Error Handling for REST with Spring

This tutorial 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 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

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() {
        //
    }
}
```

This approach has a major drawback: T**he @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 solution can be a problem for applications where, for whatever reason, that 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.

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.

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);
    }
}
```

The 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. Custom HandlerExceptionResolver

The combination of DefaultHandlerExceptionResolver and ResponseStatusExceptionResolver goes a long way toward 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**:

```
@Componentpublic 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
      handleIllegalArgument(IllegalArgumentException ex, HttpServletResponse
response)
      throws IOException {
        response.sendError(HttpServletResponse.SC_CONFLICT);
        String accept = request.getHeader(HttpHeaders.ACCEPT);
        ...
        return new ModelAndView();
    }
}
```

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 fits into the old MVC model that uses ModelAndView, so there's still room for improvement.