Exception handling Approaches

in Spring Boot Application

Prepared by:

Akshay Kumar K
Associate Software developer
Valtech India.

Contents

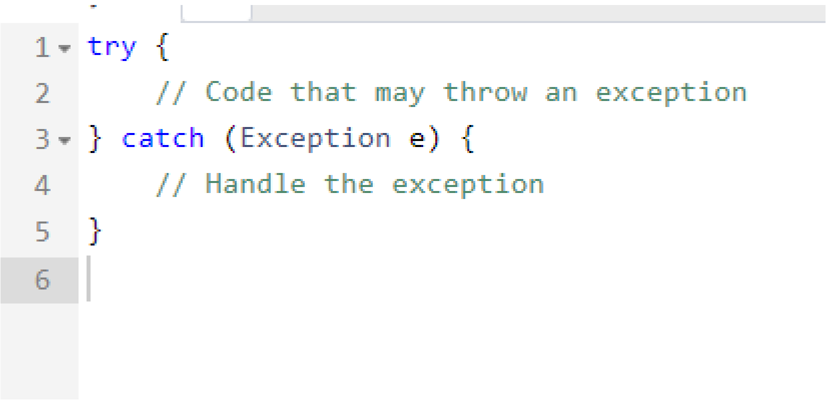
1. Introduction

Exception handling is an important part of any application, including Spring Boot applications. In a Spring Boot application, exceptions can occur in various parts of the application, such as controllers, services, or repositories.

In a Spring Boot application, there are different approaches to exception handling. In this document, we will cover some of the most common approaches, along with examples.

2. Try-Catch Block:

The try-catch block is a traditional approach to exception handling. In this approach, you wrap the code that may throw an exception inside a try block. If an exception is thrown, it is caught by the catch block, where you can handle the exception accordingly. Here's an example:

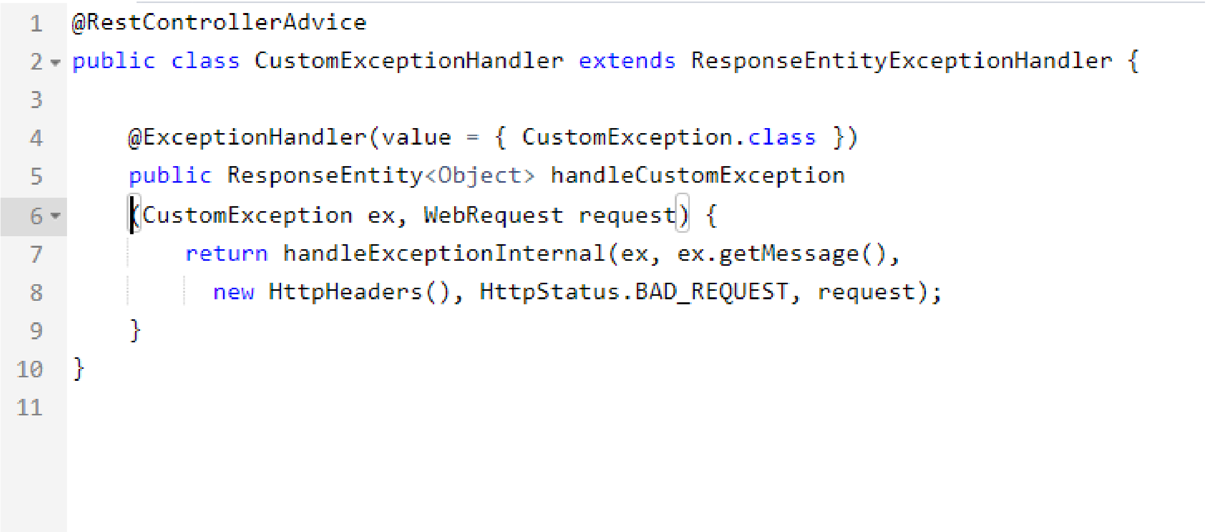


```
try {
    // Code that may throw an exception
} catch (Exception e) {
    // Handle the exception
}
```

While the try-catch block is a straightforward approach to exception handling, it can lead to code duplication and make your code harder to maintain.

3. Spring's ResponseEntityExceptionHandler:

Spring provides the ResponseEntityExceptionHandler class, which is a base class for handling exceptions in RESTful web services. This class provides a set of methods for handling common HTTP status codes and returning custom error messages. Here's an example:

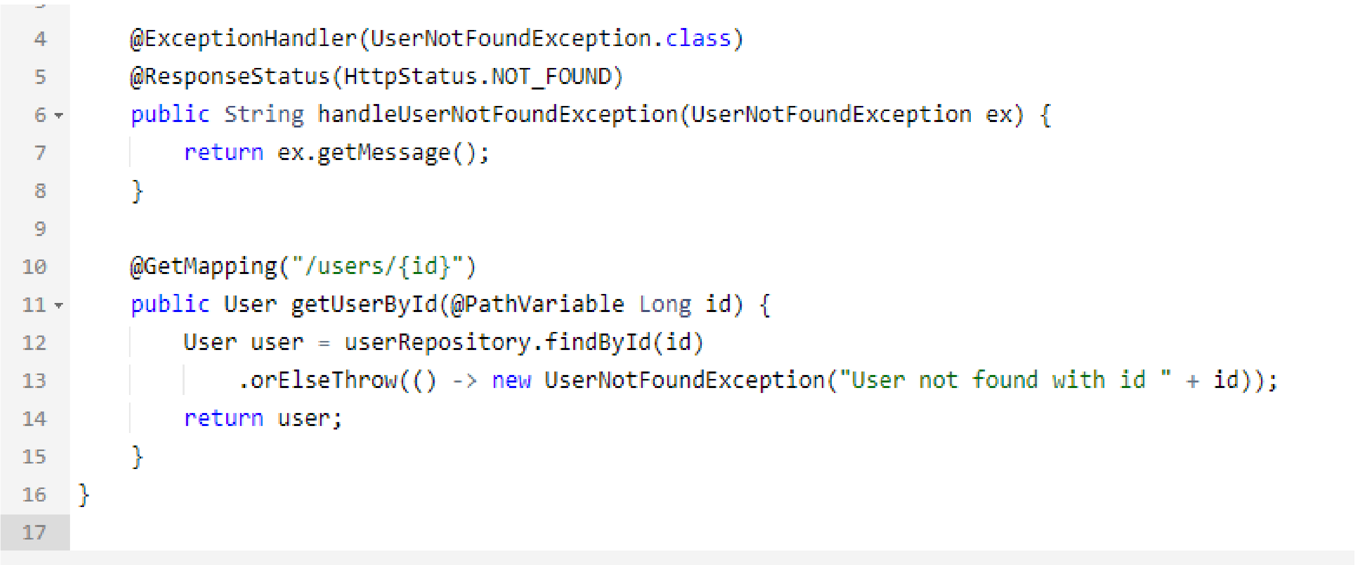


```java
@RestControllerAdvice
public class CustomExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = { CustomException.class })
    public ResponseEntity<Object> handleCustomException
    (CustomException ex, WebRequest request) {
        return handleExceptionInternal(ex, ex.getMessage(),
          new HttpHeaders(), HttpStatus.BAD_REQUEST, request);
    }
}
```

In this example, we define a custom exception handler by extending the ResponseEntityExceptionHandler class and annotating it with @RestControllerAdvice. We also define a method annotated with @ExceptionHandler, which handles a custom exception (CustomException) and returns a ResponseEntity with a custom error message.

4. Spring's @ExceptionHandler Annotation:

@ExceptionHandler is a Spring annotation that can be used to handle exceptions thrown by a controller method. To use @ExceptionHandler, you need to define a method in your controller class and annotate it with @ExceptionHandler along with the type of exception that it handles. For example:

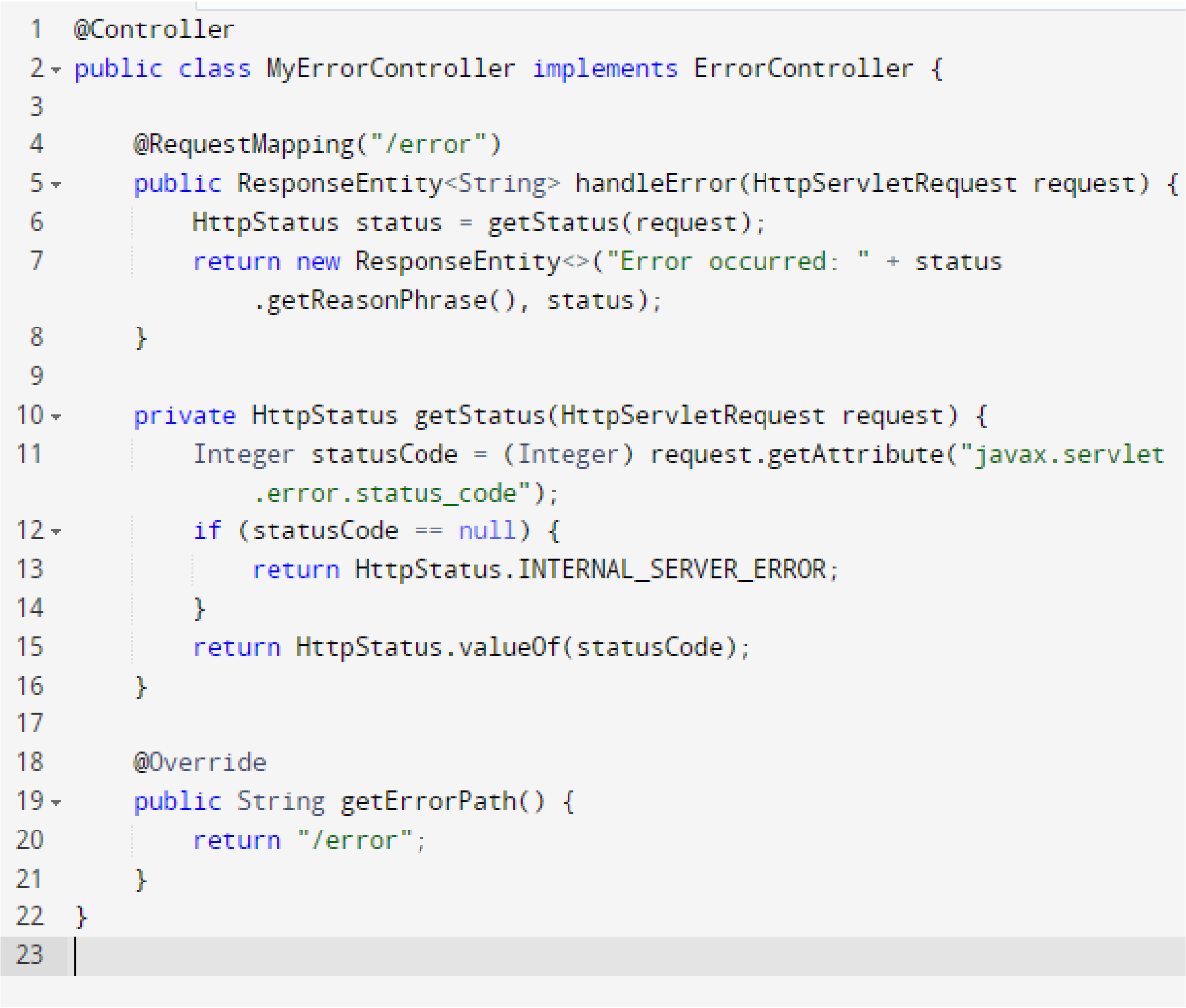


```java
    @ExceptionHandler(UserNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String handleUserNotFoundException(UserNotFoundException ex) {
        return ex.getMessage();
    }

    @GetMapping("/users/{id}")
    public User getUserById(@PathVariable Long id) {
        User user = userRepository.findById(id)
            .orElseThrow(() -> new UserNotFoundException("User not found with id " + id));
        return user;
    }
}
```

In this example, we have defined a method called handleUserNotFoundException that handles UserNotFoundException. When this exception is thrown from the getUserById method, it is caught by the handleUserNotFoundException method and a custom error message is returned.

5. Spring Boot's ErrorController:

Spring Boot's ErrorController interface provides a way to handle all types of exceptions in a single place. When an exception occurs, Spring Boot looks for a bean that implements the ErrorController interface and calls its getErrorPath() method. You can customize the error response returned to the client by implementing the ErrorController interface. Here's an example:

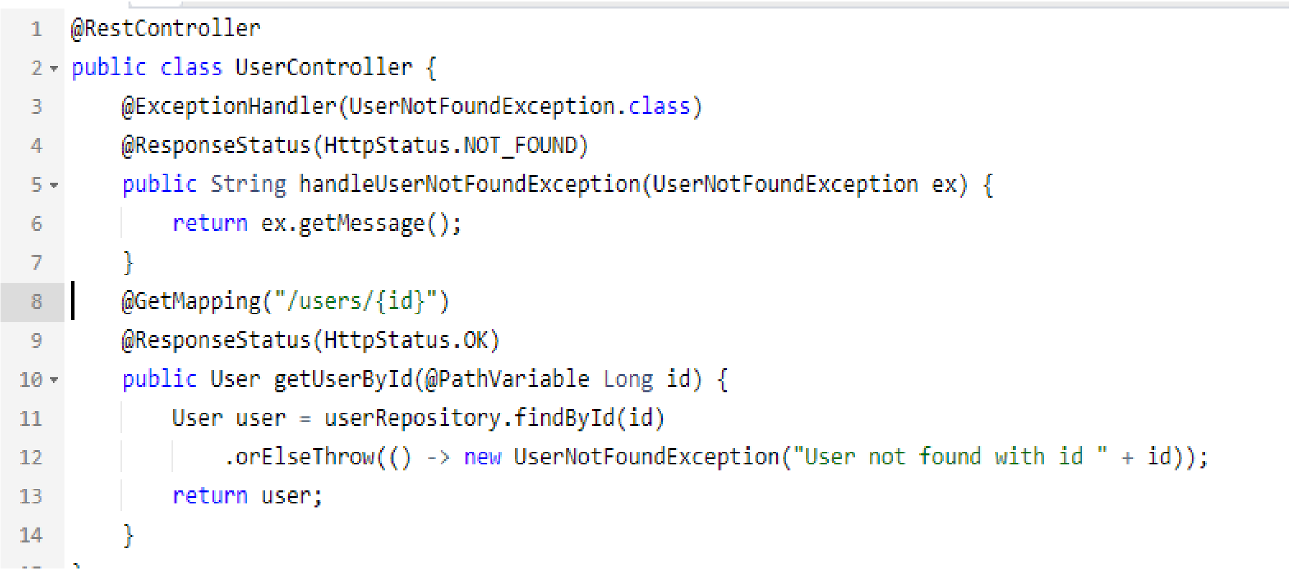


```java
@Controller
public class MyErrorController implements ErrorController {

    @RequestMapping("/error")
    public ResponseEntity<String> handleError(HttpServletRequest request) {
        HttpStatus status = getStatus(request);
        return new ResponseEntity<>("Error occurred: " + status.getReasonPhrase(), status);
    }

    private HttpStatus getStatus(HttpServletRequest request) {
        Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status_code");
        if (statusCode == null) {
            return HttpStatus.INTERNAL_SERVER_ERROR;
        }
        return HttpStatus.valueOf(statusCode);
    }

    @Override
    public String getErrorPath() {
        return "/error";
    }
}

```

In this example, we define a controller class that implements the ErrorController interface. We define a method that handles all types of exceptions by mapping to the /error endpoint. The HttpStatus returned depends on the type of exception that occurred. We also implement the getErrorPath() method, which returns the endpoint that handles all types of exceptions.

6. Spring's @ResponseStatus

@ResponseStatus is a Spring annotation that can be used to define the HTTP status code to be returned in case of an exception. You can use @ResponseStatus along with @ExceptionHandler to handle exceptions and return a specific HTTP status code. Here's an example:



```
@RestController
public class UserController {
    @ExceptionHandler(UserNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String handleUserNotFoundException(UserNotFoundException ex) {
        return ex.getMessage();
    }
    @GetMapping("/users/{id}")
    @ResponseStatus(HttpStatus.OK)
    public User getUserById(@PathVariable Long id) {
        User user = userRepository.findById(id)
            .orElseThrow(() -> new UserNotFoundException("User not found with id " + id));
        return user;
    }
}
```

In this example, we have defined @ResponseStatus(HttpStatus.OK) for the getUserById method, which means that if an exception is not thrown, the HTTP status code returned will be 200 OK.

7. Spring's @ControllerAdvice:

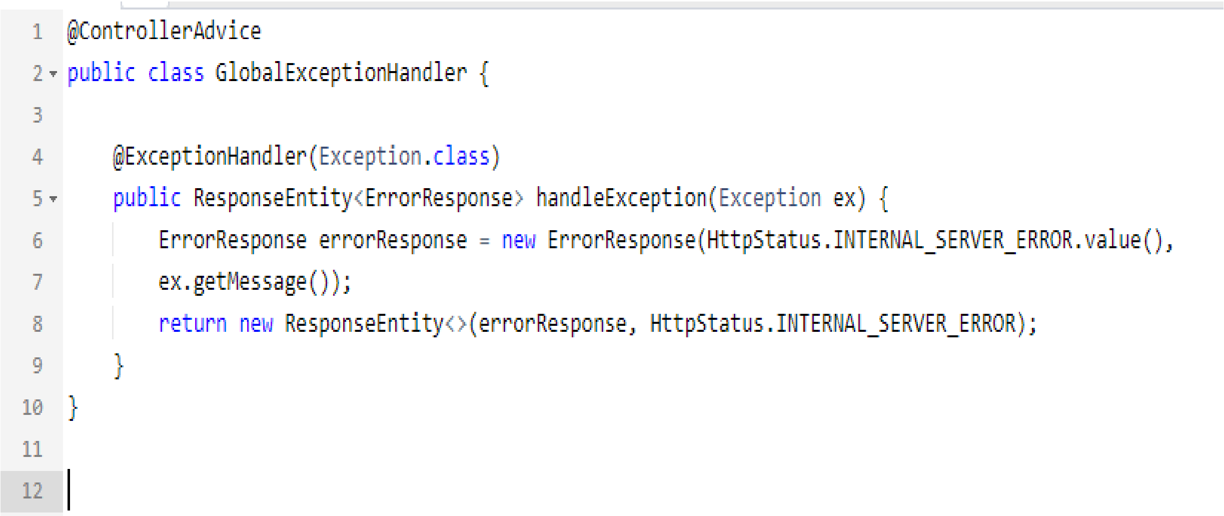
@ControllerAdvice is a Spring annotation that can be used to define global exception handling for all controllers in the application.

However, writing exception handling code in every method or controller can lead to code duplication and make the code less maintainable. To address this issue, Spring Boot provides a global exception handling mechanism that allows you to handle exceptions in a centralized location.

Implementing Global Exception Handling

To implement global exception handling in a Spring Boot application, you can define a class annotated with @ControllerAdvice. This class will contain one or more methods annotated with @ExceptionHandler, which will handle the exceptions thrown by the application.

Here is an example of how to implement global exception handling in a Spring Boot application:

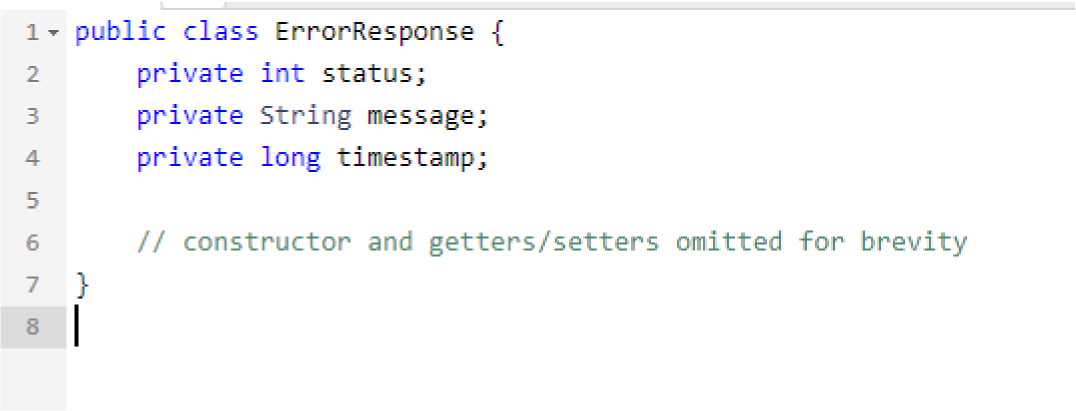


```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleException(Exception ex) {
        ErrorResponse errorResponse = new ErrorResponse(HttpStatus.INTERNAL_SERVER_ERROR.value(),
        ex.getMessage());
        return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

In this example, we define a class called GlobalExceptionHandler that is annotated with @ControllerAdvice. This class contains a method called handleException that is annotated with @ExceptionHandler. This method takes an Exception parameter and returns a ResponseEntity that contains an ErrorResponse object and an HTTP status code.

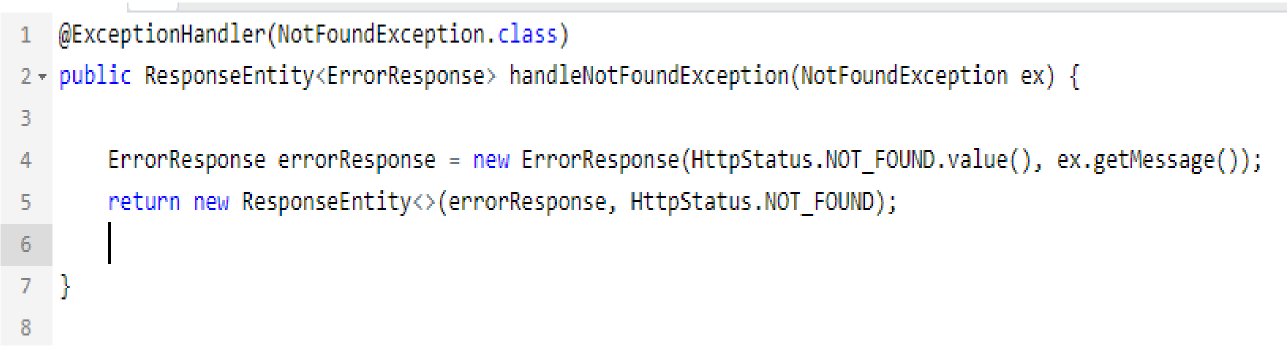
The handleException method will handle any exception thrown by the application that is not handled by other exception handlers. In this case, it creates an ErrorResponse object that contains the HTTP status code and the error message, and returns it as a ResponseEntity.

You can customize the ErrorResponse object to include additional information, such as the timestamp, the URL of the request, or the stack trace of the exception.



```java
public class ErrorResponse {
    private int status;
    private String message;
    private long timestamp;

    // constructor and getters/setters omitted for brevity
}
```

You can also define specific exception handlers for different types of exceptions. For example, to handle a NotFoundException, you can define a method like this:



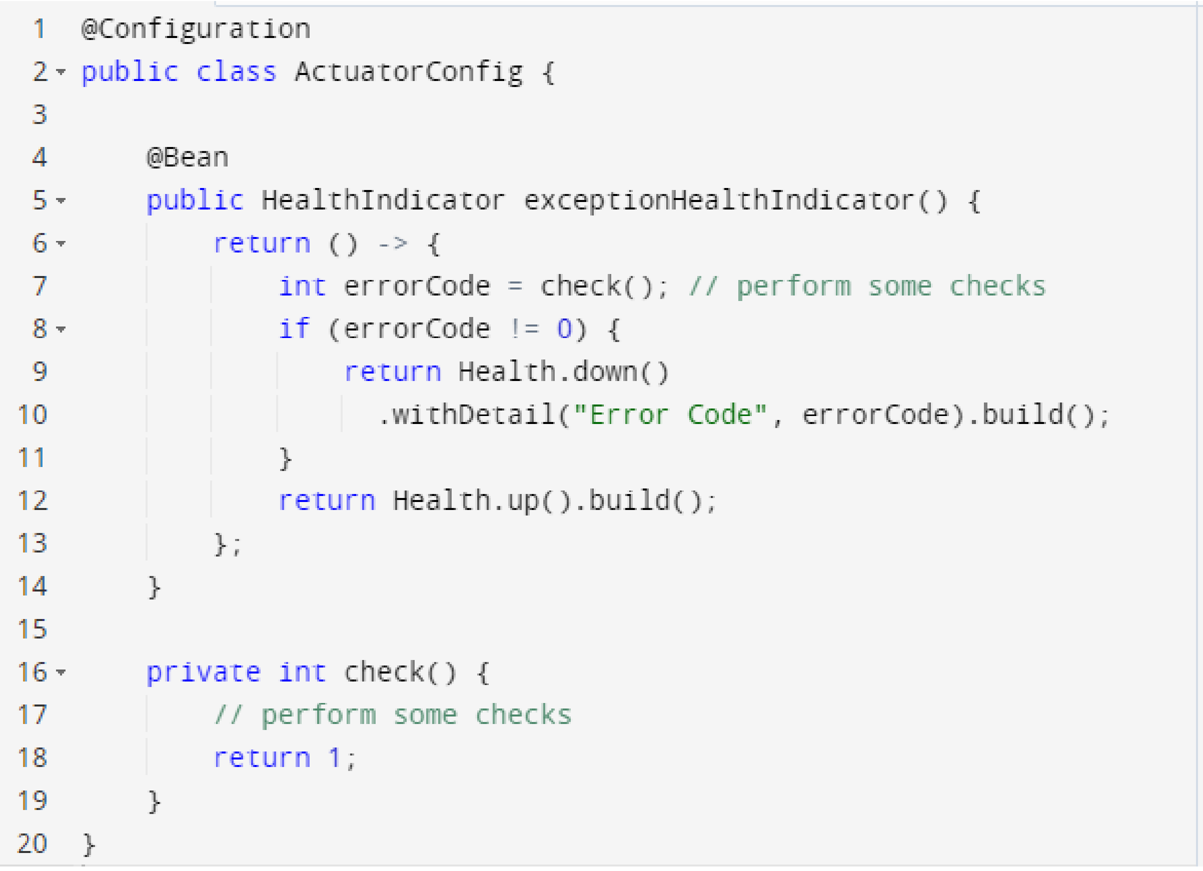
```java
@ExceptionHandler(NotFoundException.class)
public ResponseEntity<ErrorResponse> handleNotFoundException(NotFoundException ex) {

    ErrorResponse errorResponse = new ErrorResponse(HttpStatus.NOT_FOUND.value(), ex.getMessage());
    return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);

}
```

This method will handle the NotFoundException thrown by the application and return a ResponseEntity with the appropriate HTTP status code and error message.

8. Spring Boot's Actuator:

Spring Boot's Actuator provides a way to monitor and manage your application's health, metrics, and more. One of the features of the Actuator is the ability to expose information about your application's exceptions. By default, Spring Boot's Actuator exposes a /actuator/health endpoint that returns the status of your application's health. You can customize the

/actuator/health endpoint to include information about your application's exceptions. Here's an example:

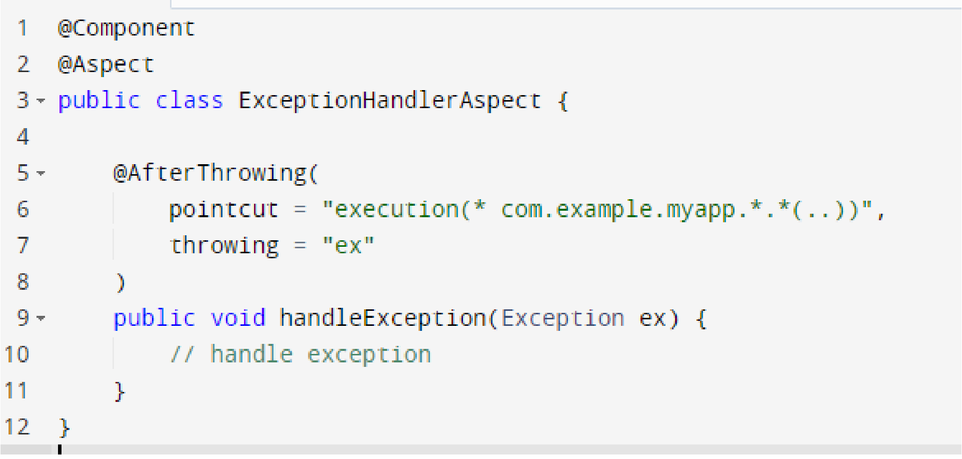


```
@Configuration
public class ActuatorConfig {

    @Bean
    public HealthIndicator exceptionHealthIndicator() {
        return () -> {
            int errorCode = check(); // perform some checks
            if (errorCode != 0) {
                return Health.down()
                    .withDetail("Error Code", errorCode).build();
            }
            return Health.up().build();
        };
    }

    private int check() {
        // perform some checks
        return 1;
    }
}
```

In this example, we define a configuration class that creates a HealthIndicator bean. The HealthIndicator bean checks the status of the application and returns a Health object that describes the application's health. We also define a check() method that performs some checks and returns an error code if the checks fail. If the checks fail, the Health object returned by the HealthIndicator includes the error code.

9. Using AOP:

Aspect-Oriented Programming (AOP) is a programming paradigm that allows you to separate cross-cutting concerns, such as logging, security, and exception handling, from the rest of your code. In the context of exception handling, you can use AOP to define a separate class that handles all exceptions thrown by your application. Here's an example:

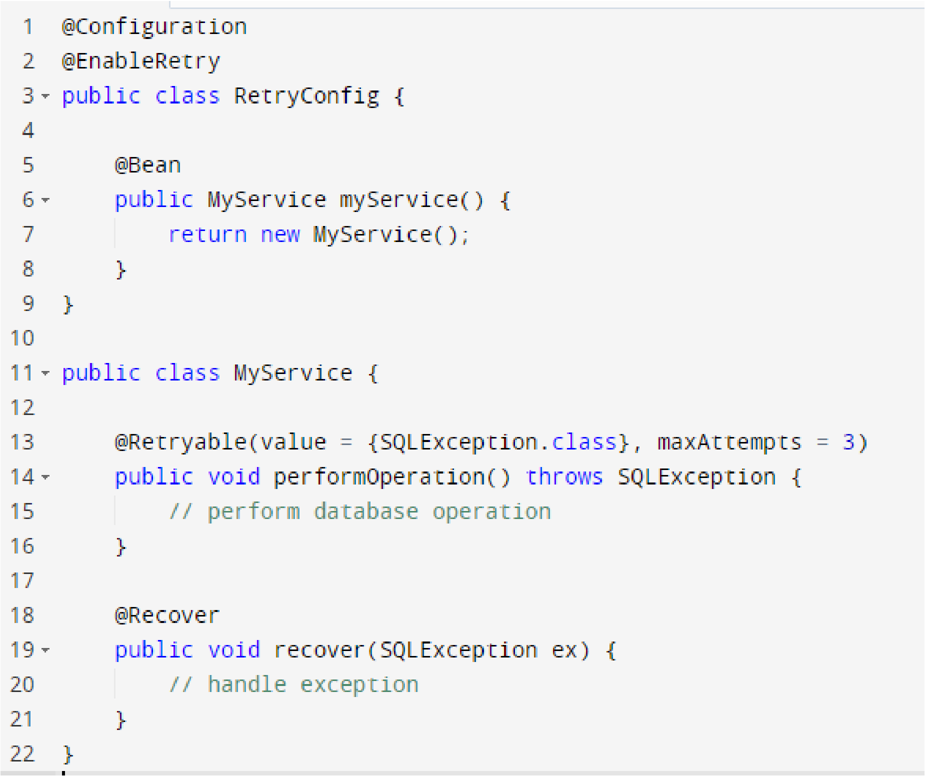


```java
@Component
@Aspect
public class ExceptionHandlerAspect {

    @AfterThrowing(
        pointcut = "execution(* com.example.myapp.*.*(..))",
        throwing = "ex"
    )
    public void handleException(Exception ex) {
        // handle exception
    }
}
```

In this example, we define an ExceptionHandlerAspect class that implements the @Aspect annotation. We also define a method that handles exceptions thrown by the com.example.myapp package. When an exception is thrown by a method in the com.example.myapp package, the handleException() method is called to handle the exception.

10. Using Spring Retry:

Spring Retry is a Spring Boot library that provides a way to retry operations that fail due to exceptions. You can use Spring Retry to retry operations that are likely to fail, such as database or network operations. Here's an example:

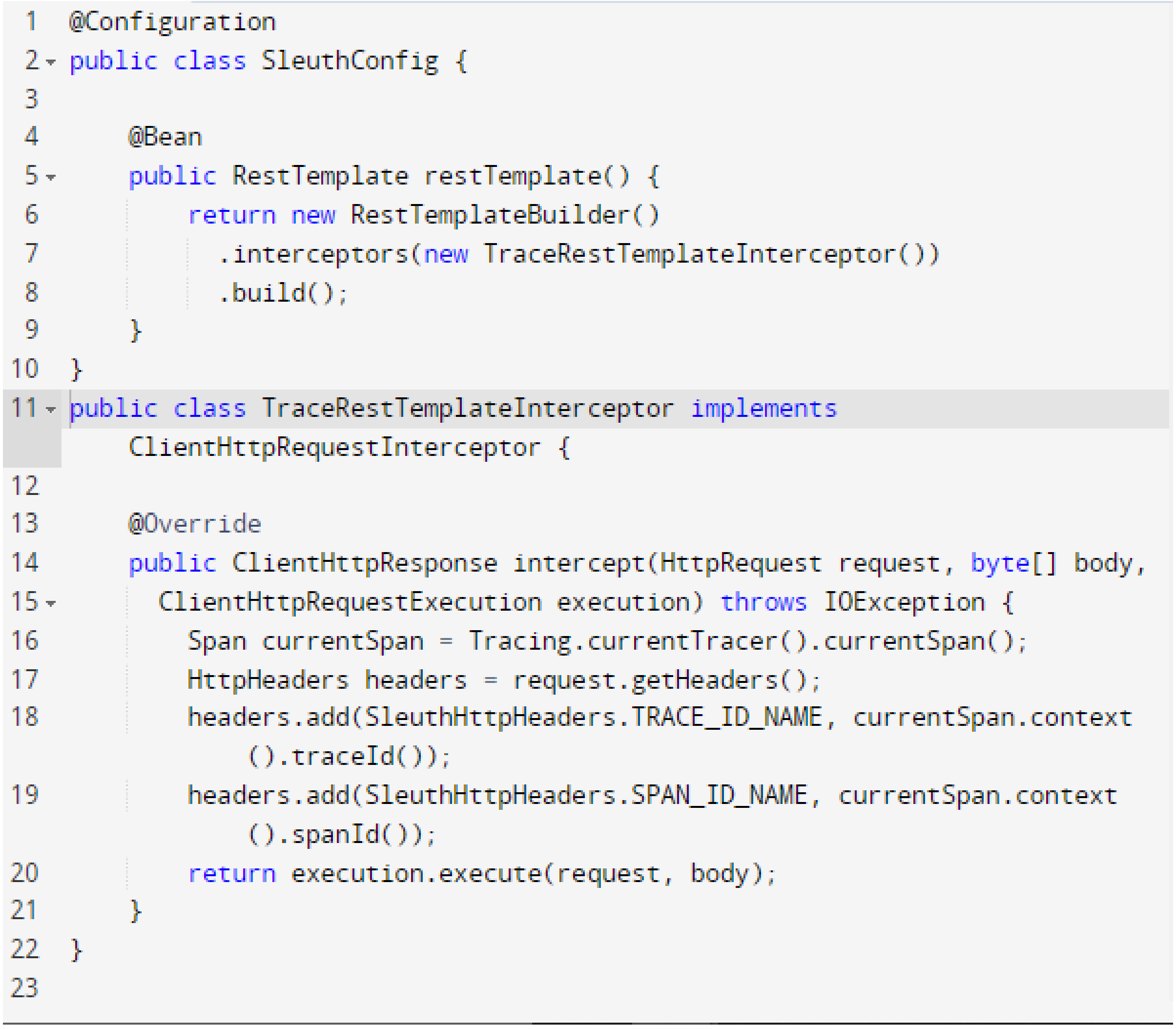


```java
@Configuration
@EnableRetry
public class RetryConfig {

    @Bean
    public MyService myService() {
        return new MyService();
    }
}

public class MyService {

    @Retryable(value = {SQLException.class}, maxAttempts = 3)
    public void performOperation() throws SQLException {
        // perform database operation
    }

    @Recover
    public void recover(SQLException ex) {
        // handle exception
    }
}
```

In this example, we define a MyService class that performs a database operation. The performOperation() method is annotated with @Retryable, which indicates that the method should be retried if a SQLException is thrown. The maxAttempts parameter specifies the maximum number of times the method should be retried. The recover() method is annotated with @Recover, which indicates that the method should be called if the performOperation() method fails to execute after the maximum number of retries.

11. Using Spring Cloud Sleuth:

Spring Cloud Sleuth is a Spring Boot library that provides distributed tracing capabilities. You can use Spring Cloud Sleuth to trace requests through your application and identify where exceptions occur. Here's an example:

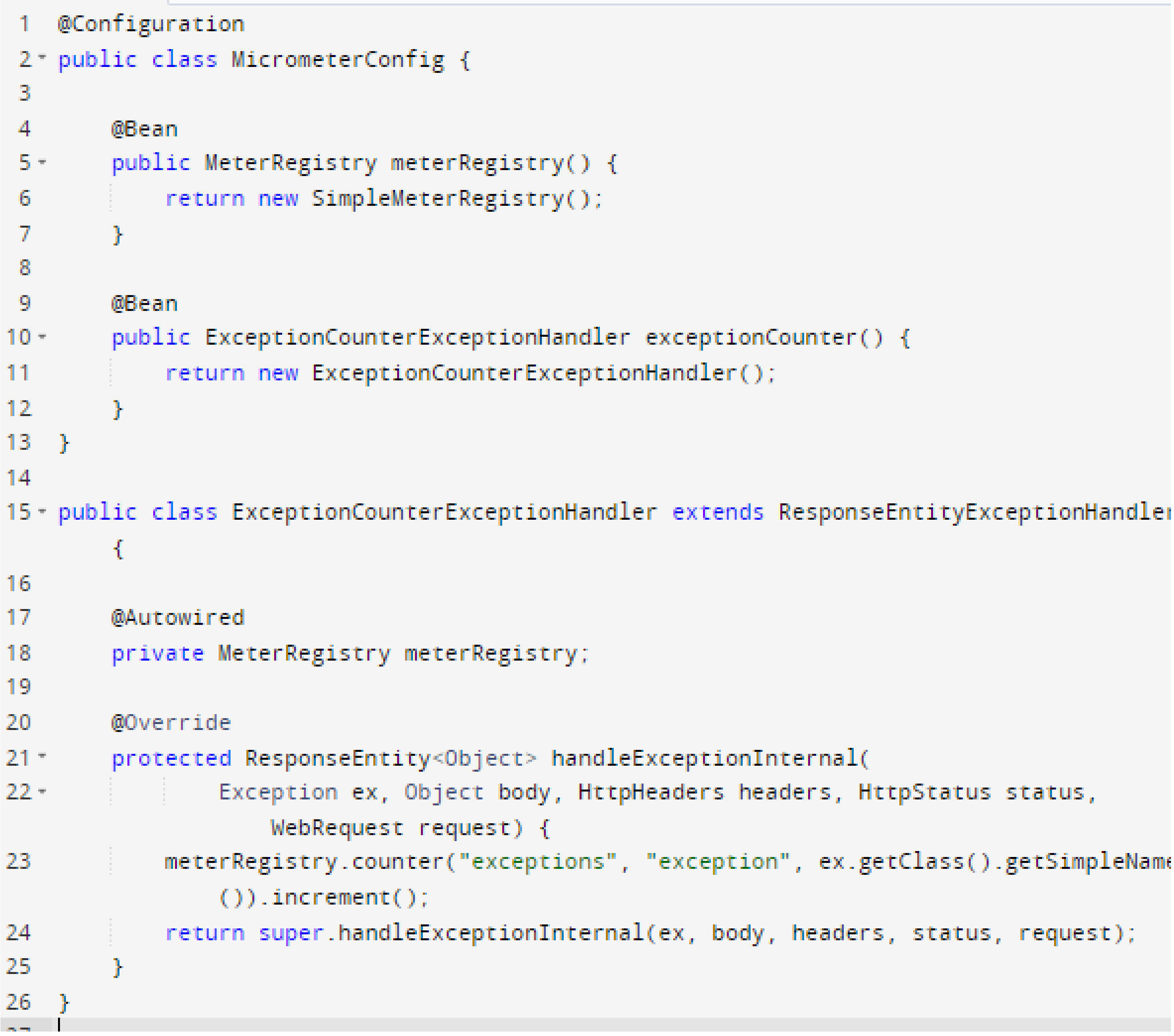


```
@Configuration
public class SleuthConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplateBuilder()
          .interceptors(new TraceRestTemplateInterceptor())
          .build();
    }
}
public class TraceRestTemplateInterceptor implements
    ClientHttpRequestInterceptor {

    @Override
    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
      ClientHttpRequestExecution execution) throws IOException {
        Span currentSpan = Tracing.currentTracer().currentSpan();
        HttpHeaders headers = request.getHeaders();
        headers.add(SleuthHttpHeaders.TRACE_ID_NAME, currentSpan.context
            ().traceId());
        headers.add(SleuthHttpHeaders.SPAN_ID_NAME, currentSpan.context
            ().spanId());
        return execution.execute(request, body);
    }
}

```

In this example, we define a TraceRestTemplateInterceptor class that intercepts HTTP requests and adds tracing information to the request headers. We also define a RestTemplate bean that uses the TraceRestTemplateInterceptor to intercept HTTP requests. When an exception occurs, you can use the tracing information to identify where the exception occurred and diagnose the issue.

12. Using Micrometer:

Micrometer is a Spring Boot library that provides a way to monitor and measure your application's performance. You can use Micrometer to monitor the rate at which exceptions occur in your application. Here's an example:



```
@Configuration
public class MicrometerConfig {

    @Bean
    public MeterRegistry meterRegistry() {
        return new SimpleMeterRegistry();
    }

    @Bean
    public ExceptionCounterExceptionHandler exceptionCounter() {
        return new ExceptionCounterExceptionHandler();
    }
}

public class ExceptionCounterExceptionHandler extends ResponseEntityExceptionHandle
    {

    @Autowired
    private MeterRegistry meterRegistry;

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(
            Exception ex, Object body, HttpHeaders headers, HttpStatus status,
                WebRequest request) {
        meterRegistry.counter("exceptions", "exception", ex.getClass().getSimpleName
            ()).increment();
        return super.handleExceptionInternal(ex, body, headers, status, request);
    }
}
```

In this example, we define a MicrometerConfig class that creates a MeterRegistry bean and an `ExceptionCounterExceptionHandler

Conclusion

In conclusion, there are several approaches to exception handling in Spring Boot applications, each with its own advantages and use cases.

The built-in exception handling capabilities of Spring Boot, such as @ExceptionHandler and @ControllerAdvice, provide a simple and effective way to handle exceptions within a single controller or across multiple controllers.

Using a custom exception handler, such as ResponseEntityExceptionHandler, allows you to customize the response returned to the client when an exception occurs.

Global exception handling with @RestControllerAdvice can be used to handle exceptions across all controllers in your application, providing a centralized and consistent way to handle exceptions.

Using AOP and Spring AOP allows you to apply exception handling logic to multiple methods in your application, reducing code duplication and providing a way to handle exceptions in a cross-cutting manner.

Other approaches, such as using Spring Retry, Spring Cloud Sleuth, and Micrometer, provide additional capabilities for handling and monitoring exceptions in your Spring Boot applications.

Choosing the right approach to exception handling depends on the specific needs and requirements of your application, but with the wide range of options available in Spring Boot, you can be sure to find an approach that suits your needs.