295 - Backend

Auftrag Fallstudie 8 – Globales Exceptionhandling

|  |  |
| --- | --- |
| Richtzeit  60 Minuten | Sozialform  Einzelarbeit |

## Beschreibung

Wenn unsere Applikation bereit für den Einsatz ist und wir sie produktiv schalten wollen, müssen wir noch eine wichtige Änderung machen. Stand jetzt werden Exceptions vom Server mit dem Statuscode 500 direkt an den Client weitergegeben. Dabei wird auch der Stacktrace mitgeschickt. So geben wir ungewollt Informationen zu unserem System an den Client weiter. Diese Informationen könnten zum Beispiel für einen Angriff verwendet werden. Zudem sind die Nachrichten für den Benutzer nicht aussagekräftig. Dafür werden wir einen globalen Exceptionhandler umsetzen, der Exceptions global behandelt. So können wir an einem Ort steuern, wie mit verschiedenen Exceptions umgegangen werden soll.

## Auftrag

### Stacktrace

Als erstes kümmern wir uns um den Stacktrace. Um diesen nicht mehr in die Antwort zu packen müssen wir eine Zeile im File application.properties anpassen. Wir stellen die Zeile

server.error.include-stacktrace=*ALWAYS*

um auf

server.error.include-stacktrace=*NEVER*

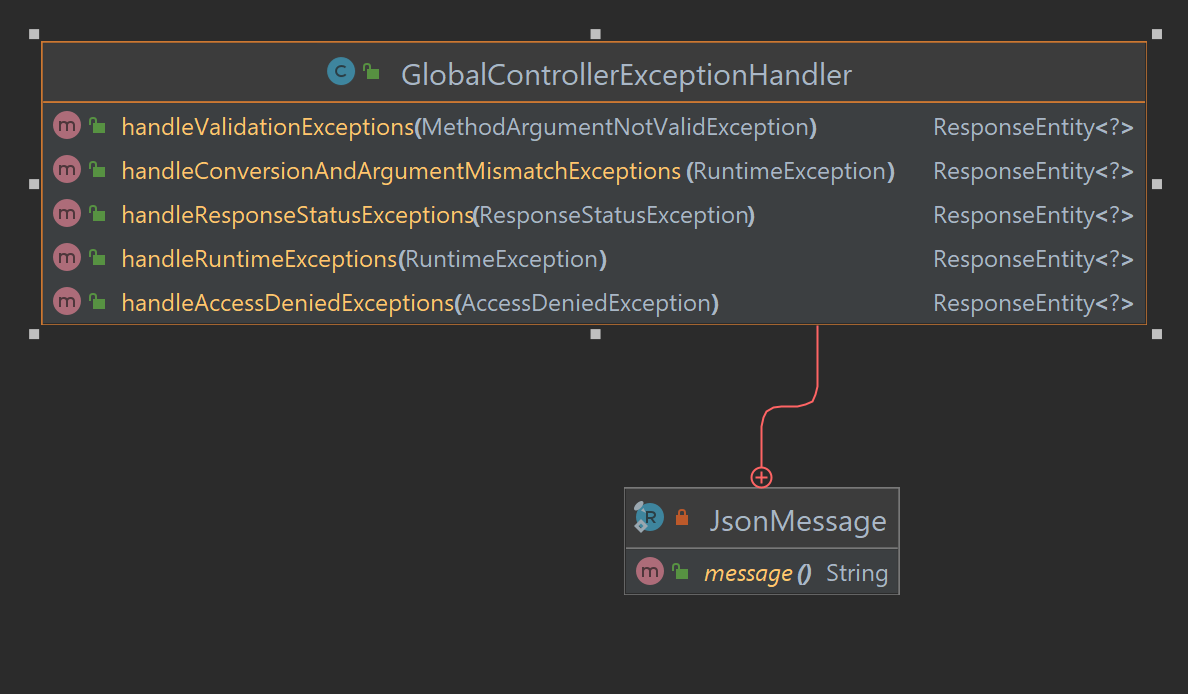
So haben wir sichergestellt, dass wir nicht ungewollt Informationen nach aussen weitergeben.

### GlobalControllerExceptionHandler

Als zweites erstellen wir im Package ch.bbcag.backend.todolist die Klasse GlobalControllerExceptionHandler.

Die Klasse selbst annotieren wir mit @RestControllerAdvice. Bei einer Klasse, die diese Annotation trägt, wird davon ausgegangen, dass es sich dabei um einen Controller zum Behandeln von Exceptions handelt. Darin erstellen wir mit @ExceptionHandler(...) annotiere Methoden. In den Klammern können wir eine oder mehrere Exception-Klassen angeben, welche behandelt werden sollen. Beispielsweise könnte das so aussehen:

@ExceptionHandler({ConversionFailedException.class, MethodArgumentTypeMismatchException.class})

In nebenstehender Grafik siehst du, welche Methoden der Controller haben soll. Es wird jedoch auf jede Methode noch einzeln eingegangen. Auch hier werden die Zurückgegebenen Werte vom Typ ResponseEntity<?> sein, wie bei für jede Response. Zusätzlich siehst du noch eine innere Klasse namens JsonMessage. Dabei handelt es sich um einen Record, also ein Klasse, die eine Instanzvariable beinhaltet, die nur im Konstruktor gesetzt und dann auch nur gelesen werden kann. Du musst nicht genau verstehen, was ein Record ist, jedoch aber wieso wir diesen brauchen. Wir könnten das auch mit einer normalen Klasse machen, die Records bieten sich jedoch genau dafür an.

Wir haben das Problem, dass wenn wir einen String in den Body der ResponseEntity geben, dieser einfach als blosser Text reingeschrieben wird. Wir erhalten also kein JSON. Um dieses Problem zu lösen haben wir den Record, der als JSON-Objekt serialisiert wird.

private record JsonMessage(String message) {  
}

So sieht dieser Record aus. Du kannst den Code einfach abschreiben.

#### handleConversionAndArgumentMismatchExceptions

Diese Methode soll alle Exceptions von den Typen ConversionFailedException.class und MethodArgumentTypeMismatchException.class behandeln. Die Methode soll eine ResponseEntity mit dem Statuscode 400 BAD REQUEST zurückgeben und als Body die Message der Exception in Form einer JsonMessage. Das sieht dann so aus:

@ExceptionHandler({ConversionFailedException.class, MethodArgumentTypeMismatchException.class})  
public ResponseEntity<?> handleConversionAndArgumentMismatchExceptions(RuntimeException ex) {  
 return ResponseEntity.*status*(HttpStatus.*BAD\_REQUEST*).body(new JsonMessage(ex.getMessage()));  
}

#### handleResponseStatusExceptions

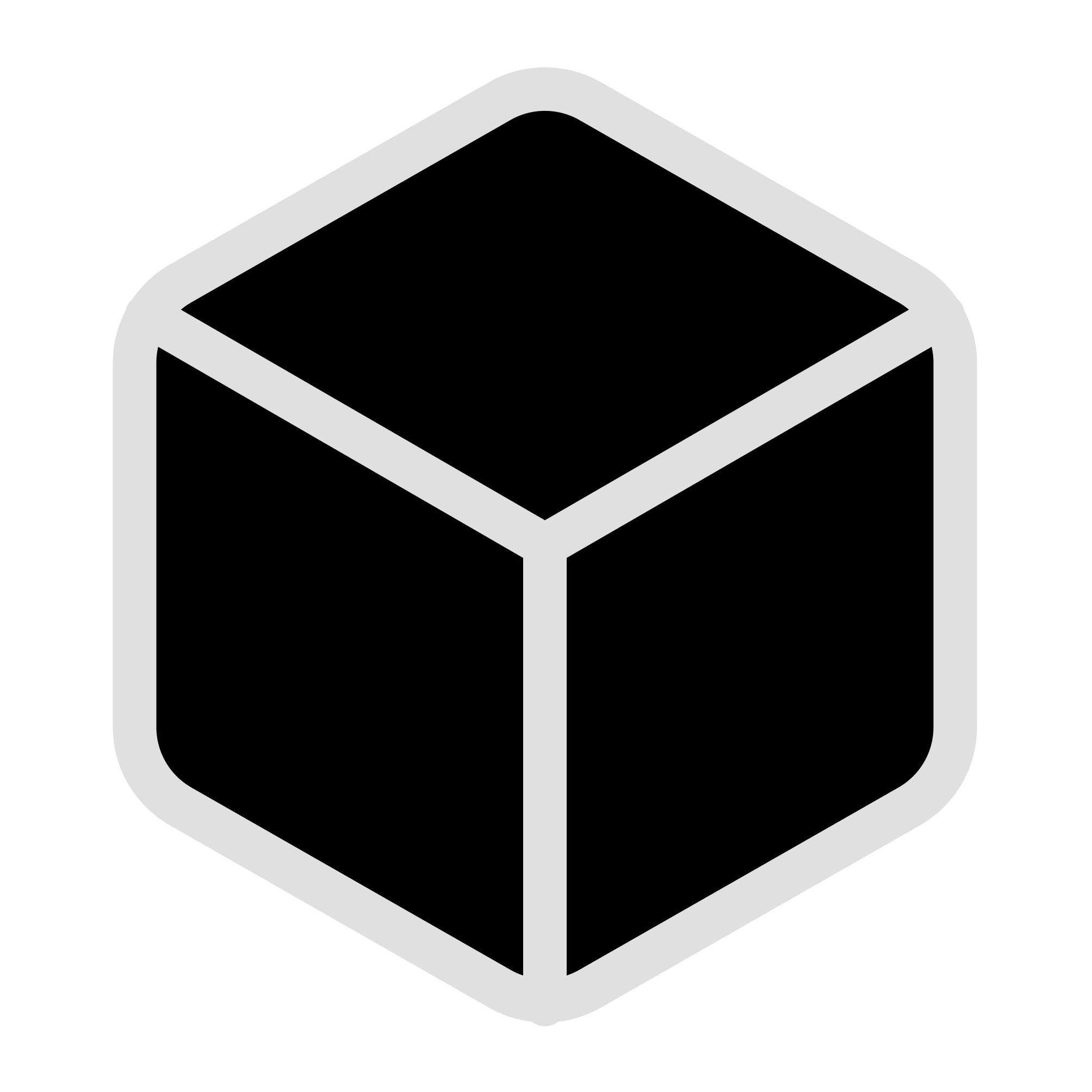
Diese Methode soll alle Exceptions vom Typ ResponseStatusException.class behandeln. Der Statuscode ist in der Exception und kann mit getStatus() übernommen werden. Als Nachricht geben wir den Grund an, welcher mit getReason() aus der Exception geholt werden kann. Auch hier wieder in Form einer JsonMessage.  
Das sind die Exceptions, die du explizit in deinen Controllern wirfst. Würden wir diese nicht catchen, so würden hier einfach viel mehr Informationen an den Client gesendet.

#### handleValidationExceptions

Hier sollen alle Exceptions vom Typ MethodArgumentNotValidException und deine eigene Exception, falls die Validierung in der merge-Methode fehlschlägt, abgefangen werden. Die Signatur soll wie folgt aussehen:

@ExceptionHandler({FailedValidationException.class, MethodArgumentNotValidException.class})  
public ResponseEntity<?> handleValidationExceptions(Exception ex)

Innerhalb musst du mit instanceof testen, welche der beiden Exceptions abgefangen wurde. Bei der FailedValidationException kannst du errors direkt in eine ResponseEntity geben.

Bei der MethodArgumentNotValidException wird es etwas komplexer, deshalb kannst du nachfolgenden Code abschreiben.

final Map<String, List<String>> errors = new HashMap<>();  
ex.getBindingResult().getAllErrors().forEach((error) -> {  
 final String fieldName = ((FieldError) error).getField();  
 final String errorMessage = error.getDefaultMessage();  
  
 if (!errors.containsKey(fieldName))  
 errors.put(fieldName, new ArrayList<>());  
  
 errors.get(fieldName).add(errorMessage);  
});

Gib am Schluss noch eine ResponseEntity mit dem Statuscode 400 BAD REQUEST und der variable errors im Body zurück. Zusätzlich wollen wir, dass in der Dokumentation nicht bei jedem Request «400 Bad Request» steht.

#### handleRuntimeExceptions

Hier werden alle RuntimeExceptions behandelt. Dabei handelt es sich meist um unerwartete Exceptions. Wir setzten den Statuscode 500 INTERNAL SERVER ERROR und geben die Message der Exception in den Body. In einem Produktivsystem wäre es hier auch sinnvoll, den Fehler zu loggen.

Wenn eine InvalidDataAccessApiUsageException geworfen wird, so kann die Ursache dafür sein, dass es sich um einen nicht existierenden Fremdschlüssel in der DB handelt. Dies erkennen wir daran, wenn der Cause eine TransientPropertyValueException ist. Dafür verwenden wir folgenden Code:

@ExceptionHandler({RuntimeException.class})  
public ResponseEntity<?> handleRuntimeExceptions(RuntimeException ex) {  
 if (ex instanceof InvalidDataAccessApiUsageException){  
 if (ex.getCause() != null && ex.getCause().getCause() instanceof TransientPropertyValueException tex) {  
 return ResponseEntity.*status*(HttpStatus.*CONFLICT*).body(new JsonMessage(tex.getPropertyName() + " must be valid"));  
 }  
 }  
 return ResponseEntity.*status*(HttpStatus.*INTERNAL\_SERVER\_ERROR*).body(new JsonMessage(ex.getMessage()));  
}

#### handleAccessDeniedExceptions

Als letztes behandeln wir noch die AccessDeniedException.class. Hier setzen wir den Statuscode 403 FORBIDDEN und geben die Message der Exception in den Body. Diese würde von Spring geworfen, wenn Berechtigungen fehlen.