Exception Handling and Logging

Content

1. Exercises
2. Disabling white label error page
3. Displaying custom error pages
4. Error handling for REST
5. Logging

Exercises

1. Write a unit test for HomeService class
2. Cover user search endpoint with tests
3. Cover fetch user transactions endpoint with tests
4. Add deleted column to users table
    a. Deleted users must be excluded from all user fetch endpoints

Disable the Whitelabel Error Page Using Config

application.properties

```
server.error.whitelabel.enabled=false
```

Disable the Whitelabel Error Page Using Annotation

MoneyTransferAppApplication.java

```
@SpringBootApplication
@EnableAutoConfiguration(exclude = {ErrorMvcAutoConfiguration.class})
public class MoneyTransferAppApplication {

  public static void main(String[] args) {
    SpringApplication.run(MoneyTransferAppApplication.class, args);
  }
}
```

Displaying Custom Error Pages

resources/templates/error.html

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Something went wrong! </h1>
    <h2>Our Engineers are on it</h2>
    <a href="/">Go Home</a>
  </body>
</html>
```

Error Handling for REST

1. Controller level **@ExceptionHandler**
2. **@ControllerAdvice**
3. **ResponseStatusException** (Spring 5 and Above)

Controller level @ExceptionHandler

```
@Controller
@RequestMapping(path = "/api/exception")
public class ExceptionController {

   @ExceptionHandler({NoSuchElementException.class, JsonMappingException.class})
   public ModelAndView handleException(Exception exception) {
      ModelAndView modelAndView = new ModelAndView();
      modelAndView.setViewName("custom-error");
      modelAndView.addObject("message", exception.getMessage());
      return modelAndView;
   }
}
```

templates/custom-error.html

@ExceptionHandler & @ControllerAdvice

```
@ControllerAdvice
class RestResponseEntityExceptionHandler
        extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = {RecordConflictException.class})
    public final ResponseEntity<ErrorResponse> handleUserNotFoundException(RecordConflictException ex) {
        List<String> details = new ArrayList<>();
        details.add(ex.getLocalizedMessage());
        ErrorResponse error = new ErrorResponse(MoneyTransferAppApplication.RECORD_CONFLICT, details);
        return new ResponseEntity<>(error, HttpStatus.CONFLICT);
    }
}

//@EnableAutoConfiguration(exclude = {ErrorMvcAutoConfiguration.class})
```

@ControllerAdvice Advantages

- Full control over the body of the response as well as the status code
- Mapping of several exceptions to the same method, to be handled together, and
- It makes good use of the newer RESTful *ResposeEntity* response

ResponseStatusException (Spring 5 and Above)

```
@Controller
@RequestMapping(path = "/api/exception")
public class ExceptionController {


   @GetMapping("/{id}")
   public ResponseEntity<String> fetchOne(@PathVariable Long id) {

      throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Exception Not Found");
   }
}
```

Error file location => **templates/error-404.html**

Disable: *//@EnableAutoConfiguration(exclude = {ErrorMvcAutoConfiguration.class})*

ResponseStatusException Benefits

- Excellent for prototyping: We can implement a basic solution quite fast
- One type, multiple status codes: One exception type can lead to multiple different responses. **This reduces tight coupling compared to the *@ExceptionHandler***
- We won't have to create as many custom exception classes
- **More control over exception handling** since the exceptions can be created programmatically

ResponseStatusException tradeoffs

- There's no unified way of exception handling: It's more difficult to enforce some application-wide conventions, as opposed to *@ControllerAdvice* which provides a global approach
- Code duplication: We may find ourselves replicating code in multiple controllers

Logging

```
logger.trace("A TRACE Message");
logger.debug("A DEBUG Message");
logger.info("An INFO Message");
logger.warn("A WARN Message");
logger.error("An ERROR Message");
```

```
logging.level.root=TRACE
logging.level.org.springframework=TRACE
logging.level.com.nursultanturdaliev=TRACE
```

Initial Setup

```
@Controller
public class HomeController {

   private Logger logger = LoggerFactory.getLogger(HomeController.class);

   @Autowired
   private HomeService homeService;

   @RequestMapping(value = "/", method = RequestMethod.GET)
   public @ResponseBody
   String home() {
      logger.info("[Home] Request received");
      return homeService.welcome();
   }
} //logs/spring-boot-logger.log
```

Logback Configuration Logging

logback-spring.xml

Logging with Lombok

1. **Install dependency**

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.4</version>
  <scope>provided</scope>
</dependency>
```

2. **Install Lombok Plugin**

Logging with Lombok

```
@RestController
@Slf4j
public class LombokLoggingController {

   @RequestMapping("/lombok")
   public String index(HttpServletRequest request) {
      log.trace("A TRACE Message");
      log.debug("A DEBUG Message");
      log.info("An INFO Message");
      log.warn("A WARN Message");
      log.error("An ERROR Message");

      return "Howdy! Check out the Logs to see the output...";
   }
}
```

References

- https://www.baeldung.com/exception-handling-for-rest-with-spring
- https://www.baeldung.com/spring-boot-logging
- https://howtodoinjava.com/spring-core/spring-exceptionhandler-annotation/
- https://www.baeldung.com/spring-boot-custom-error-page
- https://howtodoinjava.com/spring-core/spring-exceptionhandler-annotation/
- https://www.baeldung.com/spring-response-status-exception