- Start hiring
- Login

-
- Top 3%
- Why
- Clients
- Partners
- Community
- Blog
- About Us
- Start hiring
- Apply as a Developer
- Login
-
    - Questions?
    - Contact Us

Search Topics

Hire a developer

Guide to Spring Boot REST API Error Handling

View all articles

by **Bruno Leite** - Freelance Java Developer @ Toptal

#APIError #Java #SpringBoot

- 72shares

Handling errors correctly in APIs while providing meaningful error messages is a very desirable feature, as it can help the API client properly respond to issues. The default behavior tends to be returning stack traces that are hard to understand and ultimately useless for the API client. Partitioning the error information into fields also enables the API client to parse it and provide better error messages to the user. In this article, we will cover how to do proper error handling when building a REST API with Spring Boot.

Building REST APIs with Spring became the standard approach for Java developers during the last couple of years. Using Spring Boot helps substantially, as it removes a lot of boilerplate code and enables auto-configuration of various components. We will assume that you're familiar with the basics of API development with those technologies before applying the knowledge described here. If you are still unsure about how to develop a basic REST API, then you should start with this article about Spring MVC or another one about building a Spring REST Service.

Making Error Responses Clearer

Throughout this article, we'll be using the source code hosted on GitHub of an application that implements a REST API for retrieving objects that represent birds. It has the features described in this article and a few more examples of error handling scenarios. Here's a summary of endpoints implemented in that application:

`GET /birds/{birdId}`	Gets information about a bird and throws an exception if not found.
`GET /birds/noexception/{birdId}`	This call also gets information about a bird, except it doesn't throw an exception in case that the bird is not found.
`POST /birds`	Creates a bird.

The Spring framework MVC module comes with some great features to help with error handling. But it is left to the developer to use those features to treat the exceptions and return meaningful responses to the API client.

Let's look at an example of the default Spring Boot answer when we issue an HTTP POST to the `/birds` endpoint with the following JSON object, that has the string "aaa" on the field "mass," which should be expecting an integer:

```
{
 "scientificName": "Common blackbird",
 "specie": "Turdus merula",
 "mass": "aaa",
 "length": 4
}
```

The Spring Boot default answer, without proper error handling:

```
{
 "timestamp": 1500597044204,
 "status": 400,
 "error": "Bad Request",
 "exception": "org.springframework.http.converter.HttpMessageNotReadableException",
 "message": "JSON parse error: Unrecognized token 'three': was expecting ('true', 'false' or 'null'); nested exception is com.fasterxml.jacksc
 "path": "/birds"
}
```

Well… the response message has some good fields, but it is focused too much on what the exception was. By the way, this is the class `DefaultErrorAttributes` from Spring Boot. The `timestamp` field is an integer number that doesn't even carry information of what measurement unit the timestamp is in. The `exception` field is only interesting to Java developers and the message leaves the API consumer lost in all the implementation details that are irrelevant to them. And what if there were more details that we could extract from the exception that the error originated from? So let's learn how to treat those exceptions properly and wrap them into a nicer JSON representation to make life easier for our API clients.

As we'll be using Java 8 date and time classes, we first need to add a Maven dependency for the Jackson JSR310 converters. They take care of converting Java 8 date and time classes to JSON representation using the `@JsonFormat` annotation:

```
<dependency>
   <groupId>com.fasterxml.jackson.datatype</groupId>
   <artifactId>jackson-datatype-jsr310</artifactId>
</dependency>
```

Ok, so let's define a class for representing API errors. We'll be creating a class called `ApiError` that has enough fields to hold relevant information about errors that happen during REST calls.

```
class ApiError {

   private HttpStatus status;
   @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy hh:mm:ss")
   private LocalDateTime timestamp;
   private String message;
   private String debugMessage;
   private List<ApiSubError> subErrors;

   private ApiError() {
       timestamp = LocalDateTime.now();
   }

   ApiError(HttpStatus status) {
       this();
       this.status = status;
   }

   ApiError(HttpStatus status, Throwable ex) {
       this();
       this.status = status;
       this.message = "Unexpected error";
       this.debugMessage = ex.getLocalizedMessage();
   }

   ApiError(HttpStatus status, String message, Throwable ex) {
       this();
       this.status = status;
       this.message = message;
       this.debugMessage = ex.getLocalizedMessage();
   }
}
```

- The `status` property holds the operation call status. It will be anything from 4xx to signalize client errors or 5xx to mean server errors. A common scenario is a http code 400 that means a BAD_REQUEST, when the client, for example, sends an improperly formatted field, like an invalid email address.
- The `timestamp` property holds the date-time instance of when the error happened.
- The `message` property holds a user-friendly message about the error.
- The `debugMessage` property holds a system message describing the error in more detail.
- The `subErrors` property holds an array of sub-errors that happened. This is used for representing multiple errors in a single call. An example would be validation errors in which multiple fields have failed the validation. The `ApiSubError` class is used to encapsulate those.

```
abstract class ApiSubError {

}

@Data
@EqualsAndHashCode(callSuper = false)
@AllArgsConstructor
class ApiValidationError extends ApiSubError {
   private String object;
   private String field;
   private Object rejectedValue;
   private String message;

   ApiValidationError(String object, String message) {
       this.object = object;
       this.message = message;
   }
}
```

So then the `ApiValidationError` is a class that extends `ApiSubError` and expresses validation problems encountered during the REST call.

Below, you'll see some examples of JSON responses that are being generated after we have implemented the improvements described here, just to get an idea of what we'll have by the end of this article.

Here is an example of JSON returned when an entity is not found while calling endpoint `GET /birds/2`:

```
{
 "apierror": {
   "status": "NOT_FOUND",
   "timestamp": "18-07-2017 06:20:19",
   "message": "Bird was not found for parameters {id=2}"
 }
}
```

Here is another example of JSON returned when issuing a `POST /birds` call with an invalid value for the bird's mass:

```
{
 "apierror": {
   "status": "BAD_REQUEST",
   "timestamp": "18-07-2017 06:49:25",
   "message": "Validation errors",
   "subErrors": [
     {
       "object": "bird",
       "field": "mass",
       "rejectedValue": 999999,
       "message": "must be less or equal to 104000"
     }
   ]
 }
}
```

Spring Boot Error Handling

Let's explore some of the Spring annotations that will be used to handle exceptions.

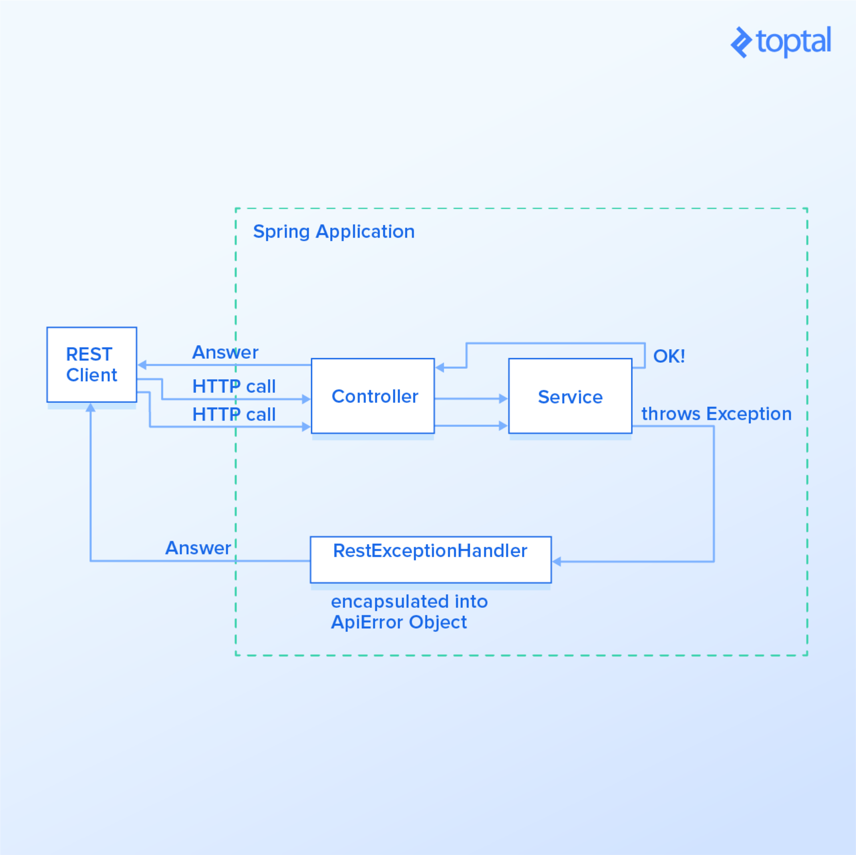
`RestController` is the base annotation for classes that handle REST operations.

`ExceptionHandler` is a Spring annotation that provides a mechanism to treat exceptions that are thrown during execution of handlers (Controller operations). This annotation, if used on methods of controller classes, will serve as the entry point for handling exceptions thrown within this controller only. Altogether, the most common way is to use `@ExceptionHandler` on methods of `@ControllerAdvice` classes so that the exception handling will be applied globally or to a subset of controllers.

`ControllerAdvice` is an annotation introduced in Spring 3.2, and as the name suggests, is "Advice" for multiple controllers. It is used to enable a single `ExceptionHandler` to be applied to multiple controllers. This way we can in just one place define how to treat such an exception and this handler will be called when the exception is thrown from classes that are covered by this `ControllerAdvice`. The subset of controllers affected can defined by using the following selectors on `@ControllerAdvice`: `annotations()`, `basePackageClasses()`, and `basePackages()`. If no selectors are provided, then the `ControllerAdvice` is applied globally to all controllers.

So by using `@ExceptionHandler` and `@ControllerAdvice`, we'll be able to define a central point for treating exceptions and wrapping them up in an `ApiError` object with better organization than the default Spring Boot error handling mechanism.

Handling Exceptions



The next step is to create the class that will handle the exceptions. For simplicity, we are calling it `RestExceptionHandler` and it must extend from Spring Boot's `ResponseEntityExceptionHandler`. We'll be extending `ResponseEntityExceptionHandler` as it already provides some basic handling of Spring MVC exceptions, so we'll be adding handlers for new exceptions while improving the existing ones.

Overriding Exceptions Handled In ResponseEntityExceptionHandler

If you take a look into the source code of `ResponseEntityExceptionHandler`, you'll see a lot of methods called `handle******()` like `handleHttpMessageNotReadable()` or `handleHttpMessageNotWritable()`. Let's first see how can we extend `handleHttpMessageNotReadable()` to handle `HttpMessageNotReadableException` exceptions. We just have to override the method `handleHttpMessageNotReadable()` in our `RestExceptionHandler` class:

```
@Order(Ordered.HIGHEST_PRECEDENCE)
@ControllerAdvice
public class RestExceptionHandler extends ResponseEntityExceptionHandler {

   @Override
   protected ResponseEntity<Object> handleHttpMessageNotReadable(HttpMessageNotReadableException ex, HttpHeaders headers, HttpStatus status, W
       String error = "Malformed JSON request";
       return buildResponseEntity(new ApiError(HttpStatus.BAD_REQUEST, error, ex));
   }

   private ResponseEntity<Object> buildResponseEntity(ApiError apiError) {
       return new ResponseEntity<>(apiError, apiError.getStatus());
   }

   //other exception handlers below

}
```

We have declared that in case of a `HttpMessageNotReadableException` being thrown, the error message will be “Malformed JSON request” and the error will be encapsulated inside the `ApiError` object. Below we can see the answer of a REST call with this new method overridden:

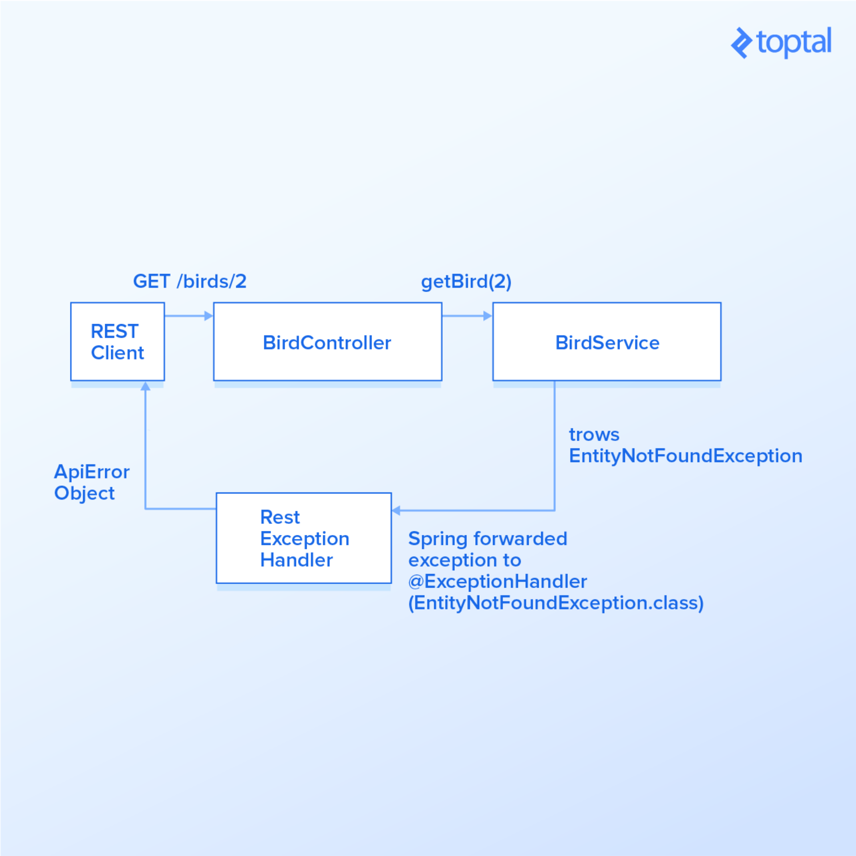
```
{
 "apierror": {
   "status": "BAD_REQUEST",
   "timestamp": "21-07-2017 03:53:39",
   "message": "Malformed JSON request",
   "debugMessage": "JSON parse error: Unrecognized token 'aaa': was expecting ('true', 'false' or 'null'); nested exception is com.fasterxml.j
 }
}
```

Handling Custom Exceptions

Now we'll see how to create a method that handles an exception that is not yet declared inside Spring Boot's `ResponseEntityExceptionHandler`.

A common scenario for a Spring application that handles database calls is to have a call to find a record by its ID using a repository class. But if we look into the `CrudRepository.findOne()` method, we'll see that it returns `null` if an object is not found. That means that if our service just calls this method and returns directly to the controller, we'll get an HTTP code 200 (OK) even if the resource isn't found. In fact, the proper approach is to return a HTTP code 404 (NOT FOUND) as specified in the HTTP/1.1 spec.

To handle this case, we'll be creating a custom exception called `EntityNotFoundException`. This one is a custom created exception and different from `javax.persistence.EntityNotFoundException`, as it provides some constructors that ease the object creation, and one may choose to handle the `javax.persistence` exception differently.


toptal
GET /birds/2
getBird(2)
REST Client
BirdController
BirdService
trows EntityNotFoundException
ApiError Object
Rest Exception Handler
Spring forwarded exception to @ExceptionHandler (EntityNotFoundException.class)

That said, let's create an `ExceptionHandler` for this newly created `EntityNotFoundException` in our `RestExceptionHandler` class. To do that, create a method called `handleEntityNotFound()` and annotate it with `@ExceptionHandler`, passing the class object `EntityNotFoundException.class` to it. This signalizes Spring that every time `EntityNotFoundException` is thrown, Spring should call this method to handle it. When annotating a method with `@ExceptionHandler`, it will accept a wide range of auto-injected parameters like `WebRequest`, `Locale` and others as described here. We'll just provide the exception `EntityNotFoundException` itself as a parameter for this `handleEntityNotFound` method.

```
@Order(Ordered.HIGHEST_PRECEDENCE)
@ControllerAdvice
public class RestExceptionHandler extends ResponseEntityExceptionHandler {

   //other exception handlers

   @ExceptionHandler(EntityNotFoundException.class)
   protected ResponseEntity<Object> handleEntityNotFound(
           EntityNotFoundException ex) {
       ApiError apiError = new ApiError(NOT_FOUND);
       apiError.setMessage(ex.getMessage());
       return buildResponseEntity(apiError);
   }
}
```

Great! In the `handleEntityNotFound()` method, we are setting the HTTP status code to `NOT_FOUND` and using the new exception message. Here is what the response for the `GET /birds/2` endpoint looks like now:

```
{
 "apierror": {
   "status": "NOT_FOUND",
   "timestamp": "21-07-2017 04:02:22",
   "message": "Bird was not found for parameters {id=2}"
 }
}
```

Conclusion

It is important to get in control of the exception handling so we can properly map those exceptions to the `ApiError` object and provide important information that allows API clients to know what happened. The next step from here would be to create more handler methods (the ones with @ExceptionHandler) for exceptions that are thrown within the application code. There are more examples for some other common exceptions like `MethodArgumentTypeMismatchException`, `ConstraintViolationException` and others in the GitHub code.

Here are some additional resources that helped in the composition of this article:

- Baeldung - Error handling for REST with Spring
- Spring Blog - Exception handling in Spring MVC

Understanding the Basics

Why should the API have a uniform error format?

So that the API client can properly parse the error object. A more complex error could make an implementation of the ApiSubError class and provide more details about the problem so that the client can know which actions to take.

How does Spring knows which ExceptionHandler to use?

There is a class called ExceptionHandlerExceptionResolver in Spring MVC. Most of the work happens in the doResolveHandlerMethodException() method.

What information is important to provide to API consumers?

Usually it is important to demonstrate where did the error came from. Were there any input parameters originated the error? It is also important to provide some guidance on how to fix the failing call.

About the author

View full profile »
Hire the Author
Bruno Leite, Brazil
member since March 10, 2014
CSSJavaJavaScriptSQLHibernateGrailsDojo ToolkitFacebook Open Graph APIGoogle MapsGitHubGitNetBeans
Bruno is a full-stack applications architect and senior developer with more than ten years of experience working with startups and agile teams. He works as an architect or leader developer with proven ability to solve challenging problems and is always looking to contribute to team synergy and growth. [click to continue...]
Hiring? Meet the Top 10 Freelance Java Developers for Hire in August 2017

4 Comments **Toptal** **Arefe**

♡ **Recommend** **Share** **Sort by Best**

Join the discussion…

Daniel Echevarria Iparraguirre • a day ago
This article is awesome, great job bro!
1 ^ | v • Reply • Share ›

Bruno ➦ Daniel Echevarria Iparraguirre • a day ago
Thanks Daniel! Glad you liked it.
^ | v • Reply • Share ›

Jordan Demeulenaere • 18 hours ago
Instead of using the @JsonFormat annotation, you could simply set the following property in your application.yml :
spring.jackson.serialization.WRITE_DATES_AS_TIMESTAMPS: false
This will serialize dates in the ISO 8601 format, which IMO is better than what you have because :
1) You don't pollute your data objects with annotations
2) You also have the time zone information (which is needed when the server and client are in different zones)
3) If needed, your javascript frontend could directly create a Date object instead of parsing the date yourself : let date = new Date(response.timestamp);
^ | v • Reply • Share ›

Bruno ➦ Jordan Demeulenaere • 15 hours ago
Great! Good tip Jordan. You are right, better to use the ISO format.
^ | v • Reply • Share ›

✉ **Subscribe** **Add Disqus to your siteAdd DisqusAdd** 🔒 **Privacy**

Subscribe
The #1 Blog for Engineers
Get the latest content first.
Enter your email address
Get Exclusive Updates
No spam. Just great engineering posts.
The #1 Blog for Engineers
Get the latest content first.
Thank you for subscribing!
You can edit your subscription preferences here.

- 121shares
-
-
-

Trending articles

Guide to Spring Boot REST API Error Handlingabout 24 hours ago

Orchestrating a Background Job Workflow in Celery for Python2 days ago

iOS Centralized and Decoupled Networking: AFNetworking Tutorial with a Singleton Class13 days ago

iOS ARKit Tutorial: Drawing in the Air with Bare Fingers14 days ago

A How-to Guide to SVG Animation16 days ago

Guide to Data Synchronization in Microsoft SQL Server19 days ago

The 12 Worst Mistakes Advanced WordPress Developers Make21 days ago

Creating a Ruby DSL: A Guide to Advanced Metaprogramming26 days ago

Relevant Technologies

- Java
- Spring

About the author

Bruno Leite
Java Developer
Bruno is a full-stack applications architect and senior developer with more than ten years of experience working with startups and agile teams. He works as an architect or leader developer with proven ability to solve challenging problems and is always looking to contribute to team synergy and growth.

Hire the Author
Toptal connects the top 3% of freelance talent all over the world.

Toptal Developers

- Android Developers
- AngularJS Developers
- Back-End Developers
- C++ Developers
- Data Scientists
- DevOps Engineers
- Ember.js Developers

- Freelance Developers
- Front-End Developers
- Full Stack Developers
- HTML5 Developers
- iOS Developers
- Java Developers
- JavaScript Developers
- Machine Learning Engineers
- Magento Developers
- Mobile App Developers
- .NET Developers
- Node.js Developers
- PHP Developers
- Python Developers
- React.js Developers
- Ruby Developers
- Ruby on Rails Developers
- Salesforce Developers
- Scala Developers
- Software Developers
- Unity or Unity3D Developers
- Web Developers
- WordPress Developers

See more freelance developers
Learn how enterprises benefit from Toptal experts.

Join the Toptal community.

Hire a developer
or
Apply as a Developer

Highest In-Demand Talent

- iOS Developer
- Front-End Developer
- UX Designer
- UI Designer
- Financial Modeling Consultants
- Interim CFOs

About

- Top 3%
- Clients
- Freelance Developers
- Freelance Designers
- Freelance Finance Experts
- About Us

Contact

- Contact Us
- Press Center
- Careers
- FAQ

Social

- Facebook
- Twitter
- Google+
- LinkedIn

Toptal

Hire the top 3% of freelance talent

- Privacy Policy
- Website Terms

Home › Blog › Guide to Spring Boot REST API Error Handling