Error and Exception handling

Created	@Oct 6, 2020 5:34 PM
Tags	
Updated	@Oct 6, 2020 5:35 PM

Spring MVC provides several complimentary approaches to exception handling.

Today I'm going to show you the various options available. Our goal is to not handle exceptions explicitly in Controller methods where possible. They are a cross-cutting concern better handled separately in dedicated code.

There are three options:

- per exception,
- per controller
- globally.

Spring MVC offers no default (fall-back) error page out-of-the-box. The most common way to set a default error page has always been the `SimpleMappingExceptionResolver` (since Spring V1 in fact). We will discuss this later.

However Spring Boot *does* provide for a fallback error-handling page.

At start-up, Spring Boot tries to find a mapping for `/error`. By convention, a URL ending in `/error` maps to a logical view of the same name: `error`. In the demo application this view maps in turn to the `error.html` Thymeleaf template. (If using JSP, it would map to `error.jsp` according to the setup of your `InternalResourceViewResolver`). The actual mapping will depend on what `ViewResolver` (if any) that you or Spring Boot has setup.

If no view-resolver mapping for `/error` can be found, Spring Boot defines its own fall-back error page - the so-called "Whitelabel Error Page" (a minimal

page with just the HTTP status information and any error details, such as the message from an uncaught exception). In the sample applicaiton, if you rename the `error.html` template to, say, `error2.html` then restart, you will see it being used.

Spring Boot also sets up a default error-page for the container, equivalent to the

```
<error-page>
```

directive in

```
web.xml
```

(although implemented very differently). Exceptions thrown outside the Spring MVC framework, such as from a servlet Filter, are still reported by the Spring Boot fallback error page.

Normally any unhandled exception thrown when processing a web-request causes the server to return an HTTP 500 response. However, any exception that you write yourself can be annotated with the @ResponseStatus annotation (which supports all the HTTP status codes defined by the HTTP specification). When an annotated exception is thrown from a controller method, and not handled elsewhere, it will automatically cause the appropriate HTTP response to be returned with the specified status-code.

```
// declaration
@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="No such Order")  // 404
 public class OrderNotFoundException extends RuntimeException {
     // ...
 }

// how to use it
@RequestMapping(value="/orders/{id}", method=GET)
 public String showOrder(@PathVariable("id") long id, Model model) {
     Order order = orderRepository.findOrderById(id);

     if (order == null) throw new OrderNotFoundException(id);

     model.addAttribute(order);
     return "orderDetail";
 }
```

Controller Based Exception Handling

You can add extra (`@ExceptionHandler`) methods to any controller to specifically handle exceptions thrown by request handling (`@RequestMapping`) methods in the same controller. Such methods can:

1. Handle exceptions without the `@ResponseStatus` annotation (typically predefined exceptions that you didn't write)
2. Redirect the user to a dedicated error view
3. Build a totally custom error response

```
@Controller
public class ExceptionHandlingController {

  // @RequestHandler methods
  ...

  // Exception handling methods

  // Convert a predefined exception to an HTTP Status code
  @ResponseStatus(value=HttpStatus.CONFLICT,
                  reason="Data integrity violation")  // 409
  @ExceptionHandler(DataIntegrityViolationException.class)
  public void conflict() {
    // Nothing to do
  }

  // Specify name of a specific view that will be used to display the error:
  @ExceptionHandler({SQLException.class,DataAccessException.class})
  public String databaseError() {
    // Nothing to do.  Returns the logical view name of an error page, passed
    // to the view-resolver(s) in usual way.
    // Note that the exception is NOT available to this view (it is not added
    // to the model) but see "Extending ExceptionHandlerExceptionResolver"
    // below.
    return "databaseError";
  }

  // Total control - setup a model and return the view name yourself. Or
  // consider subclassing ExceptionHandlerExceptionResolver (see below).
  @ExceptionHandler(Exception.class)
  public ModelAndView handleError(HttpServletRequest req, Exception ex) {
    logger.error("Request: " + req.getRequestURL() + " raised " + ex);
```

```
    ModelAndView mav = new ModelAndView();
    mav.addObject("exception", ex);
    mav.addObject("url", req.getRequestURL());
    mav.setViewName("error");
    return mav;
  }
}
```

In any of these methods you might choose to do additional processing - the most common example is to log the exception.

Exceptions and Views

Be careful when adding exceptions to the model. Your users do not want to see web-pages containing Java exception details and stack-traces. You may have security policies that expressly *forbid* putting *any* exception information in the error page. Another reason to make sure you override the Spring Boot white-label error page.

Global Exception Handling

Using @ControllerAdvice Classes

A controller advice allows you to use exactly the same exception handling techniques but apply them across the whole application, not just to an individual controller. You can think of them as an annotation driven interceptor.

Any class annotated with `@ControllerAdvice` becomes a controller-advice and three types of method are supported:

- Exception handling methods annotated with `@ExceptionHandler`.
- Model enhancement methods (for adding additional data to the model) annotated with `@ModelAttribute`. Note that these attributes are *not* available to the exception handling views.
- Binder initialization methods (used for configuring form-handling) annotated with `@InitBinder`.

Any of the exception handlers you saw above can be defined on a controller-advice class - but now they apply to exceptions thrown from *any* controller. Here is a simple example:

```
@ControllerAdvice
class GlobalControllerExceptionHandler {
    @ResponseStatus(HttpStatus.CONFLICT)  // 409
    @ExceptionHandler(DataIntegrityViolationException.class)
    public void handleConflict() {
        // Nothing to do
    }
}
```

If you want to have a default handler for *any* exception, there is a slight wrinkle. You need to ensure annotated exceptions are handled by the framework. The code looks like this:

```
@ControllerAdvice
class GlobalDefaultExceptionHandler {
  public static final String DEFAULT_ERROR_VIEW = "error";

  @ExceptionHandler(value = Exception.class)
  public ModelAndView
  defaultErrorHandler(HttpServletRequest req, Exception e) throws Exception {
    // If the exception is annotated with @ResponseStatus rethrow it and let
    // the framework handle it - like the OrderNotFoundException example
    // at the start of this post.
    // AnnotationUtils is a Spring Framework utility class.
    if (AnnotationUtils.findAnnotation
                (e.getClass(), ResponseStatus.class) != null)
      throw e;

    // Otherwise setup and send the user to a default error-view.
    ModelAndView mav = new ModelAndView();
    mav.addObject("exception", e);
    mav.addObject("url", req.getRequestURL());
    mav.setViewName(DEFAULT_ERROR_VIEW);
    return mav;
  }
}
```

HandlerExceptionResolver

Any Spring bean declared in the `DispatcherServlet`'s application context that implements `HandlerExceptionResolver` will be used to intercept and process any exception raised in the MVC system and not handled by a Controller. The interface looks like this:

```
public interface HandlerExceptionResolver {
    ModelAndView resolveException(HttpServletRequest request,
```

```
            HttpServletResponse response, Object handler, Exception ex);
}COPY
```

The `handler` refers to the controller that generated the exception (remember that `@Controller` instances are only one type of handler supported by Spring MVC. For example: `HttpInvokerExporter` and the WebFlow Executor are also types of handler).

Behind the scenes, MVC creates three such resolvers by default. It is these resolvers that implement the behaviours discussed above:

- `ExceptionHandlerExceptionResolver` matches uncaught exceptions against suitable `@ExceptionHandler` methods on both the handler (controller) and on any controller-advices.
- `ResponseStatusExceptionResolver` looks for uncaught exceptions annotated by `@ResponseStatus` (as described in Section 1)
- `DefaultHandlerExceptionResolver` converts standard Spring exceptions and converts them to HTTP Status Codes (I have not mentioned this above as it is internal to Spring MVC).

These are chained and processed in the order listed - internally Spring creates a dedicated bean (the `HandlerExceptionResolverComposite`) to do this.

Notice that the method signature of `resolveException` does not include the `Model`. This is why `@ExceptionHandler` methods cannot be injected with the model.

Errors and REST

RESTful GET requests may also generate exceptions and we have already seen how we can return standard HTTP Error response codes. However, what if you want to return information about the error? This is very easy to do. Firstly define an error class:

```
public class ErrorInfo {
    public final String url;
    public final String ex;

    public ErrorInfo(String url, Exception ex) {
        this.url = url;
        this.ex = ex.getLocalizedMessage();
```

```
    }
}
```

Now we can return an instance from a handler as the `@ResponseBody` like this:

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MyBadDataException.class)
@ResponseBody ErrorInfo
handleBadRequest(HttpServletRequest req, Exception ex) {
    return new ErrorInfo(req.getRequestURL(), ex);
}
```

What to Use When?

As usual, Spring likes to offer you choice, so what should you do? Here are some rules of thumb. However if you have a preference for XML configuration or Annotations, that's fine too.

- For exceptions you write, consider adding `@ResponseStatus` to them.
- For all other exceptions implement an `@ExceptionHandler` method on a `@ControllerAdvice` class or use an instance of `SimpleMappingExceptionResolver`. You may well have `SimpleMappingExceptionResolver` configured for your application already, in which case it may be easier to add new exception classes to it than implement a `@ControllerAdvice`.
- For Controller specific exception handling add `@ExceptionHandler` methods to your controller.
- **Warning:** Be careful mixing too many of these options in the same application. If the same exception can be handed in more than one way, you may not get the behavior you wanted. `@ExceptionHandler` methods on the Controller are always selected before those on any `@ControllerAdvice` instance. It is what order controller-advices are processed.