Error Handling with Spring

1. Overview

This article will illustrate **how to implement Exception Handling with Spring for a REST API**. We'll also get a bit of historical overview and see which new options the different versions introduced.

Before Spring 3.2, the two main approaches to handling exceptions in a Spring MVC application were: *HandlerExceptionResolver* or the *@ExceptionHandler* annotation. Both of these have some clear downsides.

Since 3.2 we've had the *@ControllerAdvice* annotation to address the limitations of the previous two solutions and to promote a unified exception handling throughout a whole application.

Now, **Spring 5 introduces the *ResponseStatusException* class**: A fast way for basic error handling in our REST APIs.

All of these do have one thing in common – they deal with the **separation of concerns** very well. The app can throw exception normally to indicate a failure of some kind – exceptions which will then be handled separately.

Finally, we'll see what Spring Boot brings to the table, and how we can configure it to suit our needs.

2. Solution 1 – The Controller level *@ExceptionHandler*

The first solution works at the *@Controller* level – we will define a method to handle exceptions, and annotate that with *@ExceptionHandler*:

```
public class FooController{

    //...
    @ExceptionHandler({ CustomException1.class, CustomException2.class })
    public void handleException() {
        //
    }
```

```
7}

8
```

This approach has a major drawback – **the *@ExceptionHandler* annotated method is only active for that particular Controller**, not globally for the entire application. Of course, adding this to every controller makes it not well suited for a general exception handling mechanism.

We can work around this limitation by having **all Controllers extend a Base Controller class** – however, this can be a problem for applications where, for whatever reason, this isn't possible. For example, the Controllers may already extend from another base class which may be in another jar or not directly modifiable, or may themselves not be directly modifiable.

Next, we'll look at another way to solve the exception handling problem – one that is global and doesn't include any changes to existing artifacts such as Controllers.

3. Solution 2 – *The HandlerExceptionResolver*

The second solution is to define an *HandlerExceptionResolver* – this will resolve any exception thrown by the application. It will also allow us to implement a **uniform exception handling mechanism** in our REST API.

Before going for a custom resolver, let's go over the existing implementations.

3.1. *ExceptionHandlerExceptionResolver*

This resolver was introduced in Spring 3.1 and is enabled by default in the *DispatcherServlet*. This is actually the core component of how the *@ExceptionHandler* mechanism presented earlier works.

3.2. *DefaultHandlerExceptionResolver*

This resolver was introduced in Spring 3.0, and it's enabled by default in the *DispatcherServlet*. It's used to resolve standard Spring exceptions to their corresponding HTTP Status Codes, namely Client error – *4xx* and Server error – *5xx* status codes. Here's the **full list** of the Spring Exceptions it handles, and how they map to status codes.

While it does set the Status Code of the Response properly, one **limitation is that it doesn't set anything to the body of the Response**. And for a REST API – the Status Code is really not enough information to present to the Client – the response has to have a body as well, to allow the application to give additional information about the failure.

This can be solved by configuring view resolution and rendering error content through *ModelAndView*, but the solution is clearly not optimal. That's why Spring 3.2 introduced a better option that we'll discuss in a later section.

3.3. *ResponseStatusExceptionResolver*

This resolver was also introduced in Spring 3.0 and is enabled by default in the *DispatcherServlet*. Its main responsibility is to use the *@ResponseStatus* annotation available on custom exceptions and to map these exceptions to HTTP status codes.

Such a custom exception may look like:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class MyResourceNotFoundException extends RuntimeException {
    public MyResourceNotFoundException() {
        super();
    }
    public MyResourceNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public MyResourceNotFoundException(String message) {
        super(message);
    }
    public MyResourceNotFoundException(Throwable cause) {
        super(cause);
    }
}
```

Same as the *DefaultHandlerExceptionResolver*, this resolver is limited in the way it deals with the body of the response – it does map the Status Code on the response, but the body is still *null.*

3.4. *SimpleMappingExceptionResolver* and *AnnotationMethodHandlerExceptionResolver*

The *SimpleMappingExceptionResolver* has been around for quite some time – it comes out of the older Spring MVC model and is **not very relevant for a REST Service**. We basically use it to map exception class names to view names.

The *AnnotationMethodHandlerExceptionResolver* was introduced in Spring 3.0 to handle exceptions through the *@ExceptionHandler* annotation but has been deprecated by *ExceptionHandlerExceptionResolver* as of Spring 3.2.

3.5. Custom *HandlerExceptionResolver*

The combination of *DefaultHandlerExceptionResolver* and *ResponseStatusExceptionResolver* goes a long way towards providing a good error handling mechanism for a Spring RESTful Service. The downside is – as mentioned before – **no control over the body of the response**.

Ideally, we'd like to be able to output either JSON or XML, depending on what format the client has asked for (via the *Accept* header).

This alone justifies creating **a new, custom exception resolver**:

```
@Component
public class RestResponseStatusExceptionResolver extends
AbstractHandlerExceptionResolver {


    @Override
    protected ModelAndView doResolveException(
      HttpServletRequest request,
      HttpServletResponse response,
      Object handler,
      Exception ex) {
        try {
            if (ex instanceof IllegalArgumentException) {
                return handleIllegalArgument((IllegalArgumentException) ex,
response, handler);
            }
            ...
        } catch (Exception handlerException) {
            logger.warn("Handling of [" + ex.getClass().getName() + "]
              resulted in Exception", handlerException);
        }
        return null;
    }

    private ModelAndView
```

```
22      handleIllegalArgument(IllegalArgumentException ex, HttpServletResponse
  response)
23      throws IOException {
24        response.sendError(HttpServletResponse.SC_CONFLICT);
25        String accept = request.getHeader(HttpHeaders.ACCEPT);
26        ...
27        return new ModelAndView();
28    }
29}
30
```

One detail to notice here is that we have access to the *request* itself, so we can consider the value of the *Accept* header sent by the client.

For example, if the client asks for *application/json* then, in the case of an error condition, we'd want to make sure we return a response body encoded with *application/json*.

The other important implementation detail is that **we return a *ModelAndView* – this is the body of the response** and it will allow us to set whatever is necessary on it.

This approach is a consistent and easily configurable mechanism for the error handling of a Spring REST Service. It does, however, have limitations: it's interacting with the low-level *HtttpServletResponse* and it fits into the old MVC model which uses *ModelAndView* – so there's still room for improvement.

4. Solution 3 – *@ControllerAdvice*

Spring 3.2 brings support for **a global *@ExceptionHandler* with the *@ControllerAdvice* annotation**. This enables a mechanism that breaks away from the older MVC model and makes use of *ResponseEntity* along with the type safety and flexibility of *@ExceptionHandler*:

```
1 @ControllerAdvice
2 public class RestResponseEntityExceptionHandler
    extends ResponseEntityExceptionHandler {
3
4
      @ExceptionHandler(value
5       = { IllegalArgumentException.class, IllegalStateException.class })
6     protected ResponseEntity<Object> handleConflict(
```

```
7       RuntimeException ex, WebRequest request) {
8         String bodyOfResponse = "This should be application specific";
9         return handleExceptionInternal(ex, bodyOfResponse,
10          new HttpHeaders(), HttpStatus.CONFLICT, request);
11    }
12 }
13
```

The*@ControllerAdvice* annotation allows us to **consolidate our multiple, scattered *@ExceptionHandler*s from before into a single, global error handling component**.

The actual mechanism is extremely simple but also very flexible. It gives us:

- 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

One thing to keep in mind here is to **match the exceptions declared with *@ExceptionHandler* with the exception used as the argument of the method**. If these don't match, the compiler will not complain – no reason it should, and Spring will not complain either.

However, when the exception is actually thrown at runtime, **the exception resolving mechanism will fail with**:

```
1 java.lang.IllegalStateException: No suitable resolver for argument [0]
  [type=...]
2 HandlerMethod details: ...
```

5. Solution 4 – *ResponseStatusException* (Spring 5 and Above)

Spring 5 introduced the *ResponseStatusException* class. We can create an instance of it providing an *HttpStatus* and optionally a *reason* and a *cause*:

```
  @GetMapping(value = "/{id}")
1
  public Foo findById(@PathVariable("id") Long id, HttpServletResponse
2 response) {
3     try {
4         Foo resourceById =
  RestPreconditions.checkFound(service.findOne(id));
5
```

```
6           eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this,
  response));
7
            return resourceById;
8
        }
9       catch (MyResourceNotFoundException exc) {
10           throw new ResponseStatusException(
11             HttpStatus.NOT_FOUND, "Foo Not Found", exc);
12      }
13  }
```

What are the benefits of using *ResponseStatusException*?

- 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

And what about the 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

We should also note that it's possible to combine different approaches within one application.

For example, we can implement a *@ControllerAdvice* globally, but also *ResponseStatusException*s locally. However, we need to be careful: If the same exception can be handled in multiple ways, we may notice some surprising behavior. A possible convention is to handle one specific kind of exception always in one way.

For more details and further examples, see our tutorial on *ResponseStatusException*.

6. Handle the Access Denied in Spring Security

The Access Denied occurs when an authenticated user tries to access resources that he doesn't have enough authorities to access.

6.1. MVC – Custom Error Page

First, let's look at the MVC style of the solution and see how to customize an error page for Access Denied:

The XML configuration:

```
<http>
    <intercept-url pattern="/admin/*" access="hasAnyRole('ROLE_ADMIN')"/>
    ...
    <access-denied-handler error-page="/my-error-page" />
</http>
```

And the Java configuration:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/*").hasAnyRole("ROLE_ADMIN")
        ...
        .and()
        .exceptionHandling().accessDeniedPage("/my-error-page");
}
```

When users try to access a resource without having enough authorities, they will be redirected to "*/my-error-page*".

6.2. Custom *AccessDeniedHandler*

Next, let's see how to write our custom *AccessDeniedHandler*:

```
@Component
public class CustomAccessDeniedHandler implements AccessDeniedHandler {

    @Override
    public void handle
      (HttpServletRequest request, HttpServletResponse response,
AccessDeniedException ex)
      throws IOException, ServletException {
```

```
        response.sendRedirect("/my-error-page");
    }
}
```

And now let's configure it using **XML Configuration**:

```
<http>
    <intercept-url pattern="/admin/*" access="hasAnyRole('ROLE_ADMIN')"/>
    ...
    <access-denied-handler ref="customAccessDeniedHandler" />
</http>
```

Or using Java Configuration:

```
@Autowired
private CustomAccessDeniedHandler accessDeniedHandler;

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/admin/*").hasAnyRole("ROLE_ADMIN")
        ...
        .and()
        .exceptionHandling().accessDeniedHandler(accessDeniedHandler)
}
```

Note how – in our *CustomAccessDeniedHandler*, we can customize the response as we wish by redirecting or display a custom error message.

6.3. REST and Method Level Security

Finally, let's see how to handle method level security *@PreAuthorize*, *@PostAuthorize*, and *@Secure* Access Denied.

We'll, of course, use the global exception handling mechanism that we discussed earlier to handle the *AccessDeniedException* as well:

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

    @ExceptionHandler({ AccessDeniedException.class })
    public ResponseEntity<Object> handleAccessDeniedException(
      Exception ex, WebRequest request) {
        return new ResponseEntity<Object>(
          "Access denied message here", new HttpHeaders(),
HttpStatus.FORBIDDEN);
    }

    ...
}
```

7. Spring Boot Support

Spring Boot provides an *ErrorController* implementation to handle errors in a sensible way.

In a nutshell, it serves a fallback error page for browsers (aka the Whitelabel Error Page), and a JSON response for RESTful, non HTML requests:

```
{
    "timestamp": "2019-01-17T16:12:45.977+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "Error processing the request!",
    "path": "/my-endpoint-with-exceptions"
}
```

As usual, Spring Boot allows configuring these features with properties:

- *server.error.whitelabel.enabled:* can be used to disable the Whitelabel Error Page and rely on the servlet container to provide an HTML error message
- *server.error.include-stacktrace:* with an *always* value, it includes the stacktrace in both the HTML and the JSON default response

Apart from these properties, **we can provide our own view-resolver mapping for /*error*, overriding the Whitelabel Page.**

We can also customize the attributes that we want to show in the response by including an *ErrorAttributes* bean in the context. We can extend the *DefaultErrorAttributes* class provided by Spring Boot to make things easier:

```
@Component
public class MyCustomErrorAttributes extends DefaultErrorAttributes {

    @Override
    public Map<String, Object> getErrorAttributes(
      WebRequest webRequest, boolean includeStackTrace) {
        Map<String, Object> errorAttributes =
          super.getErrorAttributes(webRequest, includeStackTrace);
        errorAttributes.put("locale", webRequest.getLocale()
            .toString());
        errorAttributes.remove("error");

        //...

        return errorAttributes;
    }
}
```

If we want to go further and define (or override) how the application will handle errors for a particular content type, we can register an *ErrorController* bean.

Again, we can make use of the default *BasicErrorController* provided by Spring Boot to help us out.

For example, imagine we want to customize how our application handles errors triggered in XML endpoints. All we have to do is define a public method using the *@RequestMapping* and stating it produces *application/xml* media type:

```
@Component
public class MyErrorController extends BasicErrorController {

    public MyErrorController(ErrorAttributes errorAttributes) {
        super(errorAttributes, new ErrorProperties());
    }

    @RequestMapping(produces = MediaType.APPLICATION_XML_VALUE)
    public ResponseEntity<Map<String, Object>> xmlError(HttpServletRequest
request) {

    // ...

    }
}
```

8. Conclusion

This tutorial discussed several ways to implement an exception handling mechanism for a REST API in Spring, starting with the older mechanism and continuing with the Spring 3.2 support and into 4.x and 5.x.