

[DSA](#) [Practice Problems](#) [C](#) [C++](#) [Java](#) [Python](#) [JavaScript](#) [Data Science](#) [Machine Learning](#) [C](#)

# Exception Handling in Spring Boot

Last Updated : 23 Jul, 2025

**Exception handling in Spring Boot** helps deal with errors and exceptions present in APIs, delivering a robust enterprise application. This article covers various ways in which exceptions can be handled and how to return meaningful error responses to the client in a Spring Boot Project.

## Key Approaches to Exception Handling in Spring Boot

Here are some key approaches to exception handling in Spring Boot:

- Default exception handling by Spring Boot
- Using `@ExceptionHandler` annotation
- Using `@ControllerAdvice` for global exception handling

## Spring Boot Exception Handling Simple Example Project

Let's do the initial setup to explore each approach in more depth.

### Initial Setup

To create a simple Spring Boot project using Spring Initializer, please refer to [this article](#). Now let's develop a Spring Boot RESTful web service that performs CRUD operations on a **Customer** Entity. We will be using a MYSQL database for storing all necessary data.

### Step 1: Creating a JPA Entity Class `Customer`

We will create a Customer class, which represents the entity for our database. The class is annotated with `@Entity`.

```
// Creating a JPA Entity class Customer with
// three fields: id, name, and address
package com.customer.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String address;
    private String email;
}
```

The Customer class is annotated with `@Entity` annotation and defines getters, setters, and constructors for the fields.

## Step 2: Creating a CustomerRepository Interface

Next, we create the repository interface for CRUD operations on the Customer entity. This interface extends `JpaRepository`, which provides built-in methods for data access.

```
// Creating a repository interface extending JpaRepository
package com.customer.repository;

import com.customer.model.Customer;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
```

```
    Optional<Customer> findByEmail(String email);
}
```

**Note:** The **CustomerRepository** interface is annotated with **@Repository** annotation and extends the JpaRepository of [Spring Data JPA](#).

### Step 3: Creating Custom Exceptions

Now, we will create two custom exceptions:

**CustomerAlreadyExistsException:** This exception can be thrown when the user tries to add a customer that already exists in the database.

```
// Creating a custom exception that can be thrown when a user tries to add a customer that already exists
package com.customer.exception;

public class CustomerAlreadyExistsException extends RuntimeException {
    private String message;

    public CustomerAlreadyExistsException() {}

    public CustomerAlreadyExistsException(String msg) {
        super(msg);
        this.message = msg;
    }
}
```

**NoSuchCustomerExistsException:** This exception can be thrown when the user tries to delete or update a customer record that doesn't exist in the database.

```
// Creating a custom exception that can be thrown when a user tries to update/delete a customer that doesn't exist
package com.customer.exception;

public class NoSuchCustomerExistsException extends RuntimeException {
    private String message;

    public NoSuchCustomerExistsException() {}

    public NoSuchCustomerExistsException(String msg) {
        super(msg);
        this.message = msg;
    }
}
```

```
}
```

**Note:** Both Custom Exception classes extend *RuntimeException*.

## Step 4: Creating the Service Layer

The CustomerService interface defines three different methods:

- **Customer getCustomer(Long id):** To get a customer record by its id. This method throws a NoSuchElementException exception when it doesn't find a customer record with the given id.
- **String addCustomer(Customer customer):** To add details of a new Customer to the database. This method throws a CustomerAlreadyExistsException exception when the user tries to add a customer that already exists.
- **String updateCustomer(Customer customer):** To update details of Already existing Customers. This method throws a NoSuchCustomerExistsException exception when the user tries to update details of a customer that doesn't exist in the database.

The Interface and service implementation class is as follows:

### CustomerService Interface:

```
// Creating service interface
package com.customer.service;
import com.customer.model.Customer;

public interface CustomerService {

    // Method to get customer by its Id
    Customer getCustomer(Long id);

    // Method to add a new Customer
    // into the database
    String addCustomer(Customer customer);

    // Method to update details of a Customer
    String updateCustomer(Customer customer);
}
```

## CustomerServiceImpl Implementation:

```
// Implementing the service class
package com.customer.service;

import com.customer.exception.CustomerAlreadyExistsException;
import com.customer.exception.NoSuchCustomerExistsException;
import com.customer.model.Customer;
import com.customer.repository.CustomerRepository;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class CustomerServiceImpl implements CustomerService {

    @Autowired
    private CustomerRepository customerRespository;

    // Method to get customer by Id. Throws
    // NoSuchElementException for invalid Id
    public Customer getCustomer(Long id) {
        return customerRespository.findById(id).orElseThrow(
            () -> new NoSuchElementException("NO CUSTOMER PRESENT WITH ID = "
+ id));
    }

    // Simplifying the addCustomer and updateCustomer
    // methods with Optional for better readability.

    public String addCustomer(Customer customer) {
        Optional<Customer> existingCustomer =
customerRespository.findById(customer.getId());
        if (!existingCustomer.isPresent()) {
            customerRespository.save(customer);
            return "Customer added successfully";
        } else {
            throw new CustomerAlreadyExistsException("Customer already
exists!!");
        }
    }

    public String updateCustomer(Customer customer) {
        Optional<Customer> existingCustomer =
customerRespository.findById(customer.getId());
        if (!existingCustomer.isPresent()) {
            throw new NoSuchCustomerExistsException("No Such Customer
exists!!");
        } else {
            existingCustomer.get().setName(customer.getName());
            existingCustomer.get().setAddress(customer.getAddress());
            customerRespository.save(existingCustomer.get());
            return "Record updated Successfully";
        }
    }
}
```

```
    }  
}
```

## Step 5: Creating the CustomerController

The controller exposes RESTful endpoints for customer-related operations. The methods in this class will throw exceptions, which we will handle using various exception handling techniques.

```
// Creating Rest Controller CustomerController which  
// defines various API's.  
package com.customer.controller;  
  
import com.customer.exception.CustomerAlreadyExistsException;  
import com.customer.exception.ErrorResponse;  
import com.customer.exception.NoSuchCustomerExistsException;  
import com.customer.model.Customer;  
import com.customer.service.CustomerService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.HttpStatus;  
import org.springframework.web.bind.annotation.ExceptionHandler;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.PutMapping;  
import org.springframework.web.bind.annotation.RequestBody;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.ResponseStatus;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class CustomerController {  
  
    @Autowired private CustomerService customerService;  
  
    // Get Customer by Id  
    @GetMapping("/getCustomer/{id}")  
    public Customer getCustomer(@PathVariable("id") Long id) {  
        return customerService.getCustomer(id);  
    }  
  
    // Add new Customer  
    @PostMapping("/addCustomer")  
    public String addCustomer(@RequestBody Customer customer) {  
        return customerService.addCustomer(customer);  
    }  
  
    // Update Customer details
```

```
@PutMapping("/updateCustomer")
public String updateCustomer(@RequestBody Customer customer) {
    return customerService.updateCustomer(customer);
}

// Adding exception handlers for NoSuchCustomerExistsException
// and NoSuchElementException.
@ExceptionHandler(value = NoSuchCustomerExistsException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public ErrorResponse
handleNoSuchCustomerExistsException(NoSuchCustomerExistsException ex) {
    return new ErrorResponse(HttpStatus.NOT_FOUND.value(),
ex.getMessage());
}

@ExceptionHandler(value = NoSuchElementException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public ErrorResponse handleNoSuchElementException(NoSuchElementException ex) {
    return new ErrorResponse(HttpStatus.NOT_FOUND.value(),
ex.getMessage());
}
```

## Handling Exceptions in Spring Boot

Now let's go through the various ways in which we can handle the Exceptions thrown in this project.

### 1. Default Exception Handling by Spring Boot

The **getCustomer()** method defined by **CustomerController** is used to get a customer with a given Id. It throws a **NoSuchElementException** when it doesn't find a Customer record with the given id. On Running the Spring Boot Application and hitting the **/getCustomer** API with an Invalid Customer Id, we get a **NoSuchElementException** completely handled by Spring Boot as follows:

The screenshot shows a Postman interface with a GET request to `http://localhost:8080/getCustomer/2`. The response status is 500 Internal Server Error. The JSON body contains the following data:

```

1  "timestamp": "2022-03-01T18:24:31.643+00:00",
2  "status": 500,
3  "error": "Internal Server Error",
4  "message": "NO CUSTOMER PRESENT WITH ID = 2",
5  "path": "/getCustomer/2"

```

Spring Boot provides a systematic error response to the user with information such as timestamp, HTTP status code, error, message, and the path.

## 2. Using @ExceptionHandler Annotation

- **@ExceptionHandler** annotation provided by Spring Boot can be used to handle exceptions in particular **Handler classes or Handler methods**.
- Any method annotated with this is automatically recognized by Spring Configuration as an Exception Handler Method.
- An Exception Handler method handles all exceptions and their subclasses passed in the argument.
- It can also be configured to return a specific error response to the user.

So let's create a custom **ErrorResponse** class so that the exception is conveyed to the user in a clear and concise way as follows:

```
// Custom Error Response Class
package com.customer.exception;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
```

```
@NoArgsConstructor
public class ErrorResponse {

    private int statusCode;
    private String message;

    public ErrorResponse(String message)
    {
        super();
        this.message = message;
    }
}
```

The `addCustomer()` method defined by `CustomerController` throws a `CustomerAlreadyExistsException` when the user tries to add a Customer that already exists in the database else it saves the customer details.

To handle this exception let's define a handler method `handleCustomerAlreadyExistsException()` in the `CustomerController`. So, now when `addCustomer()` throws a `CustomerAlreadyExistsException`, the handler method gets invoked which returns a proper `ErrorResponse` to the user.

```
// Exception Handler method added in CustomerController to handle
CustomerAlreadyExistsException
@ExceptionHandler(value = CustomerAlreadyExistsException.class)
@ResponseBody(HttpStatus.CONFLICT)
public ErrorResponse
handleCustomerAlreadyExistsException(CustomerAlreadyExistsException ex) {
    return new ErrorResponse(HttpStatus.CONFLICT.value(), ex.getMessage());
}
```

*Note : Spring Boot allows to annotate a method with `@ResponseStatus` to return the required Http Status Code.*

On Running the Spring Boot Application and hitting the `/addCustomer` API with an existing Customer, `CustomerAlreadyExistsException` gets completely handled by handler method as follows:

The screenshot shows a POST request to `http://localhost:8080/addCustomer`. The request body is a JSON object with fields `id`, `name`, and `address`. The response status is `409 Conflict`, and the message is `"Customer already exists!!"`.

### 3. Using `@ControllerAdvice` for Global Exception Handling

In the previous approach, the `@ExceptionHandler` annotated method can only handle exceptions thrown by that particular class. If we want to handle any exception thrown throughout the application, we can define a global exception handler class and annotate it with `@ControllerAdvice`. This annotation helps to integrate multiple exception handlers into a single global unit.

The `updateCustomer()` method defined in `CustomerController` throws a `NoSuchCustomerExistsException` if the user tries to update details of a customer that doesn't already exist in the database. To handle this exception, define a `GlobalExceptionHandler` class annotated with `@ControllerAdvice`.

```
// Class to handle exceptions globally
package com.customer.exception;

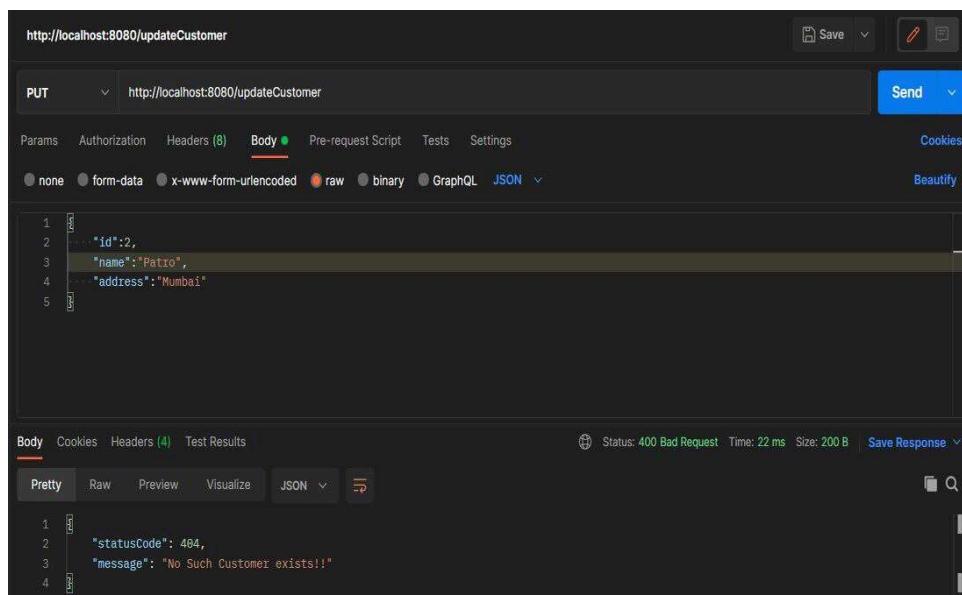
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
```

```
public class GlobalExceptionHandler {

    @ExceptionHandler(value = NoSuchCustomerExistsException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public @ResponseBody ErrorResponse
    handleException(NoSuchCustomerExistsException ex) {
        return new ErrorResponse(HttpStatus.NOT_FOUND.value(),
    ex.getMessage());
    }
}
```

On running the Spring Boot application and hitting the `/updateCustomer` API with invalid customer details, `NoSuchCustomerExistsException` gets thrown, which is completely handled by the handler method defined in the `GlobalExceptionHandler` class as follows:

[Comment](#)[More info](#)[Advertise with us](#)