object | Changes to it |
|---|---|---|
| **transient** | `new Employee()`, never saved | nothing happens until `save` |
| **managed** | loaded in the current transaction, or returned by `save` | written at commit — lecture 4's dirty checking |
| **detached** | was managed; the transaction ended | nothing; `save` makes a managed copy |
| **removed** | `delete` was called | the row is deleted at commit |

```
Employee e = employeeRepository.findById(id).orElseThrow();   // managed
e.setName(dto.name());                                          // written at commit, no save
```

A lazy field of a detached object cannot load: `LazyInitializationException`, the "no session" 500. A managed object's changes are written whether `save` is called or not.

# The transaction is the boundary

What decides between *managed* and *detached* is `@Transactional` on the service method. The same two lines, run twice:

```
@Transactional                                                  // one transaction for the whole method
public void rename(Long id) {
    Employee e = employeeRepository.findById(id).orElseThrow();   // managed until the method returns
    e.setName("Alex Brown-Smith");
}                                                               // return = commit: dirty checking runs here
```

```
Hibernate: select e1_0.id,… from employee e1_0 where e1_0.id=?
Hibernate: update employee set department_id=?,email=?,name=?,phone=?,start_date=? where id=?
```

```
public void rename(Long id) {                                   // no @Transactional
    Employee e = employeeRepository.findById(id).orElseThrow();   // the repository call had its own transaction; it is over
    e.setName("Alex Brown-Jones");                                 // a detached object: a plain Java field changes
}
```

```
Hibernate: select e1_0.id,… from employee e1_0 where e1_0.id=?
```

No `update`, no error; the next `GET` still shows the old name. That is the "my update does not persist" bug.

# Transient, managed, detached: what to do with each

| Object | Why | What to do |
| --- | --- | --- |
| transient, `new Employee()` | nobody is tracking it | `save` it: that is the `insert`, and the returned object is managed |
| managed, loaded in this transaction | Hibernate tracks it | change it; the `update` comes at commit. `save` on it is harmless and adds nothing |
| detached, loaded in a transaction that ended | tracking stopped | load it again inside `@Transactional`, or `save` it: a managed copy, then an `update` of every column |

# In one table

| Thing | Where | Does |
|---|---|---|
| `dist/` → `/var/www/html` with `rsync` | `deploy-frontend` | the built files into the server's nginx |
| `try_files $uri $uri/ /index.html` | the site file on the server | a router path reloaded in the browser gets the application |
| `build-frontend`, `deploy-frontend` | `.gitlab-ci.yml` | `npm ci && npm run build`; `rsync` of the artifact |
| `sudo chown -R ubuntu:ubuntu /var/www/html` | the server, once | lets the deploy key's user write the files |
| `FetchType.LAZY` / `EAGER` | a relation | when the other side is loaded; not how many queries |
| `@Query("… join fetch e.department")` | repository | the relation in the same query |
| `@EntityGraph(attributePaths = "…")` | repository method | the same, on a derived or inherited method |
| `select new Dto(…) … group by` | `@Query` | a projection: values, no entities |
| `Pageable`, `Page`, `offset … fetch first` | controller, repository, SQL | a slice and a count |
| `spring.jpa.show-sql=true` | `application.properties`, on the laptop | count the queries of one request |
| `cascade = ALL, orphanRemoval = true` | a `@OneToMany` whose rows exist only as part of this entity | save, delete and list removals follow the parent |
| `@Transactional` | the service method | managed until return; the `update` of a changed entity comes at commit |



Questions