Controller/Service Interface

jim stafford

Fall 2024 v2022-07-15: Built: 2024-11-19 21:32 EST

Table of Contents

1. Introduction . . . 1
    1.1. Goals . . . 1
    1.2. Objectives . . . 1
2. Roles . . . 3
3. Error Reporting . . . 4
    3.1. Complex Object Result . . . 4
    3.2. Thrown Exception . . . 4
    3.3. Exceptions . . . 5
    3.4. Checked or Unchecked? . . . 6
    3.5. Candidate Client Exceptions . . . 7
    3.6. Service Errors . . . 8
4. Controller Exception Advice . . . 10
    4.1. Service Method with Exception Logic . . . 10
    4.2. Controller Advice Class . . . 11
    4.3. Advice Exception Handlers . . . 12
5. Summary . . . 13

Chapter 1. Introduction

Many times we may think of a service from the client's perspective and term everything on the other side of the HTTP connection to be "the service". That is OK from the client's perspective, but even a moderately-sized service — there are normally a few layers of classes playing a certain architectural role and that front-line controller we have been working with should primarily be a "web facade" interfacing the business logic to the outside world.

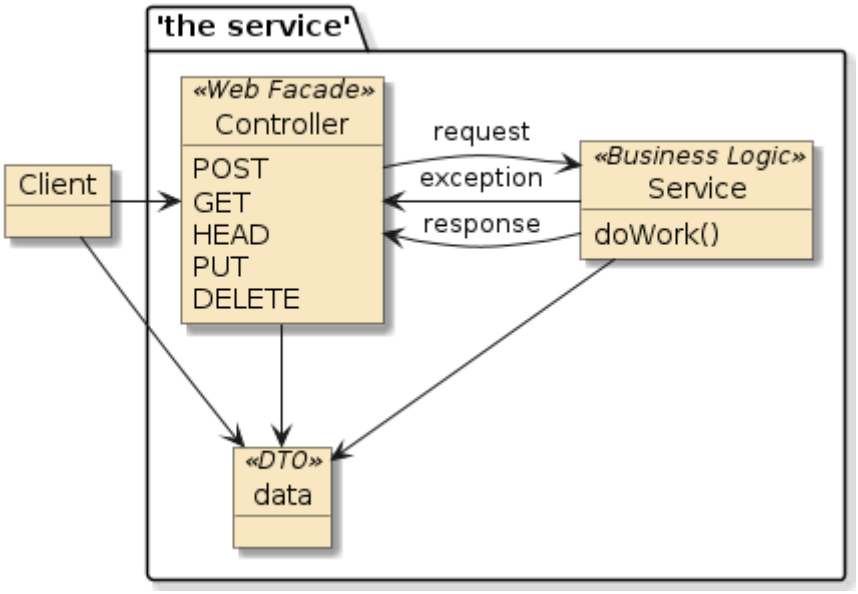

Figure 1. Controller/Service Relationship

In this lecture we are going to look more closely at how the overall endpoint breaks down into a set of "facade" and "business logic" pattern players and lay the groundwork for the "Data Transfer Object" (DTO) covered in the next lecture.

1.1. Goals

The student will learn to:

- identify the Controller class as having the role of a facade
- encapsulate business logic within a separate service class
- establish some interface patterns between the two layers so that the web facade is as clean as possible

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. implement a service class to encapsulate business logic
2. turn `@RestController` class into a facade and delegate business logic details to an injected service class
3. identify error reporting strategy options

4. identify exception design options
5. implement a set of condition-specific exceptions
6. implement a Spring `@RestControllerAdvice` class to offload exception handling and error reporting from the `@RestController`

Chapter 2. Roles

In an N-tier, distributed architecture there is commonly a set of patterns to apply to our class design.

[svciface arch] | *svciface_arch.png*

Figure 2. Controller/Service Relationship

- **Business Logic** - primary entry point for doing work. The business logic knows the why and when to do things. Within the overall service — this is the class (or set of classes) that make up the core service.
- **Data Transfer Object (DTO)** - used to describe requested work to the business logic or results from the business logic. In small systems, the DTO may be the same as the business objects (BO) stored in the database — but the specific role that will be addressed here is communicating outside of the overall service.
- **Facade** - this provides an adapter around the business logic that translates commands from various protocols and libraries — into core language commands.

I will cover DTOs in more detail in the next lecture — but relative to the client, facade, and business logic — know that all three work on the same type of data. The DTO data types pass thru the controller without a need for translation — other than what is required for communications.

Our focus in this lecture is still the controller and will now look at some controller/service interface strategies that will help develop a clean web facade in our controller classes.

Chapter 3. Error Reporting

When an error occurs — whether it be client or internal server errors — we want to have access to useful information that can be used to correct or avoid the error in the future. For example, if a client asks for information on a particular account that cannot be found, it would save minutes to hours of debugging to know whether the client requested a valid account# or the bank's account repository was not currently available.

We have one of two techniques to report error details: complex object result and thrown exception.

Design a way to allow low-level code report context of failures

The place were the error is detected is normally the place with the most amount of context details known. Design a way to have the information from the detection spot propagated to the error handling.

3.1. Complex Object Result

For the complex object result approach, each service method returns a complex result object (similar in concept to ResponseEntity). If the business method is:

- **successful**: the requested result is returned
- **unsuccessful**: the returned result expresses the error

The returned method type is complex enough to carry both types of payloads.

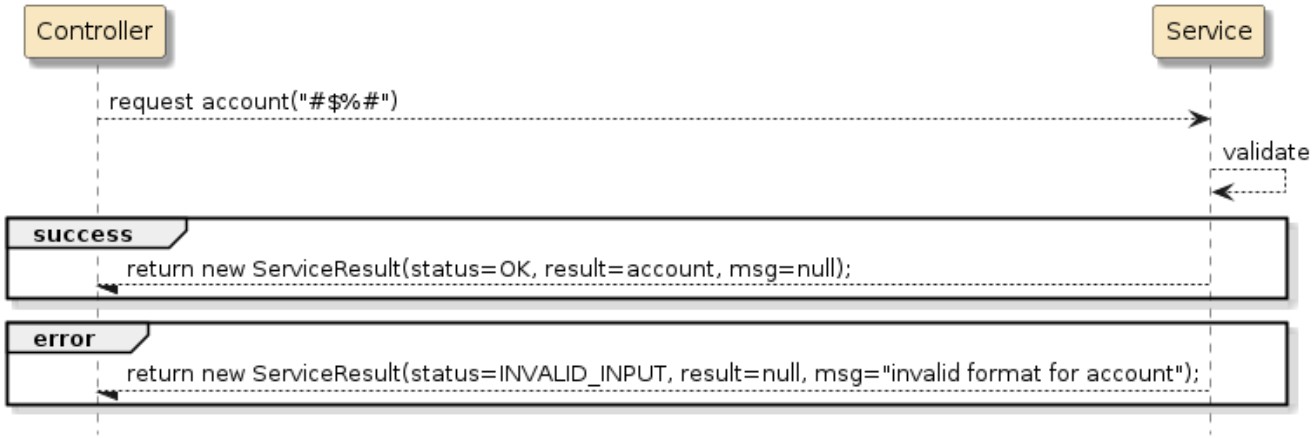

Figure 3. Service Returns Complex Object with Status and Error

Complex return objects require handling logic in caller

The complex result object requires the caller to have error handling logic ready to triage and react to the various responses. Anything that is not immediately handled may accidentally be forgotten.

3.2. Thrown Exception

For the thrown exception case, exceptions are declared to carry failure-specific error reporting. The

business method only needs to declare the happy path response in the return of the method and optionally declare try/catch blocks for errors it can address.

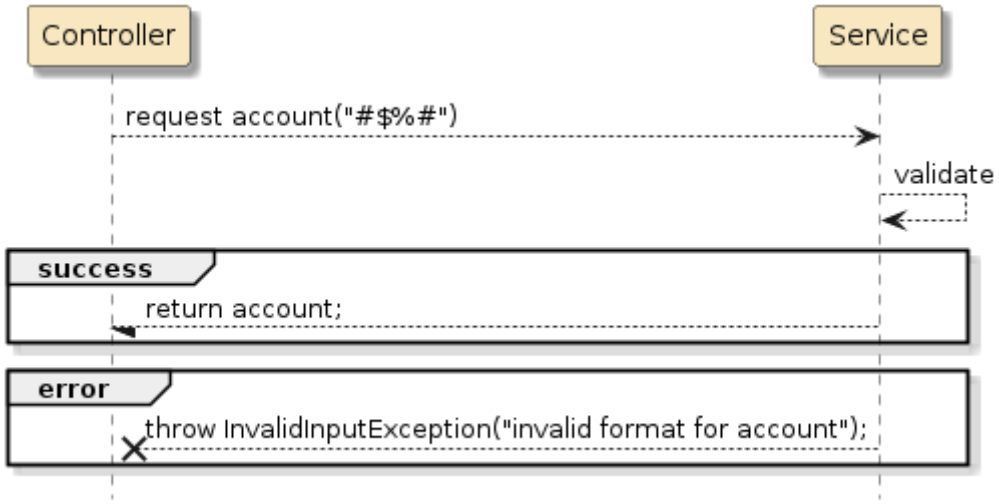

Figure 4. Service Throws Exception of Type of Error

Thrown exceptions give the caller the option to handle or delegate

The thrown exception technique gives the caller the option to construct a try/catch block and immediately handle the error or to automatically let it propagate to a caller that can address the issue.

Either technique will functionally work. However, returning the complex object versus exception will require manual triage logic on the receiving end. As long as we can create error-specific exceptions, we can create some cleaner handling options in the controller.

3.3. Exceptions

Going the exception route, we can start to consider:

- what specific errors should our services report?
- what information is reported?
  - timestamp
  - (descriptive? redacted?) error text
- are there generalizations or specializations?

The HTTP organization of status codes is a good place to start thinking of error types and how to group them (i.e., it is used by the world's largest information system — the WWW). HTTP defines two primary types of errors:

- client-based
- server-based

It could be convenient to group them into a single hierarchy — depending on how we defined the details of the exceptions.

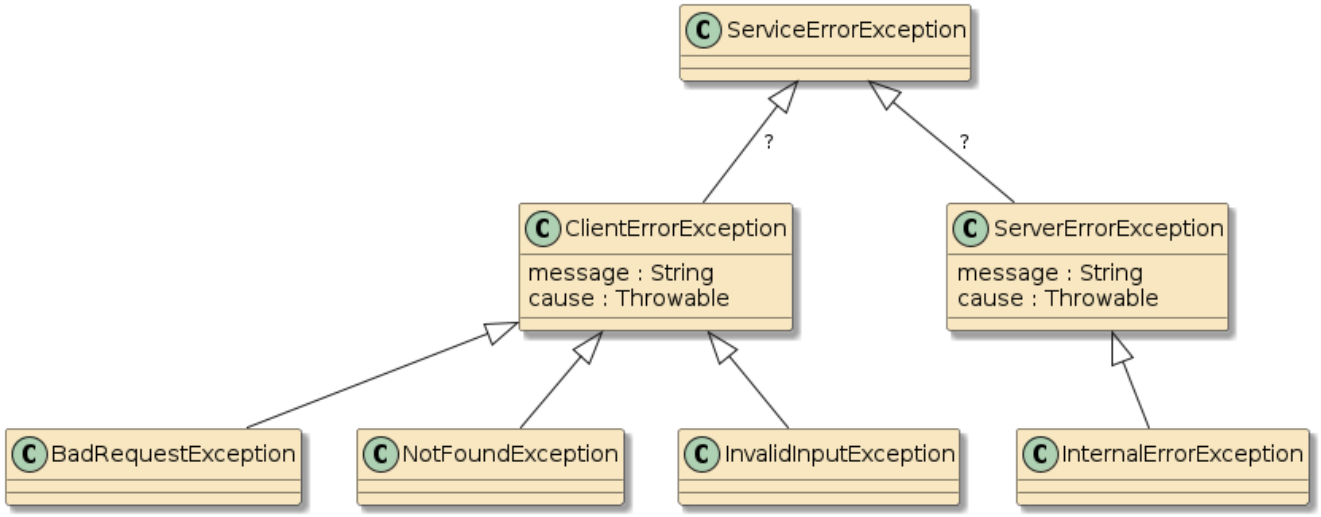

Figure 5. Example Service Exception Hierarchy

From the start, we can easily guess that our service method(s) might fail because

- **NotFoundException**: the target entity was not found
- **InvalidInputException**: something wrong with the content of what was requested
- **BadRequestException**: request was not understood or erroneously requested
- **InternalErrorException**: infrastructure or something else internal went bad

We can also assume that we would need, at a minimum

- a message - this would ideally include IDs that are specific to the context
- cause exception - commonly something wrapped by a server error

3.4. Checked or Unchecked?

Going the exception route — the most significant impact to our codebase will be the choice of checked versus unchecked exceptions (i.e., RuntimeException).

- **Checked Exception** - these exceptions inherit from java.lang.Exception and are required to be handled by a try/catch block or declared as rethrown by the calling method. It always starts off looking like a good practice, but can get quite tedious when building layers of methods.
- **RuntimeException** - these exceptions inherit from java.lang.RuntimeException and not required to be handled by the calling method. This can be a convenient way to address exceptions "not dealt with here". However, it is always the caller's option to catch any exception they can specifically address.

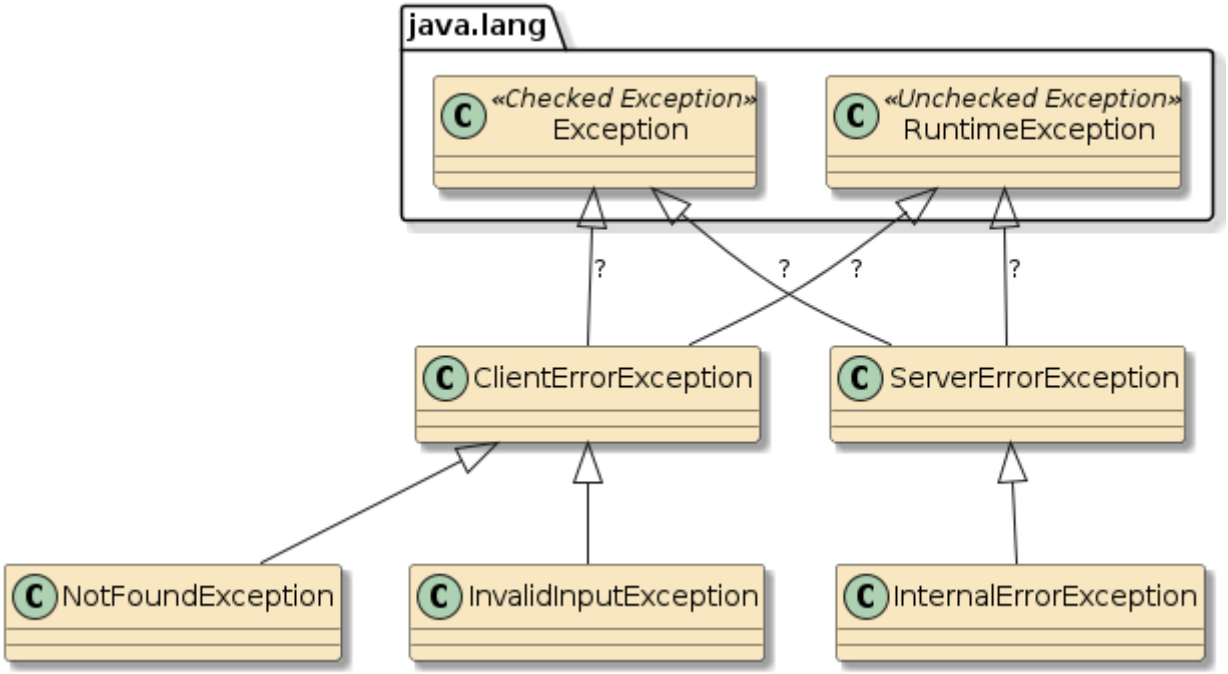

Figure 6. Should Reported Exceptions be Checked or Unchecked?

If we choose to make them different (i.e., ServerErrorException unchecked and ClientErrorException checked), we will have to create separate inheritance hierarchies (i.e., no common ServiceErrorException parent).

3.5. Candidate Client Exceptions

The following is a candidate implementation for client exceptions. I am going to go the seemingly easy route and make them unchecked/RuntimeExceptions — but keep them in a separate hierarchy from the server exceptions to allow an easy change. Complete examples can be located in the repository

Candidate Client Exceptions

```
public abstract class ClientErrorException extends RuntimeException {
    protected ClientErrorException(Throwable cause) {
        super(cause);
    }
    protected ClientErrorException(String message, Object...args) {
        super(String.format(message, args)); ①
    }
    protected ClientErrorException(Throwable cause, String message, Object...args) {
        super(String.format(message, args), cause);
    }

    public static class NotFoundException extends ClientErrorException {
        public NotFoundException(String message, Object...args)
                {  super(message, args); }
        public NotFoundException(Throwable cause, String message, Object...args)
                { super(cause, message, args); }
    }
```

```
    public static class InvalidInputException extends ClientErrorException {
        public InvalidInputException(String message, Object...args)
                {  super(message, args); }
        public InvalidInputException(Throwable cause, String message, Object...args)
                { super(cause, message, args); }
    }
}
```

① encourage callers to add instance details to exception by supplying built-in, optional formatter

The following is an example of how the caller can instantiate and throw the exception based on conditions detected in the request.

Example Client Exception Throw

```
if (null==gesture) {
    throw new ClientErrorException
            .NotFoundException("gesture type[%s] not found", gestureType);
}
```

3.6. Service Errors

The following is a candidate implementation for server exceptions. These types of errors are commonly unchecked.

```
public abstract class ServerErrorException extends RuntimeException {
    protected ServerErrorException(Throwable cause) {
        super(cause);
    }
    protected ServerErrorException(String message, Object...args) {
        super(String.format(message, args));
    }
    protected ServerErrorException(Throwable cause, String message, Object...args) {
        super(String.format(message, args), cause);
    }

    public static class InternalErrorException extends ServerErrorException {
        public InternalErrorException(String message, Object...args)
                {  super(message, args); }
        public InternalErrorException(Throwable cause, String message, Object...args)
                { super(cause, message, args); }
    }
}
```

The following is an example of instantiating and throwing a server exception based on a caught exception.

Example Server Exception Throw

```
try {
    //...
} catch (RuntimeException ex) {
    throw new InternalErrorException(ex, ①
            "unexpected error getting gesture[%s]", gestureType); ②
}
```

① reporting source exception forward

② encourage callers to add instance details to exception by supplying built-in, optional formatter

Chapter 4. Controller Exception Advice

We saw earlier where we could register an exception handler within the controller class and how that could clean up our controller methods of noisy error handling code. I want to now build on that concept and our new concrete service exceptions to define an external controller advice that will handle all registered exceptions.

The following is an example of a controller method that is void of error handling logic because of the external controller advice we will put in place.

Example Controller Method - Void of Error Handling

```
@RestController
public class GesturesController {
...
    @RequestMapping(path=GESTURE_PATH,
            method=RequestMethod.GET,
            produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> getGesture(
            @PathVariable(name="gestureType") String gestureType,
            @RequestParam(name="target", required=false) String target) {
        //business method
        String result = gestures.getGesture(gestureType, target); ①

        String location = ServletUriComponentsBuilder.fromCurrentRequest()
                    .build().toUriString();
        return ResponseEntity
                .status(HttpStatus.OK)
                .header(HttpHeaders.CONTENT_LOCATION, location)
                .body(result);
    }
```

① handles only successful result — exceptions left to controller advice

4.1. Service Method with Exception Logic

The following is a more complete example of the business method within the service class. Based on the result of the interaction with the data access tier — the business method determines the gesture does not exist and reports that error using an exception.

Example Service Method with Exception Error Reporting Logic

```
@Service
public class GesturesServiceImpl implements GesturesService {
    @Override
    public String getGesture(String gestureType, String target) {
        String gesture = gestures.get(gestureType); //data access method
        if (null==gesture) {
            throw new ClientErrorException ①
```

```
                    .NotFoundException("gesture type[%s] not found", gestureType);
        } else {
            String response = gesture + (target==null ? "" : ", " + target);
            return response;
        }
    }
...
```

① service reporting details of error

4.2. Controller Advice Class

The following is a controller advice class. We annotate this with `@RestControllerAdvice` to better describe its role and give us the option to create fully annotated handler methods.

My candidate controller advice class contains an internal helper method that programmatically builds a ResponseEntity. The type-specific exception handler must translate the specific exception into a HTTP status code and body. A more complete example — designed to be a base class to concrete `@RestControllerAdvice` classes — can be found in the repository.

Controller Advice Class

```
package info.ejava.examples.svc.httpapi.gestures.controllers;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice( ①
// wraps ==> @ControllerAdvice
//           wraps ==> @Component
    basePackageClasses = GesturesController.class) ②
public class ExceptionAdvice {  /③
    //internal helper method
    protected ResponseEntity<String> buildResponse(HttpStatus status, ④
                      String text) { ⑤

        return ResponseEntity
                .status(status)
                .body(text);
    }
...
```

① `@RestControllerAdvice` denotes this class as a `@Component` that will handle thrown exceptions

② optional annotations can be used to limit the scope of this advice to certain packages and controller classes

③ handled thrown exceptions will return the DTO type for this application — in this case just `text/plain`

④ type-specific exception handlers must map exception to an HTTP status code

⑤ type-specific exception handlers must produce error text

Example assumes DTO type is `plain/test` string

This example assumes the DTO type for errors is a `text/plain` string. More robust response type would be part of an example using complex DTO types.

4.3. Advice Exception Handlers

Below are the candidate type-specific exception handlers we can use to translate the context-specific information from the exception to a valuable HTTP response to the client.

Advice Exception Handlers

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

import static info.ejava.examples.svc.httpapi.gestures.svc.ClientErrorException.*;
import static info.ejava.examples.svc.httpapi.gestures.svc.ServerErrorException.*;
...
    @ExceptionHandler(NotFoundException.class) ①
    public ResponseEntity<String> handle(NotFoundException ex) {
        return buildResponse(HttpStatus.NOT_FOUND, ex.getMessage()); ②
    }
    @ExceptionHandler(InvalidInputException.class)
    public ResponseEntity<String> handle(InvalidInputException ex) {
        return buildResponse(HttpStatus.UNPROCESSABLE_ENTITY, ex.getMessage());
    }
    @ExceptionHandler(InternalErrorException.class)
    public ResponseEntity<String> handle(InternalErrorException ex) {
        log.warn("{}", ex.getMessage(), ex); ③
        return buildResponse(HttpStatus.INTERNAL_SERVER_ERROR, ex.getMessage());
    }
    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntimeException(RuntimeException ex) {
        log.warn("{}", ex.getMessage(), ex); ③
        String text = String.format(
                "unexpected error executing request: %s", ex.toString());
        return buildResponse(HttpStatus.INTERNAL_SERVER_ERROR, text);
    }
```

① annotation maps the handler method to a thrown exception type

② handler method receives exception and converts to a ResponseEntity to be returned

③ the unknown error exceptions are candidates for mandatory logging

Chapter 5. Summary

In this module we:

- identified the `@RestController` class' role is a "facade" for a web interface
- encapsulated business logic in a `@Service` class
- identified data passing between clients, facades, and business logic is called a Data Transfer Object (DTO). The DTO was a string in this simple example, but will be expanded in the content lecture
- identified how exceptions could help separate successful business logic results from error path handling
- identified some design choices for our exceptions
- identified how a controller advice class can be used to offload exception handling