Global Exception Handling & Validation in Spring Boot

Trainer name

Shreyansh Kumar

Introduction

Spring Boot has revolutionized Java application development by simplifying configuration and streamlining deployment processes. Among the most critical aspects of building robust applications are proper exception handling and validation mechanisms. Without these safeguards, applications may fail unexpectedly, display cryptic error messages, or expose sensitive system information through stack traces.

This comprehensive guide explores how to implement effective global exception handling and validation in Spring Boot applications. We'll progress from fundamental concepts to advanced implementations, featuring practical examples and visual diagrams that illustrate the complete exception handling flow.

The Critical Need for Global Exception Handling

In Spring Boot applications, exceptions can occur at any layer—controller, service, or repository. Without proper exception handling, these exceptions propagate to the client, typically resulting in a generic 500 Internal Server Error response with a detailed stack trace that potentially exposes sensitive information about your application's architecture.

A Common Scenario

Consider this situation: A user requests a resource that doesn't exist in your database. Without proper exception handling, your application might return:

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json

{
  "timestamp": "2023-10-15T10:15:30.123+00:00",
  "status": 500,
  "error": "Internal Server Error",
  "trace": "javax.persistence.EntityNotFoundException: Unable to find
com.example.entity.User with id 123\n at
com.example.repository.UserRepository.findById(UserRepository.java:45)\n at
com.example.service.UserService.getUserById(UserService.java:28)\n at
com.example.controller.UserController.getUser(UserController.java:32)\n ...",
  "message": "Unable to find com.example.entity.User with id 123",
  "path": "/api/users/123"
}
```

This response presents several significant issues:

- It exposes internal implementation details through the stack trace
- It uses an incorrect HTTP status code (500) for a "not found" scenario
- It provides a technical error message rather than a user-friendly one

With proper global exception handling, we can transform this response to:

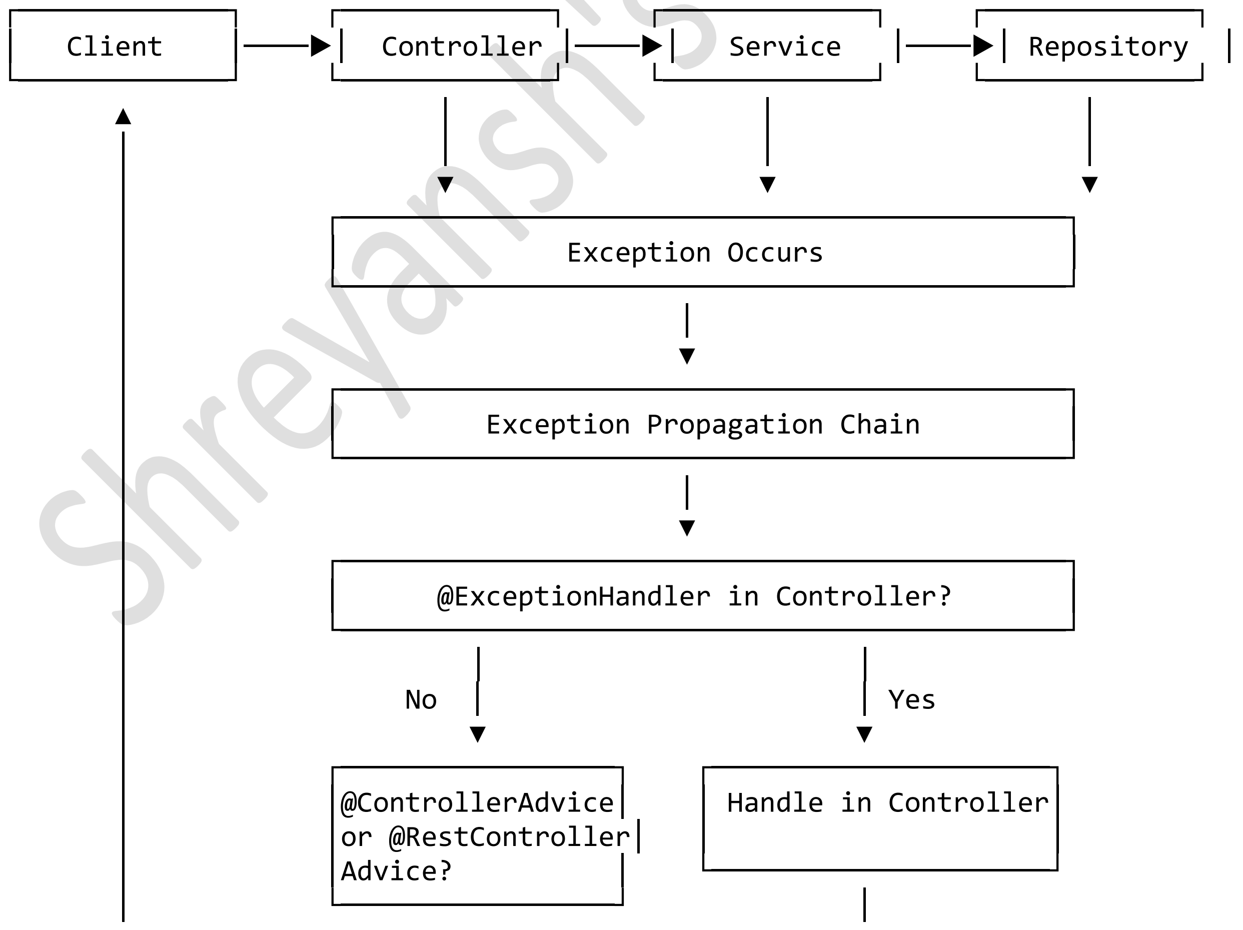
```
HTTP/1.1 404 Not Found
Content-Type: application/json

{
  "timestamp": "2023-10-15T10:15:30.123+00:00",
  "status": 404,
  "error": "Not Found",
  "message": "User with ID 123 not found",
  "path": "/api/users/123"
}
```

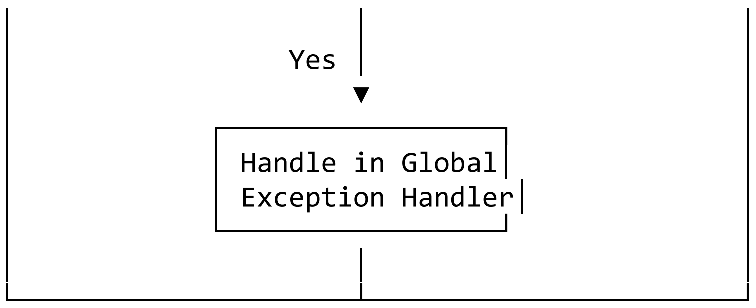
This improved response:

- Uses the semantically correct HTTP status code (404)
- Provides a clear, user-friendly message
- Conceals sensitive implementation details

Spring Boot Exception Handling Architecture

Before implementing a solution, let's understand how Spring Boot processes exceptions. The following diagram illustrates the request flow through a Spring Boot application and how exceptions are handled:


Client
Controller
Service
Repository
Exception Occurs
Exception Propagation Chain
@ExceptionHandler in Controller?
No
Yes
@ControllerAdvice or @RestController Advice?
Handle in Controller



When an exception occurs in any application layer, it propagates up the call stack. Spring Boot provides several mechanisms to catch and handle these exceptions:

1. **@ExceptionHandler** - Method-level annotation within controllers that handles exceptions thrown by specific controller methods
2. **@ControllerAdvice/@RestControllerAdvice** - Class-level annotations that apply @ExceptionHandler methods globally across all controllers
3. **HandlerExceptionResolver** - Interface that resolves exceptions thrown during controller execution

Implementing a Robust Global Exception Handling Solution

Let's build a comprehensive global exception handling solution for a Spring Boot application.

Step 1: Create a Hierarchy of Custom Exceptions

First, we'll define custom exception classes representing different error scenarios:

```
// Base exception class for our application
public class ApplicationException extends RuntimeException {
    public ApplicationException(String message) {
        super(message);
    }
&nbsp;
    public ApplicationException(String message, Throwable cause) {
        super(message, cause);
    }
}

// For resources that don't exist
public class ResourceNotFoundException extends ApplicationException {
    private final String resourceName;
    private final String fieldName;
    private final Object fieldValue;
&nbsp;
    public ResourceNotFoundException(String resourceName, String fieldName,
Object fieldValue) {
        super(String.format("%s not found with %s: '%s'", resourceName,
fieldName, fieldValue));
```

```java
        this.resourceName = resourceName;
        this.fieldName = fieldName;
        this.fieldValue = fieldValue;
    }
&nbsp;
    // Getters
}

// For invalid input data
public class BadRequestException extends ApplicationException {
    public BadRequestException(String message) {
        super(message);
    }
}

// For unauthorized access
public class UnauthorizedException extends ApplicationException {
    public UnauthorizedException(String message) {
        super(message);
    }
}

// For forbidden access (authenticated but not authorized)
public class ForbiddenException extends ApplicationException {
    public ForbiddenException(String message) {
        super(message);
    }
}
```

Step 2: Create a Standardized Error Response Model

Next, we'll create a consistent error response model:

```java
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.HashMap;

import com.fasterxml.jackson.annotation.JsonInclude;

@JsonInclude(JsonInclude.Include.NON_NULL)
public class ErrorResponse {
    private LocalDateTime timestamp;
    private int status;
    private String error;
    private String message;
    private String path;
    private List<ValidationError> validationErrors;
    private Map<String, Object> metadata;
```

```
&nbsp;
    // Constructors, getters, and setters
&nbsp;
    public ErrorResponse() {
        this.timestamp = LocalDateTime.now();
        this.validationErrors = new ArrayList<>();
        this.metadata = new HashMap<>();
    }
&nbsp;
    public ErrorResponse(int status, String error, String message, String
path) {
        this();
        this.status = status;
        this.error = error;
        this.message = message;
        this.path = path;
    }
&nbsp;
    // Inner class for validation errors
    public static class ValidationError {
        private String field;
        private String message;
        private Object rejectedValue;
&nbsp;
        // Constructors, getters, and setters
        public ValidationError(String field, String message) {
            this.field = field;
            this.message = message;
        }
&nbsp;
        public ValidationError(String field, String message, Object
rejectedValue) {
            this(field, message);
            this.rejectedValue = rejectedValue;
        }
&nbsp;
        // Getters and setters
    }
&nbsp;
    // Method to add validation errors
    public void addValidationError(String field, String message) {
        this.validationErrors.add(new ValidationError(field, message));
    }
&nbsp;
    public void addValidationError(String field, String message, Object
rejectedValue) {
        this.validationErrors.add(new ValidationError(field, message,
rejectedValue));
    }
&nbsp;
    // Add metadata
    public void addMetadata(String key, Object value) {
        this.metadata.put(key, value);
    }
}
```

Step 3: Create a Comprehensive Global Exception Handler

Now, let's implement a global exception handler using @RestControllerAdvice:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.MissingServletRequestParameterException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.context.request.ServletWebRequest;
import org.springframework.web.context.request.WebRequest;
import
org.springframework.web.method.annotation.MethodArgumentTypeMismatchException
;
import org.springframework.web.servlet.NoHandlerFoundException;

import javax.validation.ConstraintViolationException;
import java.util.Objects;

@RestControllerAdvice
public class GlobalExceptionHandler {
    private static final Logger logger =
LoggerFactory.getLogger(GlobalExceptionHandler.class);

    // Utility method to extract request path
    private String getRequestPath(WebRequest request) {
        if (request instanceof ServletWebRequest) {
            return ((ServletWebRequest)
request).getRequest().getRequestURI();
        }
        return request.getDescription(false);
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {
&nbsp;
        logger.error("Resource not found: {}", ex.getMessage());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.NOT_FOUND.value(),
                HttpStatus.NOT_FOUND.getReasonPhrase(),
                ex.getMessage(),
                getRequestPath(request)
        );
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }
&nbsp;
    @ExceptionHandler(BadRequestException.class)
    public ResponseEntity<ErrorResponse> handleBadRequestException(
            BadRequestException ex, WebRequest request) {
&nbsp;
        logger.error("Bad request: {}", ex.getMessage());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                ex.getMessage(),
                getRequestPath(request)
        );
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
&nbsp;
    @ExceptionHandler(UnauthorizedException.class)
    public ResponseEntity<ErrorResponse> handleUnauthorizedException(
            UnauthorizedException ex, WebRequest request) {
&nbsp;
        logger.error("Unauthorized access: {}", ex.getMessage());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.UNAUTHORIZED.value(),
                HttpStatus.UNAUTHORIZED.getReasonPhrase(),
                ex.getMessage(),
                getRequestPath(request)
        );
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.UNAUTHORIZED);
    }
&nbsp;
    @ExceptionHandler(ForbiddenException.class)
    public ResponseEntity<ErrorResponse> handleForbiddenException(
            ForbiddenException ex, WebRequest request) {
&nbsp;
        logger.error("Forbidden access: {}", ex.getMessage());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.FORBIDDEN.value(),
                HttpStatus.FORBIDDEN.getReasonPhrase(),
                ex.getMessage(),
                getRequestPath(request)
        );
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.FORBIDDEN);
    }
&nbsp;
    // Handle validation exceptions from @Valid
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse> handleValidationExceptions(
            MethodArgumentNotValidException ex, WebRequest request) {
&nbsp;
        logger.error("Validation error: {}", ex.getMessage());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                "Validation failed for request parameters",
                getRequestPath(request)
        );
&nbsp;
        // Add all field errors
        ex.getBindingResult().getFieldErrors().forEach(fieldError -> {
            errorResponse.addValidationError(
                    fieldError.getField(),
                    fieldError.getDefaultMessage(),
                    fieldError.getRejectedValue()
            );
        });
&nbsp;
        // Add global errors
        ex.getBindingResult().getGlobalErrors().forEach(objectError -> {
            errorResponse.addValidationError(
                    objectError.getObjectName(),
                    objectError.getDefaultMessage()
            );
        });
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
&nbsp;
    // Handle constraint violation exceptions (for @Valid on path variables
and request parameters)
    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<ErrorResponse> handleConstraintViolationException(
            ConstraintViolationException ex, WebRequest request) {
&nbsp;
        logger.error("Constraint violation: {}", ex.getMessage());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                "Validation error on request parameters",
                getRequestPath(request)
        );
&nbsp;
        ex.getConstraintViolations().forEach(violation -> {
            String fieldPath = violation.getPropertyPath().toString();
            // Extract field name (last part of the property path)
            String fieldName = fieldPath.substring(fieldPath.lastIndexOf('.')
+ 1);
&nbsp;
            errorResponse.addValidationError(
                    fieldName,
                    violation.getMessage(),
                    violation.getInvalidValue()
            );
        });
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
&nbsp;
    // Handle type mismatch exceptions
    @ExceptionHandler(MethodArgumentTypeMismatchException.class)
    public ResponseEntity<ErrorResponse> handleMethodArgumentTypeMismatch(
            MethodArgumentTypeMismatchException ex, WebRequest request) {
&nbsp;
        logger.error("Type mismatch: {}", ex.getMessage());
&nbsp;
        String errorMessage = String.format("The parameter '%s' should be of
type '%s'",&nbsp;
                ex.getName(),
Objects.requireNonNull(ex.getRequiredType()).getSimpleName());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                errorMessage,
                getRequestPath(request)
        );
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
&nbsp;
    // Handle missing request parameters
    @ExceptionHandler(MissingServletRequestParameterException.class)
    public ResponseEntity<ErrorResponse>
handleMissingServletRequestParameter(
            MissingServletRequestParameterException ex, WebRequest request) {
&nbsp;
        logger.error("Missing parameter: {}", ex.getMessage());
&nbsp;
        String errorMessage = String.format("The required parameter '%s' of
type '%s' is missing",&nbsp;
                ex.getParameterName(), ex.getParameterType());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                errorMessage,
                getRequestPath(request)
        );
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
&nbsp;
    // Handle malformed JSON
    @ExceptionHandler(HttpMessageNotReadableException.class)
    public ResponseEntity<ErrorResponse> handleHttpMessageNotReadable(
            HttpMessageNotReadableException ex, WebRequest request) {
&nbsp;
        logger.error("Malformed JSON request: {}", ex.getMessage());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                "Malformed JSON request",
                getRequestPath(request)
        );
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
&nbsp;
    // Handle 404 Not Found
    @ExceptionHandler(NoHandlerFoundException.class)
    public ResponseEntity<ErrorResponse> handleNoHandlerFoundException(
            NoHandlerFoundException ex, WebRequest request) {
&nbsp;
        logger.error("No handler found: {}", ex.getMessage());
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.NOT_FOUND.value(),
                HttpStatus.NOT_FOUND.getReasonPhrase(),
                String.format("The requested resource '%s' does not exist",
ex.getRequestURL()),
                getRequestPath(request)
        );
&nbsp;
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }
&nbsp;
    // Fallback handler for any unhandled exceptions
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGlobalException(
            Exception ex, WebRequest request) {
&nbsp;
        logger.error("Unhandled exception occurred", ex);
&nbsp;
        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase(),
                "An unexpected error occurred. Please contact support.",
                getRequestPath(request)
        );
&nbsp;
        // Add additional metadata for logging/debugging (not exposed to
client)
        errorResponse.addMetadata("exceptionClass", ex.getClass().getName());
&nbsp;
        return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

To enable the NoHandlerFoundException, add these properties to your application.properties:

```
spring.mvc.throw-exception-if-no-handler-found=true
spring.mvc.static-path-pattern=/static/**
```

Implementing Validation in Spring Boot

With our exception handling in place, let's implement comprehensive validation.

Step 1: Add Validation Dependencies

Ensure you have the necessary dependencies in your pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Step 2: Create a Model with Validation Constraints

Let's create a User model with validation constraints:

```
import javax.validation.constraints.*;
import java.time.LocalDate;

public class UserDto {
&nbsp;
    private Long id;
&nbsp;
    @NotBlank(message = "Name is required")
    @Size(min = 2, max = 50, message = "Name must be between 2 and 50 
characters")
    private String name;
&nbsp;
    @NotBlank(message = "Email is required")
    @Email(message = "Email must be valid")
    private String email;
&nbsp;
    @NotBlank(message = "Password is required")
    @Size(min = 8, message = "Password must be at least 8 characters")
    @Pattern(regexp = "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-
Z])(?=.*[@#$%^&+=]).*$",&nbsp;
             message = "Password must contain at least one digit, one 
lowercase, one uppercase, and one special character")
    private String password;
&nbsp;
    @Min(value = 18, message = "Age must be at least 18")
    @Max(value = 120, message = "Age must be less than 120")
    private int age;
&nbsp;
    @Past(message = "Birth date must be in the past")
    private LocalDate birthDate;
&nbsp;
    @Pattern(regexp = "^\\+[1-9]\\d{1,14}$", message = "Phone number must be 
in E.164 format")
    private String phoneNumber;
&nbsp;
    // Interdependent field validation can be done with custom validators
&nbsp;
    // Getters and setters
}
```

Step 3: Create a Controller with Validation

Now, let's create a controller that uses the @Valid annotation:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;
import java.util.List;

@RestController
@RequestMapping("/api/users")
@Validated  // For method parameter validation
public class UserController {
&nbsp;
```

```
    private final UserService userService;
&nbsp;
    public UserController(UserService userService) {
        this.userService = userService;
    }
&nbsp;
    @PostMapping
    public ResponseEntity<UserDto> createUser(@Valid @RequestBody UserDto
userDto) {
        UserDto createdUser = userService.createUser(userDto);
        return new ResponseEntity<>(createdUser, HttpStatus.CREATED);
    }
&nbsp;
    @GetMapping("/{id}")
    public ResponseEntity<UserDto> getUserById(
            @PathVariable @Min(value = 1, message = "ID must be positive")
Long id) {
        UserDto user = userService.getUserById(id);
        // If user not found, the service will throw
ResourceNotFoundException
        return ResponseEntity.ok(user);
    }
&nbsp;
    @GetMapping("/search")
    public ResponseEntity<List<UserDto>> searchUsers(
            @RequestParam @NotBlank(message = "Search term cannot be empty")
String query) {
        List<UserDto> users = userService.searchUsers(query);
        return ResponseEntity.ok(users);
    }
&nbsp;
    @PutMapping("/{id}")
    public ResponseEntity<UserDto> updateUser(
            @PathVariable @Min(value = 1, message = "ID must be positive")
Long id,
            @Valid @RequestBody UserDto userDto) {
&nbsp;
        UserDto updatedUser = userService.updateUser(id, userDto);
        return ResponseEntity.ok(updatedUser);
    }
&nbsp;
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(
            @PathVariable @Min(value = 1, message = "ID must be positive")
Long id) {
        userService.deleteUser(id);
        return ResponseEntity.noContent().build();
    }
}
```

Step 4: Implement Service Layer with Exception Throwing

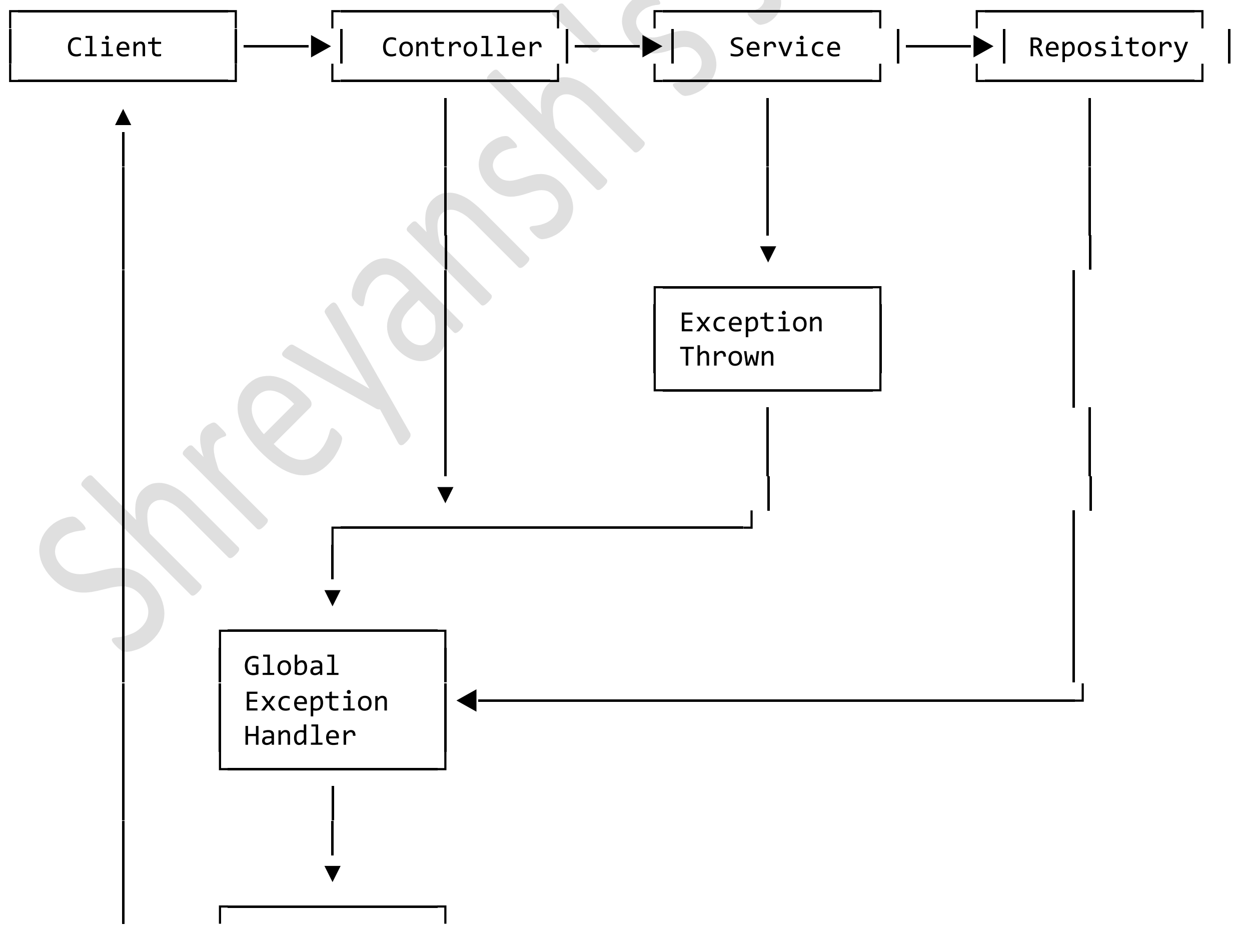
Let's implement a service layer that throws our custom exceptions:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;
import java.util.stream.Collectors;

@Service
public class UserService {
&nbsp;
    private final UserRepository userRepository;
&nbsp;
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
&nbsp;
    @Transactional
    public UserDto createUser(UserDto userDto) {
        // Check if email already exists
        if (userRepository.existsByEmail(userDto.getEmail())) {
            throw new BadRequestException("Email already in use: " +
userDto.getEmail());
        }
&nbsp;
        // Convert DTO to entity, save, and convert back to DTO
        User user = convertToEntity(userDto);
        User savedUser = userRepository.save(user);
        return convertToDto(savedUser);
    }
&nbsp;
    public UserDto getUserById(Long id) {
        User user = userRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("User",
"id", id));
&nbsp;
        return convertToDto(user);
    }
&nbsp;
    public List<UserDto> searchUsers(String query) {
        List<User> users =
userRepository.findByNameContainingIgnoreCase(query);
&nbsp;
        if (users.isEmpty()) {
            throw new ResourceNotFoundException("Users", "name containing",
query);
        }
&nbsp;
        return users.stream()
                .map(this::convertToDto)
                .collect(Collectors.toList());
    }
&nbsp;
    @Transactional
    public UserDto updateUser(Long id, UserDto userDto) {
        // Check if user exists
        User existingUser = userRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("User",
"id", id));
&nbsp;
        // Check if email is already in use by another user
        if (!existingUser.getEmail().equals(userDto.getEmail()) &&&nbsp;
                userRepository.existsByEmail(userDto.getEmail())) {
            throw new BadRequestException("Email already in use: " +
userDto.getEmail());
        }
&nbsp;
        // Update user properties
        existingUser.setName(userDto.getName());
        existingUser.setEmail(userDto.getEmail());
        existingUser.setAge(userDto.getAge());
        existingUser.setBirthDate(userDto.getBirthDate());
        existingUser.setPhoneNumber(userDto.getPhoneNumber());
&nbsp;
        // Save and return
        User updatedUser = userRepository.save(existingUser);
        return convertToDto(updatedUser);
    }
&nbsp;
    @Transactional
    public void deleteUser(Long id) {
        // Check if user exists
        if (!userRepository.existsById(id)) {
            throw new ResourceNotFoundException("User", "id", id);
        }
&nbsp;
        userRepository.deleteById(id);
    }
&nbsp;
    // Helper methods to convert between DTO and entity
    private User convertToEntity(UserDto userDto) {
        User user = new User();
        user.setName(userDto.getName());
        user.setEmail(userDto.getEmail());
        user.setPassword(encodePassword(userDto.getPassword())); // Assuming
password encoding
        user.setAge(userDto.getAge());
        user.setBirthDate(userDto.getBirthDate());
        user.setPhoneNumber(userDto.getPhoneNumber());
        return user;
    }
```

```
&nbsp;
    private UserDto convertToDto(User user) {
        UserDto userDto = new UserDto();
        userDto.setId(user.getId());
        userDto.setName(user.getName());
        userDto.setEmail(user.getEmail());
        userDto.setAge(user.getAge());
        userDto.setBirthDate(user.getBirthDate());
        userDto.setPhoneNumber(user.getPhoneNumber());
        // Don't set password in DTO for security reasons
        return userDto;
    }
&nbsp;
    private String encodePassword(String password) {
        // Password encoding logic (e.g., using BCryptPasswordEncoder)
        return password; // Placeholder - you'd use a password encoder here
    }
}
```

Exception Handling Flow Diagram

The complete flow of exception handling in our Spring Boot application:



```
      │     │ CreateGlobal Exception Handling & Validation in Spring Boot: A
Comprehensive Guide
```

Introduction

Spring Boot has revolutionized Java application development by simplifying configuration and streamlining deployment processes. Among the most critical aspects of building robust applications are proper exception handling and validation mechanisms. Without these safeguards, applications may fail unexpectedly, display cryptic error messages, or expose sensitive system information through stack traces.

This comprehensive guide explores how to implement effective global exception handling and validation in Spring Boot applications. We'll progress from fundamental concepts to advanced implementations, featuring practical examples and visual diagrams that illustrate the complete exception handling flow.

The Critical Need for Global Exception Handling

In Spring Boot applications, exceptions can occur at any layer—controller, service, or repository. Without proper exception handling, these exceptions propagate to the client, typically resulting in a generic 500 Internal Server Error response with a detailed stack trace that potentially exposes sensitive information about your application's architecture.

A Common Scenario

Consider this situation: A user requests a resource that doesn't exist in your database. Without proper exception handling, your application might return:

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json

{
  "timestamp": "2023-10-15T10:15:30.123+00:00",
  "status": 500,
  "error": "Internal Server Error",
  "trace": "javax.persistence.EntityNotFoundException: Unable to find
com.example.entity.User with id 123\n at
com.example.repository.UserRepository.findById(UserRepository.java:45)\n at
com.example.service.UserService.getUserById(UserService.java:28)\n at
com.example.controller.UserController.getUser(UserController.java:32)\n ...",
  "message": "Unable to find com.example.entity.User with id 123",
  "path": "/api/users/123"
}
```

This response presents several significant issues:

- It exposes internal implementation details through the stack trace
- It uses an incorrect HTTP status code (500) for a "not found" scenario
- It provides a technical error message rather than a user-friendly one

With proper global exception handling, we can transform this response to:

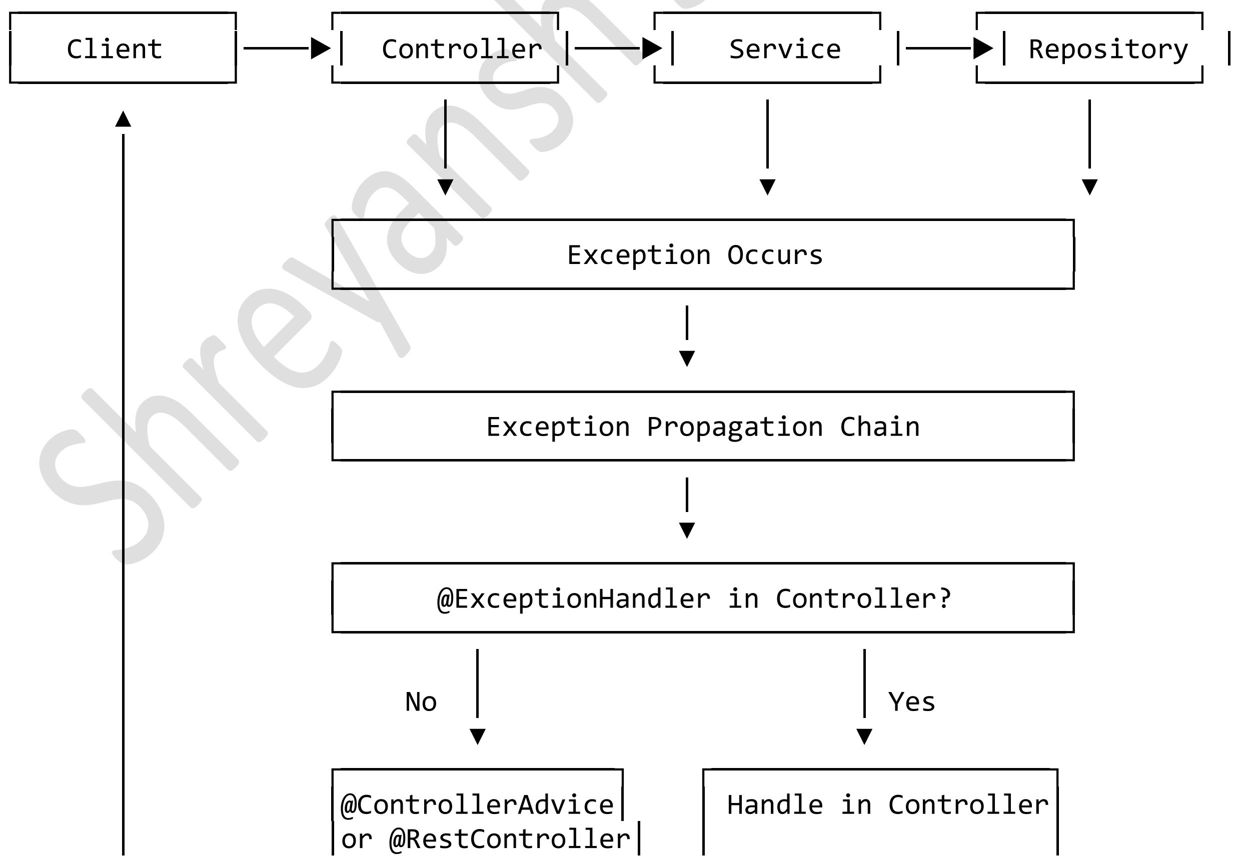
```
HTTP/1.1 404 Not Found
Content-Type: application/json

{
  "timestamp": "2023-10-15T10:15:30.123+00:00",
  "status": 404,
  "error": "Not Found",
  "message": "User with ID 123 not found",
  "path": "/api/users/123"
}
```

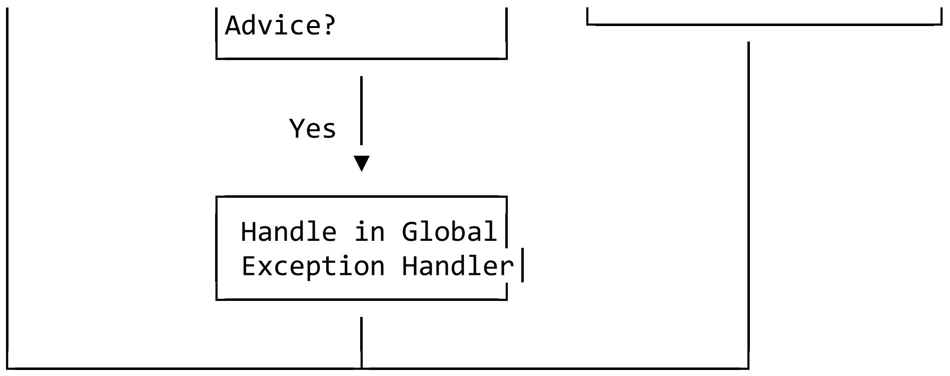
This improved response:

- Uses the semantically correct HTTP status code (404)
- Provides a clear, user-friendly message
- Conceals sensitive implementation details

Spring Boot Exception Handling Architecture

Before implementing a solution, let's understand how Spring Boot processes exceptions. The following diagram illustrates the request flow through a Spring Boot application and how exceptions are handled:


Client
Controller
Service
Repository
Exception Occurs
Exception Propagation Chain
@ExceptionHandler in Controller?
No
Yes
@ControllerAdvice or @RestController
Handle in Controller



When an exception occurs in any application layer, it propagates up the call stack. Spring Boot provides several mechanisms to catch and handle these exceptions:

1. **@ExceptionHandler** - Method-level annotation within controllers that handles exceptions thrown by specific controller methods
2. **@ControllerAdvice/@RestControllerAdvice** - Class-level annotations that apply @ExceptionHandler methods globally across all controllers
3. **HandlerExceptionResolver** - Interface that resolves exceptions thrown during controller execution

Implementing a Robust Global Exception Handling Solution

Let's build a comprehensive global exception handling solution for a Spring Boot application.

Step 1: Create a Hierarchy of Custom Exceptions

First, we'll define custom exception classes representing different error scenarios:

```java
// Base exception class for our application
public class ApplicationException extends RuntimeException {
    public ApplicationException(String message) {
        super(message);
    }

    public ApplicationException(String message, Throwable cause) {
        super(message, cause);
    }
}

// For resources that don't exist
public class ResourceNotFoundException extends ApplicationException {
    private final String resourceName;
    private final String fieldName;
    private final Object fieldValue;

    public ResourceNotFoundException(String resourceName, String fieldName,
Object fieldValue) {
        super(String.format("%s not found with %s: '%s'", resourceName,
fieldName, fieldValue));
        this.resourceName = resourceName;
        this.fieldName = fieldName;
        this.fieldValue = fieldValue;
    }

    // Getters
}

// For invalid input data
public class BadRequestException extends ApplicationException {
    public BadRequestException(String message) {
        super(message);
    }
}

// For unauthorized access
public class UnauthorizedException extends ApplicationException {
    public UnauthorizedException(String message) {
        super(message);
    }
}

// For forbidden access (authenticated but not authorized)
public class ForbiddenException extends ApplicationException {
    public ForbiddenException(String message) {
        super(message);
    }
}
```

Step 2: Create a Standardized Error Response Model

Next, we'll create a consistent error response model:

```java
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.HashMap;

import com.fasterxml.jackson.annotation.JsonInclude;

@JsonInclude(JsonInclude.Include.NON_NULL)
public class ErrorResponse {
    private LocalDateTime timestamp;
    private int status;
    private String error;
    private String message;
    private String path;
    private List<ValidationError> validationErrors;
    private Map<String, Object> metadata;

    // Constructors, getters, and setters

    public ErrorResponse() {
        this.timestamp = LocalDateTime.now();
        this.validationErrors = new ArrayList<>();
        this.metadata = new HashMap<>();
    }

    public ErrorResponse(int status, String error, String message, String
path) {
        this();
        this.status = status;
        this.error = error;
        this.message = message;
        this.path = path;
    }

    // Inner class for validation errors
    public static class ValidationError {
        private String field;
        private String message;
        private Object rejectedValue;

        // Constructors, getters, and setters
        public ValidationError(String field, String message) {
            this.field = field;
            this.message = message;
        }

        public ValidationError(String field, String message, Object
rejectedValue) {
            this(field, message);
            this.rejectedValue = rejectedValue;
        }

        // Getters and setters
    }

    // Method to add validation errors
    public void addValidationError(String field, String message) {
        this.validationErrors.add(new ValidationError(field, message));
    }

    public void addValidationError(String field, String message, Object
rejectedValue) {
        this.validationErrors.add(new ValidationError(field, message,
```

```java
rejectedValue));
    }

    // Add metadata
    public void addMetadata(String key, Object value) {
        this.metadata.put(key, value);
    }
}
```

Step 3: Create a Comprehensive Global Exception Handler

Now, let's implement a global exception handler using @RestControllerAdvice:

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.http.converter.HttpMessageNotReadableException;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.MissingServletRequestParameterException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.context.request.ServletWebRequest;
import org.springframework.web.context.request.WebRequest;
import
org.springframework.web.method.annotation.MethodArgumentTypeMismatchException
;
import org.springframework.web.servlet.NoHandlerFoundException;

import javax.validation.ConstraintViolationException;
import java.util.Objects;

@RestControllerAdvice
public class GlobalExceptionHandler {
    private static final Logger logger =
LoggerFactory.getLogger(GlobalExceptionHandler.class);

    // Utility method to extract request path
    private String getRequestPath(WebRequest request) {
        if (request instanceof ServletWebRequest) {
            return ((ServletWebRequest)
request).getRequest().getRequestURI();
        }
        return request.getDescription(false);
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {
```

```
        logger.error("Resource not found: {}", ex.getMessage());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.NOT_FOUND.value(),
                HttpStatus.NOT_FOUND.getReasonPhrase(),
                ex.getMessage(),
                getRequestPath(request)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(BadRequestException.class)
    public ResponseEntity<ErrorResponse> handleBadRequestException(
            BadRequestException ex, WebRequest request) {

        logger.error("Bad request: {}", ex.getMessage());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                ex.getMessage(),
                getRequestPath(request)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(UnauthorizedException.class)
    public ResponseEntity<ErrorResponse> handleUnauthorizedException(
            UnauthorizedException ex, WebRequest request) {

        logger.error("Unauthorized access: {}", ex.getMessage());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.UNAUTHORIZED.value(),
                HttpStatus.UNAUTHORIZED.getReasonPhrase(),
                ex.getMessage(),
                getRequestPath(request)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.UNAUTHORIZED);
    }

    @ExceptionHandler(ForbiddenException.class)
    public ResponseEntity<ErrorResponse> handleForbiddenException(
            ForbiddenException ex, WebRequest request) {
```

```
        logger.error("Forbidden access: {}", ex.getMessage());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.FORBIDDEN.value(),
                HttpStatus.FORBIDDEN.getReasonPhrase(),
                ex.getMessage(),
                getRequestPath(request)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.FORBIDDEN);
    }

    // Handle validation exceptions from @Valid
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse> handleValidationExceptions(
            MethodArgumentNotValidException ex, WebRequest request) {

        logger.error("Validation error: {}", ex.getMessage());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                "Validation failed for request parameters",
                getRequestPath(request)
        );

        // Add all field errors
        ex.getBindingResult().getFieldErrors().forEach(fieldError -> {
            errorResponse.addValidationError(
                    fieldError.getField(),
                    fieldError.getDefaultMessage(),
                    fieldError.getRejectedValue()
            );
        });

        // Add global errors
        ex.getBindingResult().getGlobalErrors().forEach(objectError -> {
            errorResponse.addValidationError(
                    objectError.getObjectName(),
                    objectError.getDefaultMessage()
            );
        });

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }

    // Handle constraint violation exceptions (for @Valid on path variables
and request parameters)
    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<ErrorResponse> handleConstraintViolationException(
            ConstraintViolationException ex, WebRequest request) {

        logger.error("Constraint violation: {}", ex.getMessage());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                "Validation error on request parameters",
                getRequestPath(request)
        );

        ex.getConstraintViolations().forEach(violation -> {
            String fieldPath = violation.getPropertyPath().toString();
            // Extract field name (last part of the property path)
            String fieldName = fieldPath.substring(fieldPath.lastIndexOf('.')
+ 1);

            errorResponse.addValidationError(
                    fieldName,
                    violation.getMessage(),
                    violation.getInvalidValue()
            );
        });

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }

    // Handle type mismatch exceptions
    @ExceptionHandler(MethodArgumentTypeMismatchException.class)
    public ResponseEntity<ErrorResponse> handleMethodArgumentTypeMismatch(
            MethodArgumentTypeMismatchException ex, WebRequest request) {

        logger.error("Type mismatch: {}", ex.getMessage());

        String errorMessage = String.format("The parameter '%s' should be of
type '%s'",
                ex.getName(),
Objects.requireNonNull(ex.getRequiredType()).getSimpleName());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                errorMessage,
                getRequestPath(request)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }
    // Handle missing request parameters
    @ExceptionHandler(MissingServletRequestParameterException.class)
    public ResponseEntity<ErrorResponse> 
handleMissingServletRequestParameter(
            MissingServletRequestParameterException ex, WebRequest request) {

        logger.error("Missing parameter: {}", ex.getMessage());

        String errorMessage = String.format("The required parameter '%s' of 
type '%s' is missing",
                ex.getParameterName(), ex.getParameterType());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                errorMessage,
                getRequestPath(request)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }

    // Handle malformed JSON
    @ExceptionHandler(HttpMessageNotReadableException.class)
    public ResponseEntity<ErrorResponse> handleHttpMessageNotReadable(
            HttpMessageNotReadableException ex, WebRequest request) {

        logger.error("Malformed JSON request: {}", ex.getMessage());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                "Malformed JSON request",
                getRequestPath(request)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }

    // Handle 404 Not Found
    @ExceptionHandler(NoHandlerFoundException.class)
    public ResponseEntity<ErrorResponse> handleNoHandlerFoundException(
            NoHandlerFoundException ex, WebRequest request) {

        logger.error("No handler found: {}", ex.getMessage());

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.NOT_FOUND.value(),
                HttpStatus.NOT_FOUND.getReasonPhrase(),
                String.format("The requested resource '%s' does not exist",
ex.getRequestURL()),
                getRequestPath(request)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    // Fallback handler for any unhandled exceptions
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGlobalException(
            Exception ex, WebRequest request) {

        logger.error("Unhandled exception occurred", ex);

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase(),
                "An unexpected error occurred. Please contact support.",
                getRequestPath(request)
        );

        // Add additional metadata for logging/debugging (not exposed to
client)
        errorResponse.addMetadata("exceptionClass", ex.getClass().getName());

        return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

To enable the NoHandlerFoundException, add these properties to your application.properties:

```
spring.mvc.throw-exception-if-no-handler-found=true
spring.mvc.static-path-pattern=/static/**
```

Implementing Validation in Spring Boot

With our exception handling in place, let's implement comprehensive validation.

Step 1: Add Validation Dependencies

Ensure you have the necessary dependencies in your pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Step 2: Create a Model with Validation Constraints

Let's create a User model with validation constraints:

```
import javax.validation.constraints.*;
import java.time.LocalDate;

public class UserDto {

    private Long id;

    @NotBlank(message = "Name is required")
    @Size(min = 2, max = 50, message = "Name must be between 2 and 50 
characters")
    private String name;

    @NotBlank(message = "Email is required")
    @Email(message = "Email must be valid")
    private String email;

    @NotBlank(message = "Password is required")
    @Size(min = 8, message = "Password must be at least 8 characters")
    @Pattern(regexp = "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-
Z])(?=.*[@#$%^&+=]).*$",
             message = "Password must contain at least one digit, one 
lowercase, one uppercase, and one special character")
    private String password;

    @Min(value = 18, message = "Age must be at least 18")
    @Max(value = 120, message = "Age must be less than 120")
    private int age;

    @Past(message = "Birth date must be in the past")
    private LocalDate birthDate;

    @Pattern(regexp = "^\\+[1-9]\\d{1,14}$", message = "Phone number must be 
in E.164 format")
    private String phoneNumber;

    // Interdependent field validation can be done with custom validators

    // Getters and setters
}
```

Step 3: Create a Controller with Validation

Now, let's create a controller that uses the @Valid annotation:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
```

```java
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;
import java.util.List;

@RestController
@RequestMapping("/api/users")
@Validated  // For method parameter validation
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping
    public ResponseEntity<UserDto> createUser(@Valid @RequestBody UserDto
userDto) {
        UserDto createdUser = userService.createUser(userDto);
        return new ResponseEntity<>(createdUser, HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
    public ResponseEntity<UserDto> getUserById(
            @PathVariable @Min(value = 1, message = "ID must be positive")
Long id) {
        UserDto user = userService.getUserById(id);
        // If user not found, the service will throw
ResourceNotFoundException
        return ResponseEntity.ok(user);
    }

    @GetMapping("/search")
    public ResponseEntity<List<UserDto>> searchUsers(
            @RequestParam @NotBlank(message = "Search term cannot be empty")
String query) {
        List<UserDto> users = userService.searchUsers(query);
        return ResponseEntity.ok(users);
    }

    @PutMapping("/{id}")
    public ResponseEntity<UserDto> updateUser(
            @PathVariable @Min(value = 1, message = "ID must be positive")
Long id,
            @Valid @RequestBody UserDto userDto) {
```

```
        UserDto updatedUser = userService.updateUser(id, userDto);
        return ResponseEntity.ok(updatedUser);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(
            @PathVariable @Min(value = 1, message = "ID must be positive")
Long id) {
        userService.deleteUser(id);
        return ResponseEntity.noContent().build();
    }
}
```

Step 4: Implement Service Layer with Exception Throwing

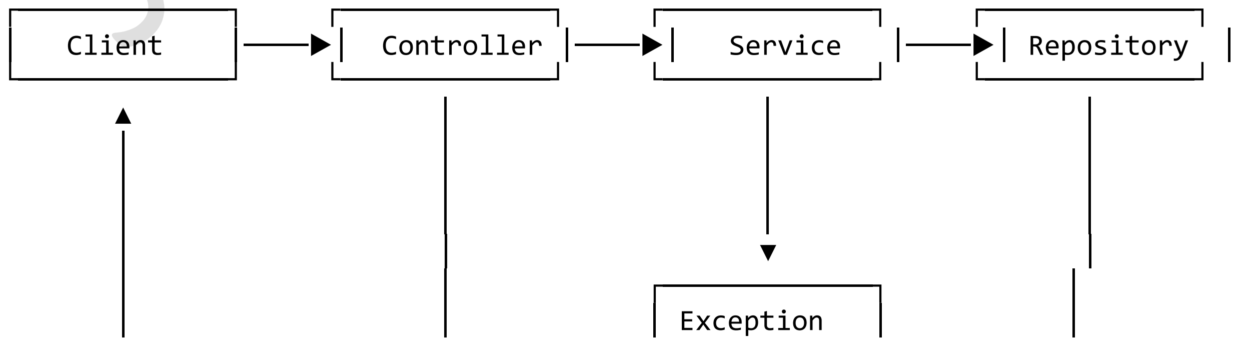
Let's implement a service layer that throws our custom exceptions:

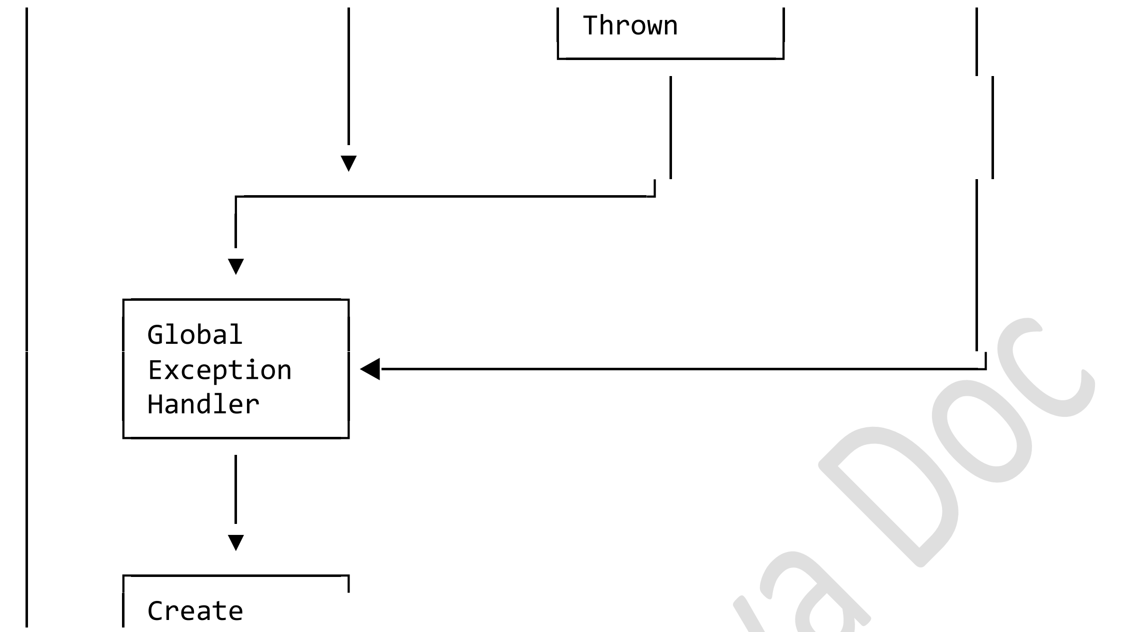
```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;
import java.util.stream.Collectors;

@Service
public class UserService {

    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Transactional
    public UserDto createUser(UserDto userDto) {
        // Check if email already exists
        if (userRepository.existsByEmail(userDto.getEmail())) {
            throw new BadRequestException("Email already in use: " +
userDto.getEmail());
        }

        // Convert DTO to entity, save, and convert back to DTO
        User user = convertToEntity(userDto);
        User savedUser = userRepository.save(user);
        return convertToDto(savedUser);
    }

    public UserDto getUserById(Long id) {
        User user = userRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("User",
"id", id));
```

```
        return convertToDto(user);
    }

    public List<UserDto> searchUsers(String query) {
        List<User> users =
userRepository.findByNameContainingIgnoreCase(query);

        if (users.isEmpty()) {
            throw new ResourceNotFoundException("Users", "name containing",
query);
        }

        return users.stream()
                .map(this::convertToDto)
                .collect(Collectors.toList());
    }

    @Transactional
    public UserDto updateUser(Long id, UserDto userDto) {
        // Check if user exists
        User existingUser = userRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("User",
"id", id));

        // Check if email is already in use by another user
        if (!existingUser.getEmail().equals(userDto.getEmail()) &&
                userRepository.existsByEmail(userDto.getEmail())) {
            throw new BadRequestException("Email already in use: " +
userDto.getEmail());
        }

        // Update user properties
        existingUser.setName(userDto.getName());
        existingUser.setEmail(userDto.getEmail());
        existingUser.setAge(userDto.getAge());
        existingUser.setBirthDate(userDto.getBirthDate());
        existingUser.setPhoneNumber(userDto.getPhoneNumber());

        // Save and return
        User updatedUser = userRepository.save(existingUser);
        return convertToDto(updatedUser);
    }

    @Transactional
    public void deleteUser(Long id) {
        // Check if user exists
        if (!userRepository.existsById(id)) {
            throw new ResourceNotFoundException("User", "id", id);
        }

        userRepository.deleteById(id);
    }

    // Helper methods to convert between DTO and entity
    private User convertToEntity(UserDto userDto) {
        User user = new User();
        user.setName(userDto.getName());
        user.setEmail(userDto.getEmail());
        user.setPassword(encodePassword(userDto.getPassword())); // Assuming
password encoding
        user.setAge(userDto.getAge());
        user.setBirthDate(userDto.getBirthDate());
        user.setPhoneNumber(userDto.getPhoneNumber());
        return user;
    }

    private UserDto convertToDto(User user) {
        UserDto userDto = new UserDto();
        userDto.setId(user.getId());
        userDto.setName(user.getName());
        userDto.setEmail(user.getEmail());
        userDto.setAge(user.getAge());
        userDto.setBirthDate(user.getBirthDate());
        userDto.setPhoneNumber(user.getPhoneNumber());
        // Don't set password in DTO for security reasons
        return userDto;
    }

    private String encodePassword(String password) {
        // Password encoding logic (e.g., using BCryptPasswordEncoder)
        return password; // Placeholder - you'd use a password encoder here
    }
}
```

Exception Handling Flow Diagram

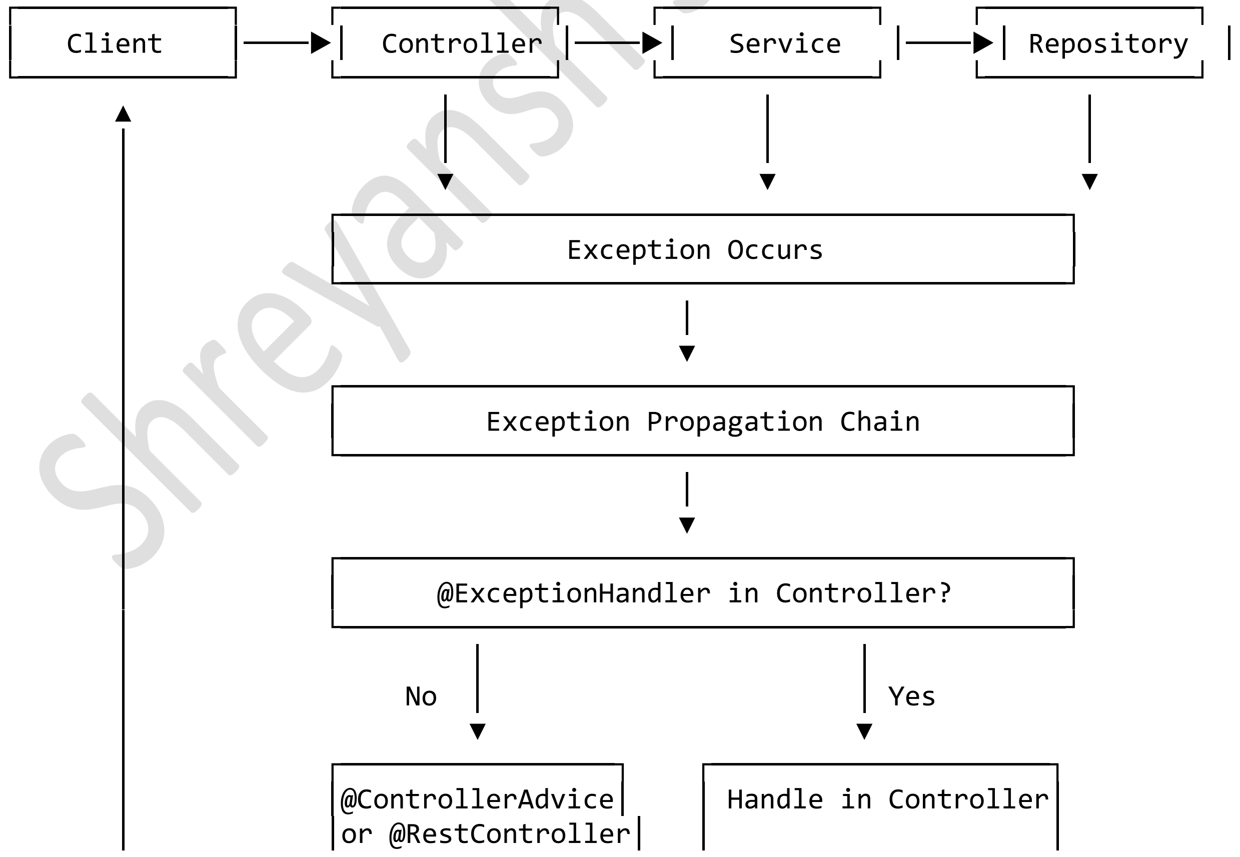
The complete flow of exception handling in our Spring Boot application:

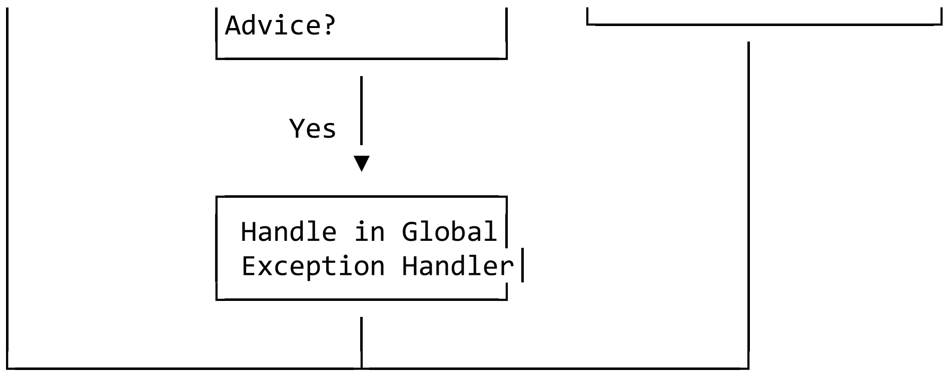




Spring Boot Exception Handling Architecture

Before diving into implementation, let's understand how Spring Boot handles exceptions. The following diagram illustrates the flow of a request through a Spring Boot application and how exceptions are handled:


Client
Controller
Service
Repository
Exception Occurs
Exception Propagation Chain
@ExceptionHandler in Controller?
No
Yes
@ControllerAdvice or @RestController
Handle in Controller



This diagram shows that when an exception occurs in any layer of your application, it propagates up the call stack. Spring Boot provides several mechanisms to catch and handle these exceptions:

1. **@ExceptionHandler** - Method-level annotation within controllers to handle exceptions thrown by specific controller methods
2. **@ControllerAdvice/@RestControllerAdvice** - Class-level annotations that apply @ExceptionHandler methods globally across all controllers
3. **HandlerExceptionResolver** - Interface that resolves exceptions thrown during controller execution

Implementing Global Exception Handling

Now, let's implement a comprehensive global exception handling solution for a Spring Boot application.

Step 1: Create Custom Exception Classes

First, let's define some custom exception classes that represent different error scenarios in our application:

```java
// Base exception class for our application
public class ApplicationException extends RuntimeException {
    public ApplicationException(String message) {
        super(message);
    }

    public ApplicationException(String message, Throwable cause) {
        super(message, cause);
    }
}

// For resources that don't exist
public class ResourceNotFoundException extends ApplicationException {
    public ResourceNotFoundException(String resourceName, String fieldName,
Object fieldValue) {
        super(String.format("%s not found with %s: '%s'", resourceName,
fieldName, fieldValue));
    }
}

// For invalid input data
public class BadRequestException extends ApplicationException {
    public BadRequestException(String message) {
        super(message);
    }
}

// For unauthorized access
public class UnauthorizedException extends ApplicationException {
    public UnauthorizedException(String message) {
        super(message);
    }
}
```

Step 2: Create Error Response Model

Next, let's create a standardized error response model that will be returned to clients:

```
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

public class ErrorResponse {
    private LocalDateTime timestamp;
    private int status;
    private String error;
    private String message;
    private String path;
    private List<ValidationError> validationErrors;

    // Constructors, getters, and setters

    public ErrorResponse() {
        this.timestamp = LocalDateTime.now();
        this.validationErrors = new ArrayList<>();
    }

    public ErrorResponse(int status, String error, String message, String
path) {
        this();
        this.status = status;
        this.error = error;
        this.message = message;
        this.path = path;
    }
    // Inner class for validation errors
    public static class ValidationError {
        private String field;
        private String message;

        // Constructors, getters, and setters
    }

    // Method to add validation errors
    public void addValidationError(String field, String message) {
        ValidationError validationError = new ValidationError();
        validationError.setField(field);
        validationError.setMessage(message);
        this.validationErrors.add(validationError);
    }
}
```

Step 3: Create Global Exception Handler

Now, let's create a global exception handler using @RestControllerAdvice:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.context.request.WebRequest;

import javax.validation.ConstraintViolationException;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.NOT_FOUND.value(),
                HttpStatus.NOT_FOUND.getReasonPhrase(),
                ex.getMessage(),
                request.getDescription(false)
        );

        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(BadRequestException.class)
    public ResponseEntity<ErrorResponse> handleBadRequestException(
            BadRequestException ex, WebRequest request) {
```

```java
    ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.BAD_REQUEST.value(),
            HttpStatus.BAD_REQUEST.getReasonPhrase(),
            ex.getMessage(),
            request.getDescription(false)
    );

    return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
}

@ExceptionHandler(UnauthorizedException.class)
public ResponseEntity<ErrorResponse> handleUnauthorizedException(
        UnauthorizedException ex, WebRequest request) {

    ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.UNAUTHORIZED.value(),
            HttpStatus.UNAUTHORIZED.getReasonPhrase(),
            ex.getMessage(),
            request.getDescription(false)
    );

    return new ResponseEntity<>(errorResponse, HttpStatus.UNAUTHORIZED);
}

// Handle validation exceptions
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ErrorResponse> handleValidationExceptions(
        MethodArgumentNotValidException ex, WebRequest request) {

    ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.BAD_REQUEST.value(),
            HttpStatus.BAD_REQUEST.getReasonPhrase(),
            "Validation error",
            request.getDescription(false)
    );

    // Add all field errors
    ex.getBindingResult().getFieldErrors().forEach(fieldError -> {
        errorResponse.addValidationError(
                fieldError.getField(),
                fieldError.getDefaultMessage()
        );
    });

    return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
}

// Handle constraint violation exceptions (for @Valid on path variables
and request parameters)
    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<ErrorResponse> handleConstraintViolationException(
            ConstraintViolationException ex, WebRequest request) {

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.BAD_REQUEST.value(),
                HttpStatus.BAD_REQUEST.getReasonPhrase(),
                "Validation error",
                request.getDescription(false)
        );

        ex.getConstraintViolations().forEach(violation -> {
            String fieldName = violation.getPropertyPath().toString();
            errorResponse.addValidationError(
                    fieldName,
                    violation.getMessage()
            );
        });

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }

    // Fallback handler for any unhandled exceptions
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGlobalException(
            Exception ex, WebRequest request) {

        ErrorResponse errorResponse = new ErrorResponse(
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase(),
                "An unexpected error occurred",
                request.getDescription(false)
        );

        return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

This global exception handler:

- Handles specific custom exceptions with appropriate HTTP status codes
- Processes validation errors from @Valid annotations
- Provides a fallback handler for any unhandled exceptions

Implementing Validation in Spring Boot

Now that we have our exception handling in place, let's implement validation using Spring Boot's validation framework.

Step 1: Add Validation Dependencies

First, ensure you have the necessary dependencies in your pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Step 2: Create a Model with Validation Constraints

Let's create a User model with validation constraints:

```
import javax.validation.constraints.*;

public class UserDto {

    private Long id;

    @NotBlank(message = "Name is required")
    @Size(min = 2, max = 50, message = "Name must be between 2 and 50 
characters")
    private String name;

    @NotBlank(message = "Email is required")
    @Email(message = "Email should be valid")
    private String email;

    @NotBlank(message = "Password is required")
    @Size(min = 8, message = "Password must be at least 8 characters")
    @Pattern(regexp = "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-
Z])(?=.*[@#$%^&+=]).*$",
             message = "Password must contain at least one digit, one 
lowercase, one uppercase, and one special character")
    private String password;

    @Min(value = 18, message = "Age must be at least 18")
    @Max(value = 120, message = "Age must be less than 120")
    private int age;

    // Getters and setters
}
```

Step 3: Create a Controller with Validation

Now, let's create a controller that uses the @Valid annotation to validate incoming requests:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import javax.validation.constraints.Min;

@RestController
@RequestMapping("/api/users")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping
    public ResponseEntity<UserDto> createUser(@Valid @RequestBody UserDto
userDto) {
        UserDto createdUser = userService.createUser(userDto);
        return new ResponseEntity<>(createdUser, HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
    public ResponseEntity<UserDto> getUserById(@PathVariable Long id) {
        UserDto user = userService.getUserById(id);
        // If user not found, the service will throw
ResourceNotFoundException
        return ResponseEntity.ok(user);
    }

    @PutMapping("/{id}")
    public ResponseEntity<UserDto> updateUser(
            @PathVariable Long id,
            @Valid @RequestBody UserDto userDto) {

        UserDto updatedUser = userService.updateUser(id, userDto);
        return ResponseEntity.ok(updatedUser);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
        return ResponseEntity.noContent().build();
    }
}
```

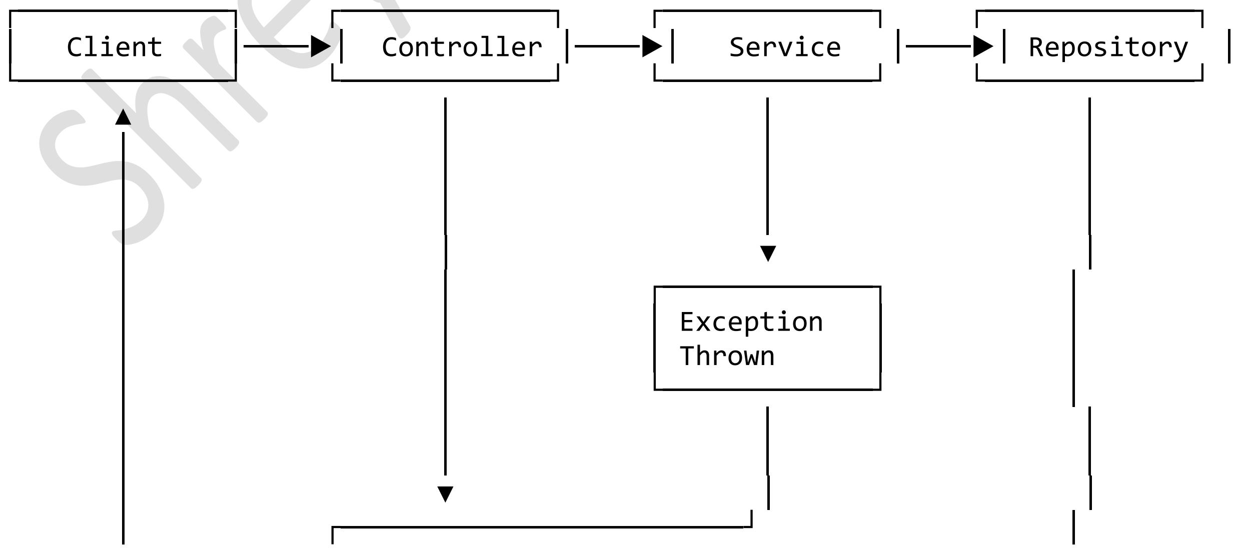
Step 4: Implement Service Layer with Exception Throwing

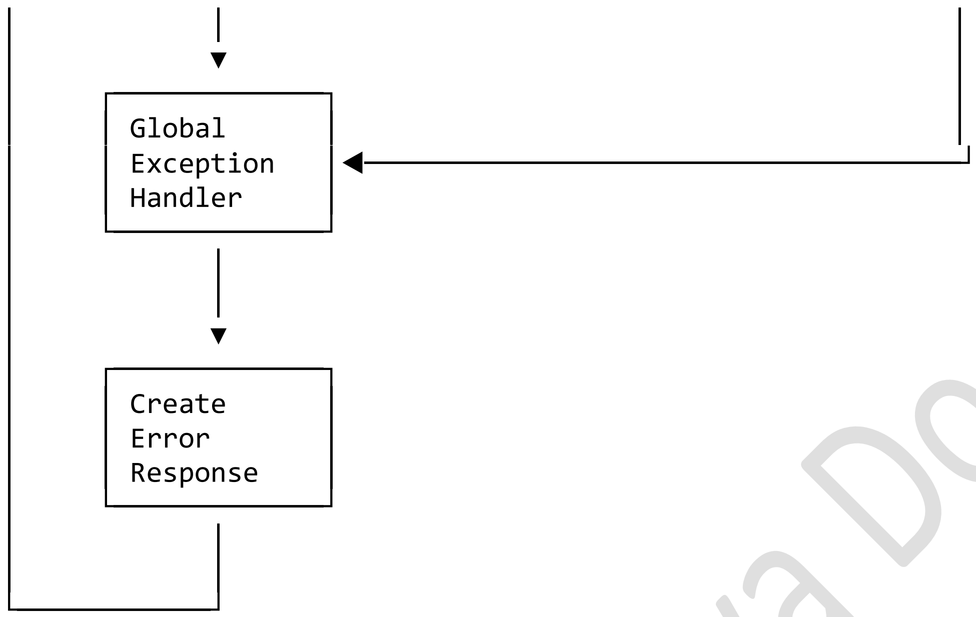
Let's implement a service layer that throws our custom exceptions:

```
import org.springframework.stereotype.Service;

@Service
public class UserService {

    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public UserDto createUser(UserDto userDto) {
        // Check if email already exists
        if (userRepository.existsByEmail(userDto.getEmail())) {
            throw new BadRequestException("Email already in use");
        }

        // Convert DTO to entity, save, and convert back to DTO
        User user = convertToEntity(userDto);
        User savedUser = userRepository.save(user);
        return convertToDto(savedUser);
    }

    public UserDto getUserById(Long id) {
        User user = userRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("User",
"id", id));

        return convertToDto(user);
    }

    public UserDto updateUser(Long id, UserDto userDto) {
        // Check if user exists
        User existingUser = userRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("User",
"id", id));

        // Check if email is already in use by another user
        if (!existingUser.getEmail().equals(userDto.getEmail()) &&
                userRepository.existsByEmail(userDto.getEmail())) {
            throw new BadRequestException("Email already in use");
        }
        // Update user properties
        existingUser.setName(userDto.getName());
        existingUser.setEmail(userDto.getEmail());
        existingUser.setAge(userDto.getAge());

        // Save and return
        User updatedUser = userRepository.save(existingUser);
        return convertToDto(updatedUser);
    }

    public void deleteUser(Long id) {
        // Check if user exists
        if (!userRepository.existsById(id)) {
            throw new ResourceNotFoundException("User", "id", id);
        }

        userRepository.deleteById(id);
    }

    // Helper methods to convert between DTO and entity
    private User convertToEntity(UserDto userDto) {
        // Implementation
    }

    private UserDto convertToDto(User user) {
        // Implementation
    }
}
```

Exception Handling Flow Diagram

Let's visualize the complete flow of exception handling in our Spring Boot application:



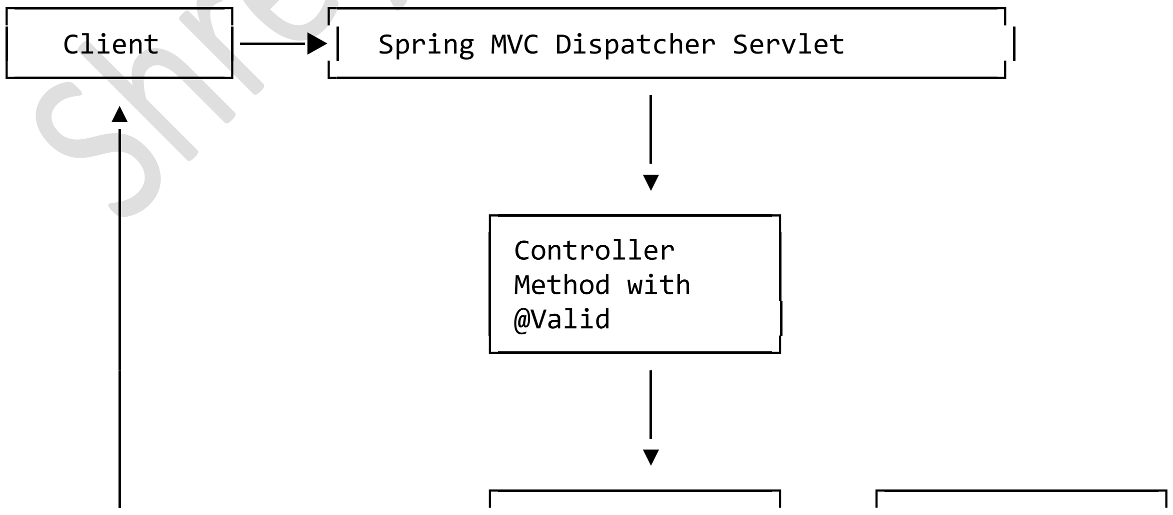


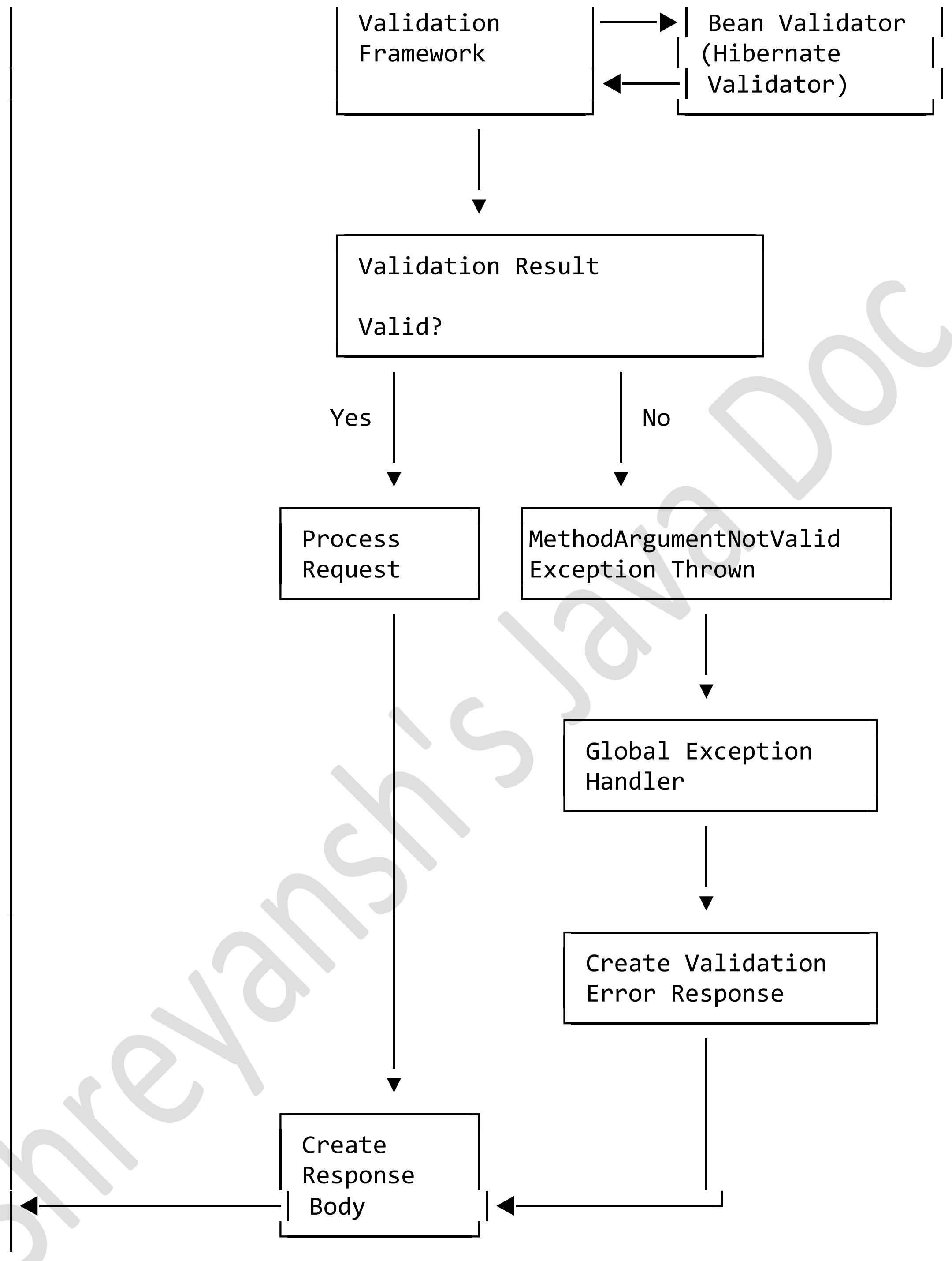
This diagram illustrates:

1. The client sends a request to the controller
2. The controller may validate the request using @Valid
3. If validation fails, MethodArgumentNotValidException is thrown
4. If validation passes, the request is processed by the service
5. The service may throw custom exceptions like ResourceNotFoundException
6. The global exception handler catches all exceptions
7. The handler creates an appropriate error response
8. The error response is returned to the client

Validation Flow Diagram

Let's also visualize the validation flow:





Validation Framework
Bean Validator (Hibernate Validator)
Validation Result
Valid?
Yes
No
Process Request
MethodArgumentNotValid Exception Thrown
Global Exception Handler
Create Validation Error Response
Create Response Body

This diagram shows:

1. The client sends a request with data
2. The Spring MVC dispatcher servlet routes the request to the controller
3. The controller method with @Valid annotation triggers validation
4. The validation framework uses Hibernate Validator to validate the request
5. If validation passes, the request is processed normally

6. If validation fails, MethodArgumentNotValidException is thrown
7. The global exception handler catches the exception
8. The handler creates a validation error response
9. The error response is returned to the client

Advanced Validation Techniques

Custom Validation Annotations

Sometimes the built-in validation annotations aren't enough. Let's create a custom validation annotation to check if a password is strong:

```
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.*;

@Documented
@Constraint(validatedBy = StrongPasswordValidator.class)
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StrongPassword {
    String message() default "Password is not strong enough";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

And its validator:

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class StrongPasswordValidator implements
ConstraintValidator<StrongPassword, String> {

    @Override
    public boolean isValid(String password, ConstraintValidatorContext
context) {
        if (password == null) {
            return false;
        }

        boolean hasLength = password.length() >= 8;
        boolean hasUpperCase = password.matches(".*[A-Z].*");
        boolean hasLowerCase = password.matches(".*[a-z].*");
        boolean hasDigit = password.matches(".*\\d.*");
        boolean hasSpecialChar = password.matches(".*[!@#$%^&*()_+\\-
=\\[\\]{};':\"\\\\|,.<>/?].*");

        return hasLength && hasUpperCase && hasLowerCase && hasDigit &&
hasSpecialChar;
    }
}
```

Now we can use it in our UserDto:

```
public class UserDto {
    // Other fields

    @NotBlank(message = "Password is required")
    @StrongPassword
    private String password;

    // Other fields, getters, and setters
}
```

Group Validation

Sometimes you need different validation rules for different operations. Spring's validation groups can help:

```
// Define validation groups
public interface ValidationGroups {
    interface Create {}
    interface Update {}
}

public class UserDto {

    @Null(groups = ValidationGroups.Create.class, message = "ID must be null
for creation")
    @NotNull(groups = ValidationGroups.Update.class, message = "ID is
required for update")
    private Long id;

    @NotBlank(message = "Name is required")
    private String name;

    @NotBlank(message = "Email is required")
    @Email(message = "Email should be valid")
    private String email;

    @StrongPassword(groups = ValidationGroups.Create.class)
    private String password;

    // Other fields, getters, and setters
}
```

In the controller:

```
@PostMapping
public ResponseEntity<UserDto> createUser(
        @Validated(ValidationGroups.Create.class) @RequestBody UserDto
userDto) {
    // Implementation
}

@PutMapping("/{id}")
public ResponseEntity<UserDto> updateUser(
        @PathVariable Long id,
        @Validated(ValidationGroups.Update.class) @RequestBody UserDto
userDto) {
    // Implementation
}
```

Programmatic Validation

Sometimes you need to perform validation programmatically:

```
import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

@Component
public class UserDtoValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return UserDto.class.equals(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        UserDto userDto = (UserDto) target;

        // Complex validation logic
        if (userDto.getName() != null &&
userDto.getName().equals(userDto.getEmail())) {
            errors.rejectValue("email", "email.same.as.name", "Email cannot
be the same as name");
        }

        // More validation rules
    }
}
```

Using it in a controller:

```
@RestController
@RequestMapping("/api/users")
```

```java
public class UserController {

    private final UserService userService;
    private final UserDtoValidator userDtoValidator;

    public UserController(UserService userService, UserDtoValidator
userDtoValidator) {
        this.userService = userService;
        this.userDtoValidator = userDtoValidator;
    }

    @InitBinder("userDto")
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(userDtoValidator);
    }

    @PostMapping
    public ResponseEntity<UserDto> createUser(@Valid @RequestBody UserDto
userDto) {
        // Implementation
    }

    // Other methods
}
```

Testing Exception Handling and Validation

It's important to test your exception handling and validation. Here's how to do it with JUnit and MockMvc:

```java
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(UserController.class)
public class UserControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    public void createUser_WithInvalidEmail_ShouldReturnBadRequest() throws
Exception {
        String userJson = "{\"name\":\"John Doe\",\"email\":\"invalid-
email\",\"password\":\"Password1!\",\"age\":25}";

        mockMvc.perform(post("/api/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content(userJson))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.status").value(400))
                .andExpect(jsonPath("$.error").value("Bad Request"))

.andExpect(jsonPath("$.validationErrors[0].field").value("email"))

.andExpect(jsonPath("$.validationErrors[0].message").value("Email should be
valid"));
    }

    @Test
    public void
createUser_WithResourceNotFoundException_ShouldReturnNotFound() throws
Exception {
        String userJson = "{\"name\":\"John
Doe\",\"email\":\"john@example.com\",\"password\":\"Password1!\",\"age\":25}"
;

        when(userService.createUser(any())).thenThrow(
                new ResourceNotFoundException("User", "email",
"john@example.com"));

        mockMvc.perform(post("/api/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content(userJson))
                .andExpect(status().isNotFound())
                .andExpect(jsonPath("$.status").value(404))
                .andExpect(jsonPath("$.error").value("Not Found"))
                .andExpect(jsonPath("$.message").value("User not found with
email: 'john@example.com'"));
    }
}
```

Best Practices for Exception Handling and Validation

Here are some best practices to follow:

Exception Handling Best Practices

1. **Create a hierarchy of custom exceptions** - This makes it easier to handle related exceptions in a similar way.
2. **Use meaningful exception names and messages** - This helps with debugging and provides better information to API consumers.
3. **Don't expose sensitive information** - Never include stack traces, database details, or other sensitive information in error responses.
4. **Use appropriate HTTP status codes** - Match your exceptions to the most appropriate HTTP status code.
5. **Log exceptions appropriately** - Log detailed information for server-side debugging, but return sanitized responses to clients.
6. **Handle unexpected exceptions gracefully** - Always have a catch-all handler for unexpected exceptions.