ENGINEERING

Exception Handling in Spring MVC

ENGINEERING PAUL CHAPMAN NOVEMBER 01, 2013
17 COMMENTS

ALL POSTS
ENGINEERING
RELEASES
NEWS AND EVENTS

Spring MVC provides several complimentary approaches to exception handling but, when teaching Spring MVC, I often find that my students are confused or not comfortable with them.

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 or globally.

A demonstration application that shows the points discussed here can be found at
http://github.com/paulc4/mvc-exceptions.
See Sample Application below for details.

NOTE: The demo applications has been revamped and updated (October 2014) to use Spring Boot 1.1.8 and is (hopefully) easier to use and understand.

Using HTTP Status Codes

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.

For example, here is an exception for a missing order.

```
    @ResponseStatus(value=HttpStatus.NOT_FOUND, reason="No su
ch Order")  // 404
    public class OrderNotFoundException extends RuntimeExcept
ion {
        // ...
    }
```

And here is a controller method using it:

```
    @RequestMapping(value="/orders/{id}", method=GET)
    public String showOrder(@PathVariable("id") long id, Mode
l model) {
        Order order = orderRepository.findOrderById(id);
        if (order == null) throw new OrderNotFoundException(i
d);
        model.addAttribute(order);
        return "orderDetail";
    }
```

A familiar HTTP 404 response will be returned if the URL handled by this method includes an unknown order id.

Controller Based Exception Handling

Using @ExceptionHandler

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

The following controller demonstrates these three options:

```
@Controller
public class ExceptionHandlingController {

  // @RequestHandler methods
  ...

  // Exception handling methods

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

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

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

    ModelAndView mav = new ModelAndView();
    mav.addObject("exception", exception);
    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.

Handler methods have flexible signatures so you can pass in obvious servlet-related objects such as `HttpServletRequest`, `HttpServletResponse`, `HttpSession` and/or `Principle`. **Important Note:** the `Model` may **not** be a parameter of any `@ExceptionHandler` method. Instead, setup a model inside the method using a `ModelAndView` as shown by `handleError()` above.

Exceptions and Views

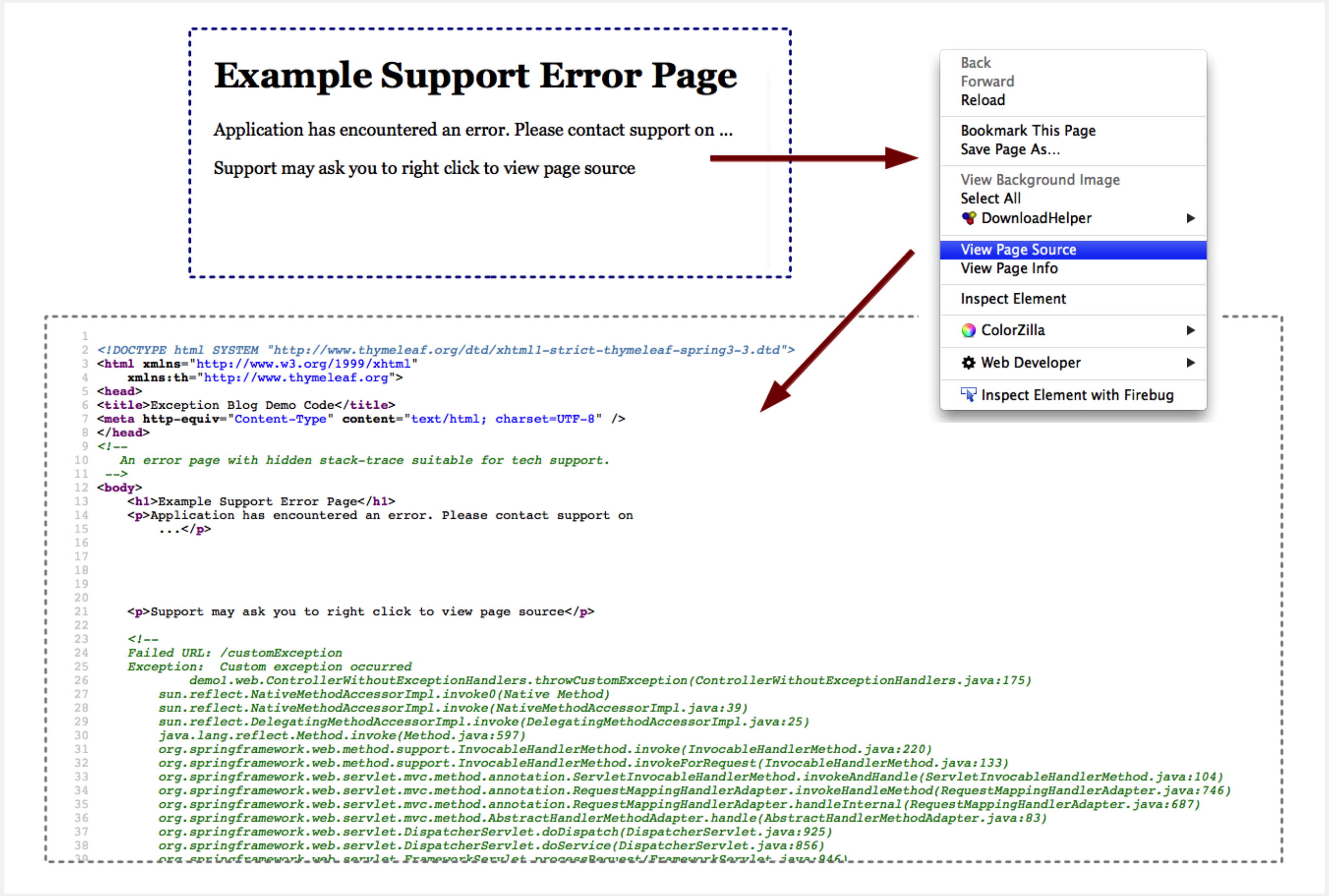
Be careful when adding exceptions to the model. Your users do not want to see
web-pages containing Java exception details and stack-traces. However, it can be useful to put exception
details in the page source as a comment, to assist your support people. If using JSP, you could
do something like this to output the exception and the corresponding stack-trace (using a hidden
`<div>` is another option).

```
    <h1>Error Page</h1>
    <p>Application has encountered an error. Please contact s
upport on ...</p>

    <!--
    Failed URL: ${url}
    Exception:  ${exception.message}
        <c:forEach items="${exception.stackTrace}" var="ste">
${ste}
    </c:forEach>
    -->
```

For the Thymeleaf equivalent see
support.html

in the demo application. The result looks like this.


Example Support Error Page

Application has encountered an error. Please contact support on ...

Support may ask you to right click to view page source

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-spring3-3.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:th="http://www.thymeleaf.org">
<head>
<title>Exception Blog Demo Code</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<!--
   An error page with hidden stack-trace suitable for tech support.
 -->
<body>
    <h1>Example Support Error Page</h1>
    <p>Application has encountered an error. Please contact support on
        ...</p>





    <p>Support may ask you to right click to view page source</p>

    <!--
    Failed URL: /customException
    Exception:  Custom exception occurred
            demo1.web.ControllerWithoutExceptionHandlers.throwCustomException(ControllerWithoutExceptionHandlers.java:175)
        sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        java.lang.reflect.Method.invoke(Method.java:597)
        org.springframework.web.method.support.InvocableHandlerMethod.invoke(InvocableHandlerMethod.java:220)
        org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:133)
        org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:104)
        org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandleMethod(RequestMappingHandlerAdapter.java:746)
        org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:687)
        org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:83)
        org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:925)
        org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:856)
        org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:946)
```

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`.

We are only going to look at exception handling - see the online manual for more on `@ControllerAdvice` methods.

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(HttpServletReques
t req, Exception e) throws Exception {
        // If the exception is annotated with @ResponseStatus
rethrow it and let
```

```
        // the framework handle it - like the OrderNotFoundEx
ception example
        // at the start of this post.
        // AnnotationUtils is a Spring Framework utility clas
s.
        if (AnnotationUtils.findAnnotation(e.getClass(), Resp
onseStatus.class) != null)
            throw e;

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

Going Deeper

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, Exc
eption ex);
}
```

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 for
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.

You can, if you wish, implement your own `HandlerExceptionResolver` to setup your own custom
exception handling system. Handlers typically implement Spring's `Ordered` interface so you can define the
order that the handlers run in.

SimpleMappingExceptionResolver

Spring has long provided a simple but convenient implementation of `HandlerExceptionResolver` that you may well find being used in your appication already - the `SimpleMappingExceptionResolver`. It provides options to:

- Map exception class names to view names - just specify the classname, no package needed.
- Specify a default (fallback) error page for any exception not handled anywhere else
- Log a message (this is not enabled by default).
- Set the name of the `exception` attribute to add to the Model so it can be used inside a View (such as a JSP). By default this attribute is named `exception`. Set to `null` to disable. Remember that views returned from `@ExceptionHandler` methods do not have access to the exception but views defined to `SimpleMappingExceptionResolver` do.

Here is a typical configuration using XML:

```
    <bean id="simpleMappingExceptionResolver"
          class="org.springframework.web.servlet.handler.Simp
leMappingExceptionResolver">
        <property name="exceptionMappings">
            <map>
                <entry key="DatabaseException" value="databas
eError"/>
                <entry key="InvalidCreditCardException" value
="creditCardError"/>
            </map>
        </property>
        <!-- See note below on how this interacts with Spring
Boot -->
        <property name="defaultErrorView" value="error"/>
        <property name="exceptionAttribute" value="ex"/>

        <!-- Name of logger to use to log exceptions. Unset b
y default, so logging disabled -->
        <property name="warnLogCategory" value="example.MvcLo
gger"/>
    </bean>
```

Or using Java Configuration:

```
@Configuration
@EnableWebMvc   // Optionally setup Spring MVC defaults if yo
```

```
u aren't doing so elsewhere
public class MvcConfiguration extends WebMvcConfigurerAdapter
{
    @Bean(name="simpleMappingExceptionResolver")
    public SimpleMappingExceptionResolver createSimpleMapping
ExceptionResolver() {
        SimpleMappingExceptionResolver r =
              new SimpleMappingExceptionResolver();

        Properties mappings = new Properties();
        mappings.setProperty("DatabaseException", "databaseEr
ror");
        mappings.setProperty("InvalidCreditCardException", "c
reditCardError");

        r.setExceptionMappings(mappings);  // None by default
        r.setDefaultErrorView("error");    // No default
        r.setExceptionAttribute("ex");     // Default is "exc
eption"
        r.setWarnLogCategory("example.MvcLogger");     // No
default
        return r;
    }
    ...
}
```

The defaultErrorView property is especially useful as it ensures any

uncaught exception generates
a suitable application defined error page. (The default for most
application servers is to display a Java
stack-trace - something your users should never see).

Extending SimpleMappingExceptionResolver

It is quite common to extend `SimpleMappingExceptionResolver` for
several reasons:

- Use the constructor to set properties directly - for example to enable exception logging and set the
  logger to use
- Override the default log message by overriding `buildLogMessage`. The default implementation
  always returns this fixed text:

  > Handler execution resulted in exception

- To make additional information available to the error view by overriding `doResolveException`

For example:

```
public class MyMappingExceptionResolver extends SimpleMapping
ExceptionResolver {
    public MyMappingExceptionResolver() {
        // Enable logging by providing the name of the logger
to use
        setWarnLogCategory(MyMappingExceptionResolver.class.g
etName());
    }

    @Override
    public String buildLogMessage(Exception e, HttpServletReq
uest req) {
        return "MVC exception: " + e.getLocalizedMessage();
    }

    @Override
    protected ModelAndView doResolveException(HttpServletRequ
est request,
            HttpServletResponse response, Object handler, Exc
eption exception) {
        // Call super method to get the ModelAndView
        ModelAndView mav = super.doResolveException(request,
response, handler, exception);

        // Make the full URL available to the view - note Mod
elAndView uses addObject()
        // but Model uses addAttribute(). They work the same.
        mav.addObject("url", request.getRequestURL());
        return mav;
    }
}
```

This code is in the demo application as
ExampleSimpleMappingExceptionResolver

Extending ExceptionHandlerExceptionResolver

It is also possible to extend `ExceptionHandlerExceptionResolver` and
override its
`doResolveHandlerMethodException` method in the same way. It has
almost the same signature
(it just takes the new `HandlerMethod` instead of a `Handler`).

To make sure it gets used, also set the inherited order property (for
example in the constructor of
your new class) to a value less than `MAX_INT` so it runs before the
default
ExceptionHandlerExceptionResolver instance (it is easier to create your
own handler instance than try to
modify/replace the one created by Spring). See
ExampleExceptionHandlerExceptionResolver

in the demo app for more.

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 r
eq, 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 undefined what order controller-advices are processed.

Sample Application

A demonstration application can be found at github.
It uses Spring Boot and Thymeleaf to build a simple web application.

The application was revised (Oct 2014) and is (hopefully) better and easier to understand. The fundamentals stay the same. It uses Spring Boot V1.1.8 and Spring 4.1 but the code is applicable to Spring 3.x also.

The demo is running on Cloud Foundry at http://mvc-exceptions-v2.cfapps.io/.

About the Demo

The application leads the user through 5 demo pages, highlighting different exception handling techniques:

1. A controller with `@ExceptionHandler` methods to handle its own

exceptions

2. A contoller that throws exceptions for a global ControllerAdvice to handle
3. Using a `SimpleMappingExceptionResolver` to handle exceptions
4. Same as demo 3 but with the `SimpleMappingExceptionResolver` disabled for comparison
5. Shows how Spring Boot generates its error page

A description of the most important files in the application and how they relate to each demo can be found in the project's README.md.

The home web-page is index.html which:

- Links to each demo page
- Links (bottom of the page) to Spring Boot endpoints for those interested in Spring Boot.

Each demo page contains several links, all of which deliberately raise exceptions. You will need to use the back-button on your browser each time to return to the demo page.

Thanks to Spring Boot, you can run this demo as a Java application (it runs an embedded Tomcat container). To run the application, you can use one of the following (the second is thanks to the Spring Boot maven plugin):

- `mvn exec:java`
- `mvn spring-boot:run`

Your choice. The home page URL will be http://localhost:8080.

Spring Boot and Error Handling

Spring Boot allows a Spring project to be setup with
minimal configuration. Spring Boot creates sensible defaults
automatically when it detects
certain key classes and packages on the classpath. For example if it sees
that you are using a Servlet
environment, it sets up Spring MVC with the most commonly used view-resolvers, hander mappings and so forth.
If it sees JSP and/or Thymeleaf, it sets up these view-technologies.

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). However
Spring Boot also provides 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`).

If no mapping from `/error` to a View 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). If you rename the
`error.html` template to, say, `error2.html`
then restart, you will see it being used.

By defining a Java configuration `@Bean` method called
`defaultErrorView()` you can return your own error `View` instance. (see

Spring Boot's `ErrorMvcAutoConfiguration` class for more information).

What if you are already using `SimpleMappingExceptionResolver` to setup a default
error view? Simple, make sure the `defaultErrorView` defines the same view that Spring Boot uses: `error`. Or you can disable Spring boot's error page by setting the property
`error.whitelabel.enabled` to `false`. Your container's default error page is used instead.
There are examples of setting Spring Boot properties in the constructor of
Main.

Note that in the demo, the `defaultErrorView` property of the `SimpleMappingExceptionResolver` is
deliberately set not to `error` but to `defaultErrorPage` so you can see when the handler is generating the error page and when
Spring Boot is responsible. Normally both would be set to `error`.

Also in the demo application I show how to create a support-ready error page with a stack-trace hidden in the HTML source (as a comment). Ideally support should get this information from the logs, but life isn't always ideal. Regardless, what this page does show is how the underlying

error-handling method `handleError` creates its own `ModelAndView` to provide extra information in the error page. See:
* `ExceptionHandlingController.handleError()` on github
* `GlobalControllerExceptionHandler.handleError()` on github

17 Comments spring.io

Recommend 4 Share

Join the discussion…

Jesper de Jong • 7 months ago

I have a remark, or question, about the @ResponseStatus annotation.

With this annotation you can specify the HTTP status code and mess
specific type of exception occurs. I find it a bit inflexible that you can or
have something like this:

@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class OrderNotFoundException {

super("There is no order with id: " + orderId);
}
}

Instead of specifying a fixed message string with 'reason = ...' in the ar
exception message, which is in this case "There is no order with id: ...
status code = 404, message string = "There is no order with id: ...".

Is this currently possible? If not, then is it a reasonable feature request

6 ^ | ⌄ • Reply • Share ›

Thomas Jarnot ➦ Jesper de Jong • 4 months ago

As you might have noticed from the javadoc of @ExceptionHar
allowed to return various types. Returning an instance of Resp
message) should meet you flexibility requirements.

^ | ⌄ • Reply • Share ›

Jan Šťastný • a year ago

Hi,

thanks a lot for a nice post.

I have used the technique of handling generic Exception in my @Conti
exceptions annotated with @ResponseStatus annotation as proposed
if (AnnotationUtils.findAnnotation(e.getClass(), ResponseStatus.class)
throw e;

However, using this technique causes an error message with text:

"Failed to invoke @ExceptionHandler method ..." to appear in the log.
ExceptionHandlerExceptionResolver:

try {

if (logger.isDebugEnabled()) {

logger.debug("Invoking @ExceptionHandler method: " + exceptionHan

see more

1 ^ | ⌄ • Reply • Share ›

Brian Clozel Mod ➦ Jan Šťastný • a year ago
StackOverflow is probably a better place for questions: could y
your comment? https://stackoverflow.com/ques...

^ | ⌄ • Reply • Share ›

Jan Šťastný ➦ Brian Clozel • a year ago
Ok, Thanks Brian. I have created a new question for thi
https://stackoverflow.com/ques...

^ | ⌄ • Reply • Share ›

Kirandeep Rana • a year ago
Very good and comprehensive explanation..great JOB. I would surly re

1 ^ | ⌄ • Reply • Share ›

bblain7 • a year ago

Never reveal information about your implementation to outside entities

1 ⌃ | ⌄ • Reply • Share ›

Guest • a year ago

I've implemented a convenient exception handler (extends AbstractHa
meets the IETF draft Problem Details for HTTP APIs.
It's very easy to handle custom exceptions without repeating yourself,
customize error responses and even localize them. Also solves some
pitfalls in Spring MVC with a content negotiation when producing an
error response.

You can find it on GitHub under jirutka/spring-rest-exception-handler ar
will be useful for others.

1 ⌃ | ⌄ • Reply • Share ›

Lavesh Singhal ➦ Guest • a year ago

nice explanation.

⌃ | ⌄ • Reply • Share ›

JAEHYUNG cho • a month ago

thank so much

⌃ | ⌄ • Reply • Share ›

Mattias Severson • 9 months ago

Nice! I discovered this post when I was working on my latest blog post
have written similar blog posts before, for example about a custom err
above), and one about generalizing error responses (by using the `@Cor`

^ | ⌄ • Reply • Share ›

Franjo Markovic • 10 months ago

Very nice article, well explained, thanks!

I agree with bblain7 that posting stack trace (even as hidden div) is a p
someone too many details about your configuration. The intended use
stack trace) is also inappropriate - support should have access to the
information.

^ | ⌄ • Reply • Share ›

Gareth Barnard • a year ago

Thank you Paul for this illuminating post.

In 'Using @ExceptionHandler' you have 'Handle exceptions without the
predefined exceptions that you didn't write)' but the example in the san

@ResponseStatus(value=HttpStatus.CONFLICT, reason="Data integ
@ExceptionHandler(DataIntegrityViolationException.class)

So, should the 'without' be 'with'?

^ | ⌄ • Reply • Share ›

Kumar • a year ago

Thank you for the awesome post.

Very good article about best practices.

^ | ⌄ • Reply • Share ›

alienacidtechno · a year ago

Hi

Thanks for the tutorial. I have one concern though. Please don't post s
r.setWarnLogCategory("example.MvcLogger"); or mav.setViewName(
resolver or modelAndView, but it encourages good naming practices a
brain cycles to process it). A lot of people copy-paste code from such
one letter variables everywhere.

Thank you.

· Reply · Share ›

Paul Chapman ↪ alienacidtechno · a year ago

Fair point, no argument from me there. In my defence, I have to
lines in code-snippets short enough to fit on the page and be ea
bars annoying).

Anyway, I hope you found the blog useful and thanks for taking

· Reply · Share ›

SolrWind · a year ago

Thanks, this was a thorough explanation and walkthrough. I needed m
defined in XML for SimpleMappingExceptionResolver. I ended up imple
advised so I could handle exceptions on a more granular basis depend
there's an entire section of my app that communicates with the client s
send a custom JSON structure back. I now get to clean up a lot of my

· Reply · Share ›

ALSO ON **SPRING.IO**

Webinar Replay: A Spring Showcase: Turkcell's …

1 comment • 9 days ago

Spring Boo

13 comments •

Spring Cloud service release: Angel.SR3

2 comments • a month ago

Spring Boo

9 comments • a

Subscribe Add Disqus to your site Privacy

SUBSCRIBE TO OUR NEWSLETTER

Email Address