Veebirakenduse projekt (ITI0302)

Loeng 7

Errors and status codes · validation · logging · OpenAPI · tests · Siim Rebane

What the frontend gets when something goes wrong

The frontend's `fetch` gets a status code and a body, and decides from them what to show. The backend of lecture 6, two requests that are wrong in a normal way:

```
$ curl -i -X POST localhost:8087/api/employees -H 'Content-Type: application/json' -d '{"name":"Mari Maasikas","email":"mari@example.com"}'
HTTP/1.1 500
{"timestamp":"2026-10-15T02:32:49.723Z","status":500,"error":"Internal Server Error","path":"/api/employees"}

$ curl -i -X POST localhost:8087/api/employees -H 'Content-Type: application/json' -d '{"name":"","email":"not-an-email","departmentId":1}'
HTTP/1.1 201
{"id":4,"name":"","email":"not-an-email","departmentName":"Sales"}
```

A missing field is answered as a server crash. An empty name and a wrong e-mail are stored and answered as success. The frontend cannot tell its user what to fix in either case, and the database now holds a row that nothing can use.

Status codes an API uses

Code	Meaning	Whose fault
`200 OK`	done, here is the result	—
`201 Created`	done, a new row exists; its representation is the body	—
`204 No Content`	done, nothing to return (a delete)	—
`400 Bad Request`	the request itself is wrong: not JSON, a field missing or invalid	client
`404 Not Found`	the thing the path or body names does not exist	client
`409 Conflict`	the request contradicts existing data: a duplicate, a row still referenced	client
`500 Internal Server Error`	the backend failed; the request may have been fine	server

The `4xx` family says "change your request", the `5xx` family says "we broke". A frontend shows a `4xx` message to the user and reports a `5xx` as "try again later". That only works when the backend chooses the codes on purpose.

What a 500 is

A `500` is an exception that nobody caught. The `POST` without a `departmentId`, in the backend's log:

```
ERROR 65820 --- [nio-8087-exec-7] o.a.c.c.C.[.[.[/].[dispatcherServlet] : Servlet.service() for servlet [dispatcherServlet] threw exception

java.lang.IllegalArgumentException: The given id must not be null
        at org.springframework.util.Assert.notNull(Assert.java:182)
        at org.springframework.data.jpa.repository.support.SimpleJpaRepository.findById(SimpleJpaRepository.java:330)
        ...
        at ee.taltech.staff.employee.EmployeeService.create(EmployeeService.java:46)
```

`departmentId` was missing, so the service called `findById(null)`, Spring Data threw, nothing caught it, and Spring's default error page turned it into a `500` with a body that names no cause.

The developer finds the cause in the log. The user gets "Internal Server Error" for forgetting a field.

Exceptions in Java

An **exception** is an object that a method throws instead of returning. It travels up the call stack until a `catch` takes it; if none does, the thread ends with the stack trace above.

```
throw new IllegalArgumentException("The given id must not be null");   // throw

try { ... } catch (IllegalArgumentException e) { ... }                  // catch
```

Two kinds. A **checked** exception (`IOException` , `SQLException`) must be declared with `throws` or caught where it occurs. An **unchecked** exception extends `RuntimeException` and needs neither; it passes through every method on its way up.

Spring and JPA throw unchecked exceptions only. An application's own exceptions are unchecked too: the code that detects a problem throws, and the code that knows how to answer the client is far away, in one place.

Where an error is decided

A request `GET /api/employees/99` goes controller → service → repository. The repository returns an empty `Optional`. Who turns that into a `404`?

- the **repository** knows nothing about employees as a concept, only rows
- the **service** knows that the request named an employee that does not exist; it does not know HTTP
- the **controller** knows HTTP, but there are many controllers and many services

So the service throws an exception that says what happened in the application's own words, and nobody catches it on the way. One class, next to the controllers, turns every such exception into a status code and a body. The controllers stay free of `try` / `catch`.

The path of an exception through a request

Every request enters Spring through one servlet, the **DispatcherServlet**: it reads the URL, calls the matching controller method, and turns the return value into the answer. It also surrounds that call with a `catch`.

```
service throws NotFoundException("employee 99 not found")
  → the service method ends, nothing caught
  → the controller method ends, nothing caught
  → the exception reaches the DispatcherServlet
  → Spring looks in every @RestControllerAdvice class for an @ExceptionHandler of that type
      found: that method runs; its return value is the body, its @ResponseStatus the code
      none:  Spring's own error answer, status 500, a generic body
```

The most specific handler wins: one for `NotFoundException` before one for `Exception`. So throwing is enough. The code that detects the problem throws; the code that knows how to answer sits in one place; nothing in between needs to know.

A custom exception

`ee.taltech.staff.common.NotFoundException`:

```
/** Thrown by a service when the row a request names does not exist. Answered with 404. */
public class NotFoundException extends RuntimeException {

    public NotFoundException(String message) {
        super(message);
    }
}
```

It extends `RuntimeException`, so no method has to declare it. Its only content is the message. The class name is the category, "not found"; the message is the instance, "employee 99 not found".

Three or four such classes cover a whole project: not found, conflict, forbidden. A class per endpoint is too many; one class for everything cannot be mapped to codes.

The service throws it

`EmployeeService`, the three places that used to return nothing useful:

```
public EmployeeDto findById(Long id) {
    return employeeRepository.findById(id)
            .map(employeeMapper::toDto)
            .orElseThrow(() -> new NotFoundException("employee " + id + " not found"));
}

@Transactional
public EmployeeDto create(EmployeeCreateDto dto) {
    Department department = departmentRepository.findById(dto.departmentId())
            .orElseThrow(() -> new NotFoundException("department " + dto.departmentId() + " not found"));
    ...
}

@Transactional
public void delete(Long id) {
    if (!employeeRepository.existsById(id)) {
        throw new NotFoundException("employee " + id + " not found");
    }
    employeeRepository.deleteById(id);
}
```

`Optional.orElseThrow` takes the function that makes the exception. The message says which thing with which id, because that is what the frontend will print.

Where each check belongs

Check	Where	How	Answer
the request has the right shape: fields present, types, formats, lengths	controller	validation rules written on the DTO, checked by Spring before the method runs	`400`, every broken field named
the request makes sense against the data: the department exists, the e-mail is free, the state allows it	service	the service reads what it needs and throws the application's own exception	`404`, `409`, `403`
a rule the database enforces no matter who writes: unique, not null, foreign key	database	a constraint in a Liquibase changeset	`409`
anything nobody foresaw	handler	the handler for `Exception`	`500`, and a bug to fix

The controller validates shape and nothing else: it never looks into the database. The service validates meaning and never looks at HTTP. The database is the last line, because two requests at the same moment both pass the service's check and only the constraint sees them together.

One body for every error

The frontend reads every error the same way, so every error has the same shape. `ee.taltech.staff.common.ApiError`:

```
@JsonInclude(JsonInclude.Include.NON_NULL)
public record ApiError(
        int status,
        String message,
        String path,
        Instant timestamp,
        Map<String, String> errors) {

    public static ApiError of(int status, String message, String path) {
        return new ApiError(status, message, path, Instant.now(), null);
    }
}
```

`status` repeats the code so the body is complete on its own. `message` is for a human. `errors` is filled only by validation, one entry per field; `@JsonInclude(NON_NULL)` leaves it out of the JSON otherwise.

The global handler

`ee.taltech.staff.common.GlobalExceptionHandler`, the part for `NotFoundException`:

```
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(NotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ApiError notFound(NotFoundException e, HttpServletRequest request) {
        return ApiError.of(404, e.getMessage(), request.getRequestURI());
    }
}
```

- `@RestControllerAdvice`: this class applies to every `@RestController`
- `@ExceptionHandler(NotFoundException.class)`: this method runs when a controller lets that exception out
- `@ResponseStatus`: the status code of the answer
- the return value becomes the JSON body, as in a controller

One class, one method per exception type, no `try` / `catch` anywhere else.

404 after the handler

```
$ curl -s -i localhost:8087/api/employees/99
HTTP/1.1 404
Content-Type: application/json

{"status":404,"message":"employee 99 not found","path":"/api/employees/99","timestamp":"2026-10-15T02:36:33.076617Z"}
```

```
$ curl -s -i -X POST localhost:8087/api/employees -H 'Content-Type: application/json' -d '{"name":"Mari Maasikas","email":"mari@example.com","departmentId":9}'
HTTP/1.1 404
Content-Type: application/json

{"status":404,"message":"department 9 not found","path":"/api/employees","timestamp":"2026-10-15T02:36:33.242320Z"}
```

Same shape, two services, two messages. The frontend prints `message` and is right both times.

What an error body must not contain

An attacker reads error answers on purpose. Each of these tells them something to use:

Never in the body	What it gives away
a stack trace	library names, versions and line numbers: which known vulnerabilities apply
the exception class, `SQLException`, `PSQLException`	the database product
SQL text, table, column or constraint names	the schema, for the queries they will try
file paths, host names, internal addresses	where the application runs

The body gets the status, a message in the application's words, and the DTO's field names. Everything above goes into the log, with the request that caused it, where the developer reads it.

Spring's default error body leaves the trace out but can include the exception message. The handler for `Exception` writes the body itself, so nothing leaks by default.

A 500 is a bug to fix

```
@ExceptionHandler(Exception.class)
@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
public ApiError unexpected(Exception e, HttpServletRequest request) {
    log.error("unhandled exception for {} {}", request.getMethod(), request.getRequestURI(), e);
    return ApiError.of(500, "internal error", request.getRequestURI());
}
```

A `500` is never the right answer to a request a user can make. Every `500` in the log is a case the developer did not think of; the fix is to find which `4xx` it really is and answer that, with the reason. The `POST` without a `departmentId` became a `400` through validation. A sort by a field that does not exist throws `PropertyReferenceException`; one more handler method, and:

```
$ curl -s -i 'localhost:8087/api/employees/paged?sort=nosuchfield'
HTTP/1.1 400
{"status":400,"message":"unknown sort field: nosuchfield","path":"/api/employees/paged","timestamp":"…"}
```

The net stays for the next unknown case. The log line with the request is how it gets found.

Before and after

Request	Lecture 6 backend	With the handler
`GET /api/employees/99`	`404`, Spring's body: `"error":"Not Found"`	`404`, `"message":"employee 99 not found"`
`POST`, `departmentId` missing	`500`, stack trace in the log	`400`, `"errors":{"departmentId":"must not be null"}`
`POST`, department 9	`400`, `"no such department"`	`404`, `"message":"department 9 not found"`
`POST`, empty name	`201`, the row is stored	`400`, `"errors":{"name":"must not be blank"}`
`?sort=nosuchfield`	`500`, stack trace in the log	`400`, `"message":"unknown sort field: nosuchfield"`
a case nobody foresaw	`500`, stack trace in the log	`500`, generic body, one `ERROR` line with the request, a to-do

The same five classes, `NotFoundException`, `ApiError`, `GlobalExceptionHandler` and the DTOs with their rules, give every endpoint of the project this behaviour.

Problem Details, a standard error body

Every API team invents its own error shape, as `ApiError` is one, and every client has to learn each. A published standard, RFC 9457 "Problem Details for HTTP APIs" (an RFC is an internet standard, the series HTTP itself is defined in), gives one shape with a content type of its own, `application/problem+json`, so a client can read any API's errors the same way. The sample's 404 in that shape:


```
{"type":"about:blank","title":"Not Found","status":404,"detail":"employee 99 not found","instance":"/api/employees/99"}
```


Field	Meaning	In `ApiError`
`type`	a URI that documents this kind of error; `about:blank` = no specific kind	—
`title`	the name of the kind, the same for every 404	—
`status`	the status code, repeated	`status`
`detail`	what happened in this one case	`message`
`instance`	the request this is about	`path`

Spring has a `ProblemDetail` class a handler can return. The course does not require it: one frontend, written by the same team, reads the body. The name is worth knowing for the day an API you call answers with `application/problem+json`.

Data the backend must not accept

```
$ curl -i -X POST localhost:8087/api/employees -H 'Content-Type: application/json' -d '{"name":"","email":"not-an-email","departmentId":1}'
HTTP/1.1 201
{"id":4,"name":"","email":"not-an-email","departmentName":"Sales"}
```

The frontend has a form. A form can be submitted empty, with a space in the e-mail, with a negative salary, from a browser where someone edited the HTML, or from `curl` with no form at all. The backend is the only place every request passes.

Validation is the check that the data of a request obeys the rules of the application before anything is done with it, and the answer that names every broken rule at once, so that the form can show all of them.

Rules of the kind "the name is not empty", "the e-mail looks like an e-mail", "the amount is positive" belong on the DTO that the controller receives.

Bean Validation

Bean Validation (Jakarta Validation) is the Java standard for declaring rules as annotations on fields and checking an object against them. **Hibernate Validator** is the implementation Spring Boot uses; it has nothing to do with Hibernate ORM beyond the name.

The dependency has been in the project since lecture 4 (`pom.xml` or `build.gradle`):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

The rule is an annotation on the field. The check runs where the object enters the application: in the controller, on the request body, when the parameter is marked `@Valid`. Without that mark the annotations are decoration.

The annotations

Annotation	Passes when	Note
`@NotNull`	the value is not `null`	an empty string passes
`@NotBlank`	a string with at least one non-space character	the one for names
`@NotEmpty`	a string or collection with at least one element	`" "` passes
`@Size(min, max)`	length or element count in the range	strings, lists
`@Email`	the string looks like an e-mail address	`null` passes
`@Min(1)`, `@Max(5)`, `@Positive`	numbers in range	`null` passes
`@Past`, `@Future`, `@PastOrPresent`	dates relative to now	`LocalDate`, `Instant`
`@Pattern(regexp = "…")`	the string matches	phone numbers, codes

Every annotation except the `@Not…` ones accepts `null`: "if there is a value, it must look like this". A required field needs `@NotNull` or `@NotBlank` in addition.

The DTO of this sample

```
public record EmployeeCreateDto(
        @NotBlank @Size(max = 100) String name,
        @NotBlank @Email String email,
        @NotNull Long departmentId) {
}
```

```
public record DepartmentCreateDto(@NotBlank @Size(max = 100) String name) {
}
```

On a record the annotations go on the components. A name must be present and at most 100 characters, because the column is `text` and the screen is not infinite. An e-mail must be present and well-formed. A department id must be present; whether that department exists is a question for the service and the database, not for validation.

The rules sit on the create DTO, the one that comes in. `EmployeeDto`, the one that goes out, has none: the backend made it.

`@Valid` in the controller

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

@PutMapping("/{id}")
public EmployeeDto update(@PathVariable Long id, @Valid @RequestBody EmployeeCreateDto dto) {
    return employeeService.update(id, dto);
}
```

`@Valid` tells Spring to check the body against its annotations after reading the JSON and before calling the method. If a rule fails, the method is never called: Spring throws `MethodArgumentNotValidException` with the list of field errors.

That exception is caught like any other, by the global handler.

400 with every broken field

The handler's method for it collects the field errors into the `errors` map:

```
@ExceptionHandler(MethodArgumentNotValidException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
public ApiError invalid(MethodArgumentNotValidException e, HttpServletRequest request) {
    Map<String, String> errors = new LinkedHashMap<>();
    e.getBindingResult().getFieldErrors()
            .forEach(f -> errors.put(f.getField(), f.getDefaultMessage()));
    return new ApiError(400, "validation failed", request.getRequestURI(), Instant.now(), errors);
}
```

```
$ curl -s -i -X POST localhost:8087/api/employees -H 'Content-Type: application/json' -d '{"name":"","email":"not-an-email","departmentId":1}'
HTTP/1.1 400

{"status":400,"message":"validation failed","path":"/api/employees","timestamp":"2026-10-15T02:36:33.189396Z",
 "errors":{"name":"must not be blank","email":"must be a well-formed email address"}}
```

Both fields in one answer. The form marks both inputs. The default messages come from Hibernate Validator; `@NotBlank(message = "name is required")` replaces one.

The missing field, and the body that is not JSON

The `POST` without a `departmentId`, the one the lecture 6 backend answered with a `500`:

```
$ curl -s -i -X POST localhost:8087/api/employees -H 'Content-Type: application/json' -d '{"name":"Mari Maasikas","email":"mari@example.com"}'
HTTP/1.1 400
{"status":400,"message":"validation failed","path":"/api/employees","timestamp":"…","errors":{"departmentId":"must not be null"}}
```

A body that cannot be parsed never reaches validation; Spring throws `HttpMessageNotReadableException` while reading it. One more handler method, same shape:

```
@ExceptionHandler(HttpMessageNotReadableException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
public ApiError unreadable(HttpMessageNotReadableException e, HttpServletRequest request) {
    return ApiError.of(400, "request body is not valid JSON", request.getRequestURI());
}
```

```
$ curl -s -i -X POST localhost:8087/api/employees -H 'Content-Type: application/json' -d '{"name": "Mari'
HTTP/1.1 400
{"status":400,"message":"request body is not valid JSON","path":"/api/employees","timestamp":"…"}
```

Why the backend writes a log

The backend runs on the server in a container. Nobody watches it. When a user says "it did not work yesterday at four", the only witness is what the application wrote down.

`System.out.println` is the wrong tool for that, three times over:

- the line has no time, no level, no origin: nobody knows when it happened or which class wrote it
- every developer writes a different line, so the output is a mix of styles that no tool can read
- it cannot be turned off, made quieter or redirected without changing the code and redeploying

Logging is writing through one library that every class uses the same way, so that every line looks the same and is written, filtered and formatted by configuration in one place.

One format for every line

The sample's output on the server, written by two different classes:

```
2026-10-15T05:36:33.296+03:00  INFO 69957 --- [staff] [io-8087-exec-10] e.t.staff.employee.EmployeeService       : employee 7 created in department Sales
2026-10-15T05:36:33.362+03:00 ERROR 69957 --- [staff] [nio-8087-exec-4] e.t.staff.common.GlobalExceptionHandler  : unhandled exception for GET /api/employees/paged

org.springframework.data.core.PropertyReferenceException: No property 'nosuchfield' found for type 'Employee'
        at org.springframework.data.core.PropertyPath.<init>(PropertyPath.java:91)
        ...
```

Every line has the same fields in the same order: time with zone, level, process id, application name, thread, the class that wrote it, the message. The stack trace of an exception comes under its line. None of the classes wrote any of that; the library did.

The format is one setting. `logging.pattern.console` changes it for every line of the application at once, and `logging.structured.format.console=ecs` makes every line a JSON object for a log collector, with no change in any class.

SLF4J, Logback, `@Slf4j`

SLF4J (Simple Logging Facade for Java) is the interface the code calls: `log.info(...)`. **Logback** is the implementation behind it that Spring Boot ships and configures; the code never names it. Both are already in the project through `spring-boot-starter-webmvc`.

A logger is one object per class. Lombok's `@Slf4j` on a class writes the field:

```
@Service
@RequiredArgsConstructor
@Slf4j
public class EmployeeService {
    // Lombok generated: private static final Logger log = LoggerFactory.getLogger(EmployeeService.class);
```

The class name in every line comes from that `getLogger` argument. That is why the field is per class and not shared.

Levels

Level	What belongs there
`ERROR`	something failed and someone has to look: an unhandled exception, a call to another system that gave up
`WARN`	something is wrong but the request was served: a retry, a deprecated path, a constraint hit
`INFO`	the facts of normal operation: started, an employee created, a job ran
`DEBUG`	details for a developer: the parameters of a request, the branch taken
`TRACE`	everything: every SQL bind value, every loop step

A level is a threshold: with the threshold at `INFO`, the lines of `INFO`, `WARN` and `ERROR` are written and the lines of `DEBUG` and `TRACE` are not. Spring Boot's default threshold is `INFO`, on the laptop and on the server alike. Levels are set per package, so the application's own `DEBUG` can be turned on while Hibernate's stays off.

The code

```
Employee saved = employeeRepository.save(employee);
log.info("employee {} created in department {}", saved.getId(), department.getName());
```

```
log.error("unhandled exception for {} {}", request.getMethod(), request.getRequestURI(), e);
```

- `{}` is a placeholder; the arguments fill it. Not `"employee " + id + " created"` : the string is built only if the level is on, and the message stays one searchable sentence
- an exception as the **last** argument, after the placeholders are filled, prints its stack trace under the line
- a log line is a sentence about the application: what happened, to which id; never the whole object

Each event is logged **once**, at the one place that knows it. Not a line in the controller, another in the service and a third in the handler for the same request: with real traffic every extra line multiplies, and the one line that matters is buried. The same for errors: the code that meets an exception does not catch it, log it and throw it again; the handler logs it once, with the request and the stack trace. Reads are not events: a line on every `GET` is noise.

Levels in the configuration

`application.properties`:

```
logging.level.root=INFO
logging.level.ee.taltech.staff=DEBUG
logging.level.org.hibernate.SQL=DEBUG
```

- `root` is the default for every package
- a package name sets its own level and everything under it
- `org.hibernate.SQL` at `DEBUG` prints every statement, as `spring.jpa.show-sql` did in lecture 6, through the log with time stamps instead of to the console

On the server the same keys are environment variables, as every Spring property is: `LOGGING_LEVEL_EE_TALTECH_STAFF=DEBUG` in `compose.yaml` turns the application's details on for one restart without a build.

What never goes into a log

A log is read by every developer, kept in the pipeline's job output, pasted into tickets and chats. It is not a private place.

- **passwords and tokens**: not the request body of a login, not the `Authorization` header, not the database URL with its password
- **personal data beyond the need**: an id identifies the row; the name, e-mail and phone of a person do not belong in a line about a request
- **whole objects**: `log.info("saved {}", employee)` prints every field, now and after every field anyone adds later

A line names what happened and the id it happened to. Everything else is in the database already.

OpenAPI

- a description of a REST API in a standard JSON or YAML format: every path, its operations, parameters, request and response bodies, status codes
- the frontend developer reads it instead of the Java code or a chat message; the backend developer does not write it by hand
- generated from the code, so it is always the same as the code
- old name: Swagger. Swagger UI is the page that shows the document and lets you call each operation
- the other direction exists too: some teams agree on the document first and generate the code from it, with OpenAPI Generator (formerly swagger-codegen): controller interfaces for the backend, a client for the frontend
- version 3.1 is current

springdoc

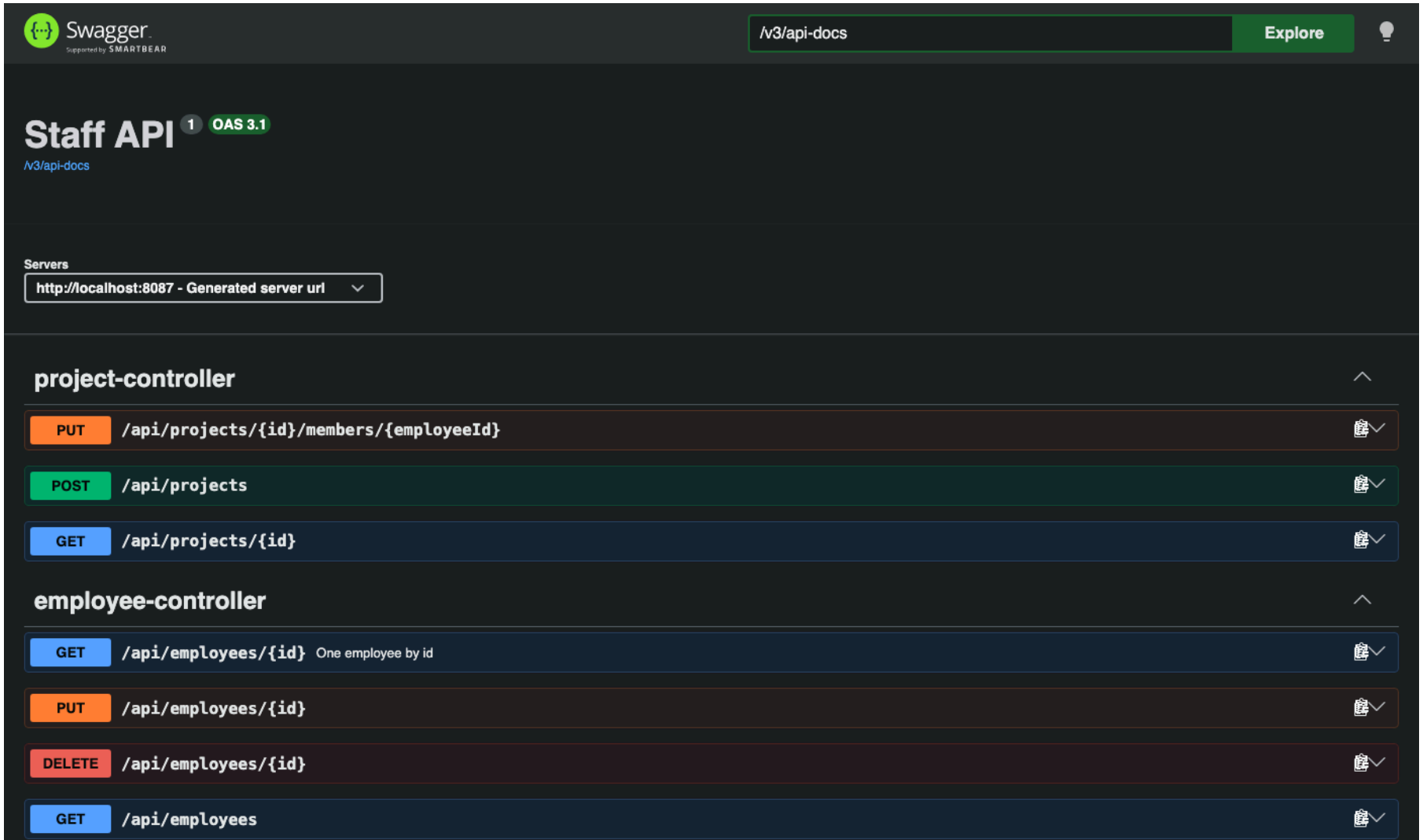
- the library that generates the OpenAPI document from a running Spring application
- one dependency, no configuration:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>3.1.1</version>
</dependency>
```

- after that, with the application running:
  - `http://localhost:8080/swagger-ui.html`, the page
  - `http://localhost:8080/v3/api-docs`, the document
- guide: https://springdoc.org/

Swagger UI

The sample, every controller as a group, every operation openable and callable with "Try it out":



What springdoc reads from the code

In the code	In the document
`@RestController`, `@GetMapping("/{id}")`, `@PathVariable`, `@RequestParam`	paths, operations, parameters
parameter and return types: `EmployeeCreateDto`, `EmployeeDto`, `List<…>`	schemas
`@NotBlank`, `@Size(max = 100)`, `@Email`	`required`, `maxLength`, `format: email`
`@ResponseStatus` on the global handler's methods	the status codes of every operation

`components.schemas.EmployeeCreateDto` in the sample's document:

```
{"type": "object",
 "properties": {"name":         {"type": "string", "maxLength": 100, "minLength": 0},
                "email":        {"type": "string", "format": "email", "minLength": 1},
                "departmentId": {"type": "integer", "format": "int64"}},
 "required": ["departmentId", "email", "name"]}
```

Annotations

- `@OpenAPIDefinition`
  - purpose: title and version of the API
  - usage: on the application class
- `@Tag`
  - purpose: group the operations of one controller under a name
  - usage: on the controller class
- `@Operation`
  - purpose: describe what an operation does
  - usage: on the controller method
- `@ApiResponse`
  - purpose: describe one status code of an operation
  - usage: on the controller method
- `@Schema`
  - purpose: describe a DTO or one of its fields, with an example
  - usage: on the DTO class or field

Annotation sample

```
@SpringBootApplication
@OpenAPIDefinition(info = @Info(title = "Staff API", version = "1"))
public class StaffApplication { ... }
```

```
@Operation(summary = "One employee by id", description = "404 when there is no employee with that id.")
@GetMapping("/{id}")
public EmployeeDto findById(@PathVariable Long id) { ... }
```

The operation in the generated document:

```
{"summary": "One employee by id",
 "description": "404 when there is no employee with that id.",
 "operationId": "findById",
 "parameters": [{"name": "id", "in": "path", "required": true, "schema": {"type": "integer", "format": "int64"}}],
 "responses": 200 (EmployeeDto), 400, 404, 409, 500 (ApiError)}
```

- the status codes come from the global handler's `@ResponseStatus` annotations, not from this method
- `springdoc.swagger-ui.enabled=false` hides the page on the server; `/v3/api-docs` stays

Testing a web application

- a test runs a piece of code with known input and fails when the output is not what was expected; you have written them before
- the problem with a web application: nothing runs alone
  - the service needs the repository, the repository needs JPA and a database
  - the controller needs Spring to receive the request and turn the body into a DTO
  - some code needs another system: a mail server, a payment API
- two ways out, and a project uses both:
  - **unit test**: one class alone; every dependency is replaced by a stand-in that answers what the test tells it to. No Spring, no database. Milliseconds; a failure points at that one class
  - **integration test**: the application is started, with the real controller, service, JPA and PostgreSQL, and gets a request as a client would. Seconds; a failure says the parts do not fit: URL, JSON, SQL, status code
- the pipeline's `build-backend` job runs the build, `./mvnw package` or `./gradlew build`, which runs every test before it packages the jar; a failing test fails the job and the merge request cannot merge

Where tests live

```
backend/src/main/java/ee/taltech/staff/employee/EmployeeService.java
backend/src/test/java/ee/taltech/staff/employee/EmployeeServiceTest.java
backend/src/test/java/ee/taltech/staff/employee/EmployeeControllerIntegrationTest.java
```

- `src/test/java`, the same package as the class under test; the same layout in Maven and Gradle
- a class per class under test, named `…Test`; the build runs every class under `src/test/java` whose name ends in `Test` or `Tests`
- the test's package is the class's package, so package-private methods and fields are reachable

The test starters in the project since lecture 4 bring everything the tests need: JUnit 6 to run them, AssertJ to assert, Mockito for the stand-ins, Spring's `MockMvc` for requests.

A JUnit test is a method

JUnit is the framework that finds and runs tests. A test is a method with `@Test` in a class in `src/test/java`. It passes when it returns, and fails when it throws; an assertion is a method that throws when a condition does not hold.

```
class EmployeeServiceTest {

    @Test
    void findById_returnsTheDto_whenTheEmployeeExists() {
        ...
        assertThat(result.name()).isEqualTo("Jane Doe");
    }
}
```

`assertThat(…)` is **AssertJ**: `isEqualTo`, `isNull`, `hasSize`, `contains`, `isInstanceOf`, and a readable message when it fails. The method name is a sentence: what is called, what it does, under which condition. It is what the report shows when it goes red.

Testing a service without a database

```
public class EmployeeService {
    private final EmployeeRepository employeeRepository;
    private final DepartmentRepository departmentRepository;
    private final EmployeeMapper employeeMapper;
```

- to construct the service, the test needs those three
- the repositories need JPA and a database: they are replaced
- the mapper is generated code with no dependencies: the real one is used, so a wrong mapping fails the test
- a **mock** is an object of the required type whose every method does nothing and returns `null`, until the test says "when `findById(7)` is called, return this"; it records every call, so the test can ask "was `save` called?"
- **Mockito** is the library that makes mocks from interfaces and classes

Mockito in a JUnit test

```
@ExtendWith(MockitoExtension.class)
class EmployeeServiceTest {

    @Mock
    EmployeeRepository employeeRepository;

    @Mock
    DepartmentRepository departmentRepository;

    @Spy
    EmployeeMapper employeeMapper = Mappers.getMapper(EmployeeMapper.class);

    @InjectMocks
    EmployeeService employeeService;
```

- `@ExtendWith(MockitoExtension.class)` : Mockito fills the annotated fields before each test
- `@Mock` : this field is a stand-in of that type
- `@Spy` : this field is a real object; `Mappers.getMapper` gives the class MapStruct generated
- `@InjectMocks` : construct this one for real and pass the mocks and spies to its constructor, as Spring passes the real beans
- no `@SpringBootTest`, no context, no database

Given, when, then

```
@Test
void findById_returnsTheDto_whenTheEmployeeExists() {
    // given
    Department engineering = new Department();
    engineering.setName("Engineering");
    Employee employee = new Employee();
    employee.setId(7L);
    employee.setName("Jane Doe");
    employee.setDepartment(engineering);
    when(employeeRepository.findById(7L)).thenReturn(Optional.of(employee));

    // when
    EmployeeDto result = employeeService.findById(7L);

    // then
    assertThat(result.name()).isEqualTo("Jane Doe");
    assertThat(result.departmentName()).isEqualTo("Engineering");
}
```

- **given** the state: what the mocks answer; `when(…).thenReturn(…)` is "this call gives this"; a call the test did not describe returns `null` or an empty `Optional`
- **when**: the one call under test
- **then**: the assertions. `departmentName` came through the real mapper's `@Mapping(source = "department.name")`

Testing the exception

```
@Test
void findById_throwsNotFound_whenThereIsNoSuchEmployee() {
    when(employeeRepository.findById(99L)).thenReturn(Optional.empty());

    assertThatThrownBy(() -> employeeService.findById(99L))
            .isInstanceOf(NotFoundException.class)
            .hasMessage("employee 99 not found");
}
```

`assertThatThrownBy` runs the lambda and fails the test if nothing is thrown; the chain then checks the type and the message.

This is the test of the lecture's first part: the service's answer to a missing row is a `NotFoundException` with the message the frontend will show. It runs without the handler, without HTTP, in a millisecond.

Checking that a method was not called

```
@Test
void create_doesNotSave_whenTheDepartmentDoesNotExist() {
    when(departmentRepository.findById(5L)).thenReturn(Optional.empty());
    EmployeeCreateDto dto = new EmployeeCreateDto("Mari Maasikas", "mari@example.com", 5L);

    assertThatThrownBy(() -> employeeService.create(dto))
            .isInstanceOf(NotFoundException.class)
            .hasMessage("department 5 not found");
    verify(employeeRepository, never()).save(any());
}
```

`verify(mock, never()).save(any())` fails the test if `save` was called with anything. The rule under test is not only "it throws" but "it throws before writing". Mocks record calls, which is what makes that checkable.

The fourth test of the class checks the happy path of `create`: with `when(employeeRepository.save(any())).thenAnswer(call -> call.getArgument(0))` the mock returns what it was given, and the assertion is that the DTO names the department, which the real mapper read from the entity.

Integration test

- the whole application starts and gets requests as a client would send them: controller, service, JPA, PostgreSQL
- each test run wants the database in a known state: an empty database with the schema and a few known rows
- **Testcontainers** starts a PostgreSQL container for the test run and removes it afterwards; it needs Docker on the laptop, and in the pipeline the build job has it too
- dependencies, Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-testcontainers</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers-postgresql</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers-junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>
```

- Gradle:

```
testImplementation 'org.springframework.boot:spring-boot-testcontainers'
testImplementation 'org.testcontainers:testcontainers-postgresql'
testImplementation 'org.testcontainers:testcontainers-junit-jupiter'
```

- no versions: Spring Boot manages them (Testcontainers 2.0.5 with Boot 4.1.1)

Define the test container

One base class; every integration test extends it:

```
@SpringBootTest
@AutoConfigureMockMvc
@Testcontainers
public abstract class AbstractIntegrationTest {

    @Container
    @ServiceConnection
    static final PostgreSQLContainer<?> POSTGRES = new PostgreSQLContainer<>("postgres:18-alpine");
}
```

- `@SpringBootTest` : start the whole application for the test
- `@AutoConfigureMockMvc` : make `MockMvc` available for sending requests
- `@Testcontainers` + `@Container` : start the container before the tests, stop it after
- `static` : one container for all tests of the class, not one per test
- `@ServiceConnection` : give the container's address, user and password to Spring as the datasource; no properties to write

Test database and test data

- the tests need the same schema as production, plus a few known rows; production must never get those rows
- so the tests run **two changelogs in order**: first the real one, then a test one on top of it
  - `src/test/resources/application.properties` is used by the tests instead of the one in `src/main/resources`, and names the test changelog:

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

- the test changelog's first line includes the real changelog, so every changeset of the application runs on the fresh container exactly as on the server; `ddl-auto=validate` then checks the entities against it as in production
- after the include, its own changeset inserts the test data:

```
<include file="db/changelog/changelog-master.xml"/>

<changeSet id="test-data" author="siim">
    <insert tableName="department"><column name="name" value="Sales"/></insert>
    <insert tableName="employee">
        <column name="name" value="John Smith"/>
        <column name="email" value="john.smith@example.com"/>
        <column name="department_id" valueNumeric="1"/>
    </insert>
</changeSet>
```

- the file lives under `src/test/resources`, so it is not in the jar and never reaches the server

Writing a test

```
@Transactional
class EmployeeControllerIntegrationTest extends AbstractIntegrationTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    void get_list_answersTheSeededEmployee() throws Exception {
        // the test-data changeset inserted John Smith in Sales
        mockMvc.perform(get("/api/employees"))
                // we expect that the response is 200
                .andExpect(status().isOk())
                // and that the one employee in it is John Smith, in Sales
                .andExpect(jsonPath("$", hasSize(1)))
                .andExpect(jsonPath("$[0].name").value("John Smith"))
                .andExpect(jsonPath("$[0].departmentName").value("Sales"));
    }
}
```

- `MockMvc` is injected with `@Autowired`; `perform` sends the request inside the application, no port, no network
- `andExpect` checks the answer; `jsonPath("$[0].name")` reads into the JSON body, `$` is the root
- `@Transactional` on a test class: every test runs in a transaction that is rolled back at the end, so the next test sees the data as the changelog left it

A request that creates, and one that is refused

```
@Test
void post_createsTheEmployee_andAnswers201() throws Exception {
    mockMvc.perform(post("/api/employees")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content("{\"name\":\"Mari Maasikas\",\"email\":\"mari.it@example.com\",\"departmentId\":1}"))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.id").isNumber())
            .andExpect(jsonPath("$.name").value("Mari Maasikas"));
}

@Test
void post_withAnEmptyNameAndABadEmail_answers400WithBothFields() throws Exception {
    mockMvc.perform(post("/api/employees")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content("{\"name\":\"\",\"email\":\"not-an-email\",\"departmentId\":1}"))
            .andExpect(status().isBadRequest())
            .andExpect(jsonPath("$.errors.name").value("must not be blank"))
            .andExpect(jsonPath("$.errors.email").value("must be a well-formed email address"));
}
```

- the second test checks the validation and the global handler together: the status, and the field names in the body the frontend reads

Running the tests

```
$ ./mvnw test
org.testcontainers.DockerClientFactory -- Testcontainers version: 2.0.5
tc.postgres:18-alpine -- Container postgres:18-alpine started in PT1.263671S
[INFO] Running ee.taltech.staff.employee.EmployeeControllerIntegrationTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 7.520 s -- in ee.taltech.staff.employee.EmployeeControllerIntegrationTest
[INFO] Running ee.taltech.staff.employee.EmployeeServiceTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.237 s -- in ee.taltech.staff.employee.EmployeeServiceTest
[INFO] Running ee.taltech.staff.StaffApplicationTests
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.429 s -- in ee.taltech.staff.StaffApplicationTests
[INFO] Results:
[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0
[INFO] BUILD SUCCESS
[INFO] Total time:  11.243 s
```

- the container starts once per test class and is gone when the run ends; the integration tests took 8 seconds, the application start included, the unit tests 0.2
- `./gradlew test` runs the same tests; its report is `build/reports/tests/test/index.html`
- the pipeline's build job runs them before it packages the jar; a red test stops the pipeline

A failing test

The same run with one expectation changed to `hasMessage("employee not found")`:

```
[INFO] Running ee.taltech.staff.employee.EmployeeServiceTest
[ERROR] Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.242 s <<< FAILURE! -- in ee.taltech.staff.employee.EmployeeServiceTest
[ERROR] ee.taltech.staff.employee.EmployeeServiceTest.findById_throwsNotFound_whenThereIsNoSuchEmployee -- Time elapsed: 0.009 s <<< FAILURE!
org.opentest4j.AssertionFailedError:

Expecting message to be:
  "employee not found"
but was:
  "employee 99 not found"

[INFO] Results:
[ERROR] Failures:
[ERROR]   EmployeeServiceTest.findById_throwsNotFound_whenThereIsNoSuchEmployee:61
[ERROR] Tests run: 9, Failures: 1, Errors: 0, Skipped: 0
[INFO] BUILD FAILURE
```

The method name says which rule broke, the assertion says expected and actual, the line number says where. In the pipeline this is a red `build-backend` job, the test report on the merge request page from lecture 3's `junit:` artifact, and no jar, no image, no deploy.

What to test first

For a project of three people with a deadline, in this order:

1. **Every rule in a service**, as unit tests: not found, the conditions under which a write is refused, a calculation. They are fast to write and fast to run, and the rules are where logic lives.
2. **One integration test per controller** for the happy path: the request as the frontend sends it, the status, a field of the body. It catches a wrong URL, a renamed field, a broken mapper.
3. **One integration test per error the frontend relies on**: the `400` with its field names, the `404` body.

Not getters, not MapStruct's generated code, not Spring. A test that cannot fail when the code is wrong is weight, not proof.

A test is written with the code it tests, in the same merge request. A test written a month later tests what the code happens to do.

Errors, validation and logging in one table

Thing	Where	Does
`NotFoundException extends RuntimeException`	`common/`, thrown by services	names what is missing, in the application's words
`ApiError` record	`common/`	the one body of every error: status, message, path, timestamp, field errors
`@RestControllerAdvice` + `@ExceptionHandler` + `@ResponseStatus`	`GlobalExceptionHandler`	exception type → status code and body, for every controller
`@NotBlank`, `@Email`, `@Size`, `@NotNull`	the create DTOs	the rules of a request
`@Valid`	controller parameter	runs the rules; `MethodArgumentNotValidException` → `400` with every field
`@Slf4j`, `log.info("… {}", id)`, `log.error("…", e)`	services, the handler	lines with time, level, class; the stack trace under an error

OpenAPI and tests in one table

Thing	Where	Does
`logging.level.<package>=DEBUG`	`application.properties`, or an environment variable on the server	which lines are written
`springdoc-openapi-starter-webmvc-ui`	`pom.xml` or `build.gradle`	`/v3/api-docs`, `/swagger-ui.html` from the code
`@ExtendWith(MockitoExtension.class)`, `@Mock`, `@InjectMocks`, `when`, `verify`	`src/test/java`, `…Test`	a unit test of one class with stand-ins
`@SpringBootTest`, `@Testcontainers`, `@ServiceConnection`, `MockMvc`	`src/test/java`	a request through the whole application, on a PostgreSQL container
`./mvnw test` or `./gradlew test`, the build	laptop, `build-backend`	every test; a red one stops the pipeline



Questions