Enterprise Programmering 2

Lesson 06: Errors, Mocking and Circuit Breakers

Dr. Andrea Arcuri

Goals

- Understand and tune how Spring deals with uncaught exceptions and bean validation
- Learn how to test services relying on other external services, by mocking those external services
- Understand how and why Circuit Breakers are used when dealing with external services

Errors/Validation In Spring

Validation

- In JEE, we have seen how *@annotation* constraints could be put on the inputs of EJB beans
  - same way as on JPA @Entity
- In Spring, we can do the same, which can be useful when handling query parameters
- But Spring does not do validation by default, needs to be activated with *@Validated*
  - *org.springframework.validation.annotation.Validated*

Exceptions

- When an exception is thrown and not caught, Spring creates a 500 HTTP response
- When using Wrapped Responses, the format used by Spring might be different from ours, so we might want to override it
- Might be cases in which we want to manually throw exceptions, which should results in 400 responses with our Wrapped format

Overriding Spring Defaults

- To change how Spring deals with exceptions, we can create a bean tagged with *@ControllerAdvice,* extending *ResponseEntityExceptionHandler*

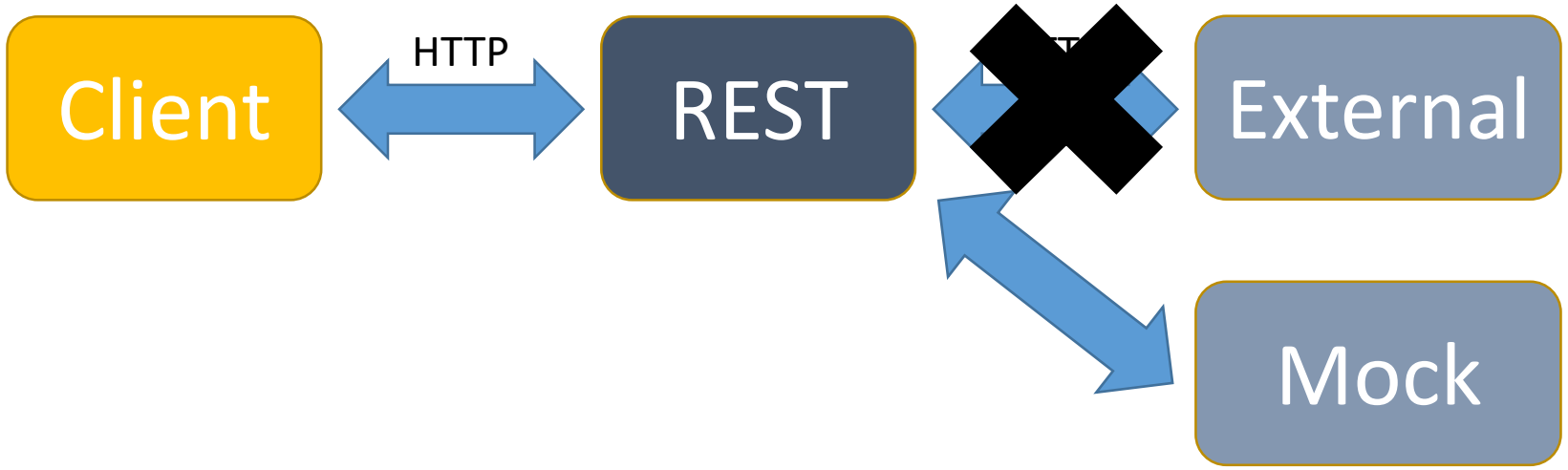
- Can have different exception handlers by using annotation *@ExceptionHandler* on different methods

Service Mocking

External Dependencies

- A client calls our REST, and we need to call an external service to compute our response
- What if the external service is currently down?
- What if the external service has a temporary bug?
- Test cases have to be *deterministic*
- External service poses a major challenge for testing

Mocking



- When running tests, run a Mock process listening to TCP connections
- Change in REST the IP address of External to point to Mock
- From the tests, specify what Mock should return when receiving HTTP messages
- REST does not know that it is speaking with Mock instead of the real External

Benefits

- Tests become *deterministic*
  - do not need to worry of network and External's state (eg its database)
- Can easily test scenarios which would be hard to configure for External
  - eg, special rare responses
- Can implement and test our REST even if External is not working/implemented yet
  - Example: 2 students writing 2 REST APIs. First student can implement and test X which depends on Y, even if other student is not done with Y yet

Downsides

- We are not testing how REST would behave in a real context, but in what is our *expectation* of how External interacts with it
- If lots of interactions, might need to write a lot of mocked responses
- If External does change often, then *maintaining* the mocks becomes expensive
- Still need some *live* tests with real External anyways
  - but those will be handled specially

Circuit Breaker

Circuit Breaker

- If too many connections to a server fail, stop ALL future attempts at connecting
- Can use a library (eg Netflix Hystrix) to wrap each call to external services
- Once the circuit breaker is on after several failures, it will periodically check if the server comes up again. If so, all communications are restored

Why?

- TCP/HTTP communications are expensive
- If an external service is down, want to avoid wasting resources in trying to connect to it
- Performance gain: can return response immediately instead of trying to connect to external services which are down
- When external service will come up again, don’t want to bombard it immediately with all clients resending all the messages that failed before, all at the same time (which might congest the external service again)

Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **advanced/rest/exception-handling**
- **advanced/rest/wiremock**
- **advanced/rest/circuit-breaker**