

```

package
com.codingshuttle.springbootwebtutorial.SpringbootWebtutorial.advic
es;

import org.jspecify.annotations.Nullable;
import org.springframework.core.MethodParameter;
import org.springframework.http.MediaType;
import org.springframework.http.converter.HttpMessageConverter;
import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseBody
Advice;

@RestControllerAdvice
public class GlobalResponseHandler implements
ResponseBodyAdvice<Object> {

    @Override
    public boolean supports(MethodParameter returnType, Class<?
extends HttpMessageConverter<?>> converterType) {
        return true;
    }

    @Override
    public @Nullable Object beforeBodyWrite(@Nullable Object body,
MethodParameter returnType, MediaType selectedContentType,
Class<? extends HttpMessageConverter<?>> selectedConverterType,
ServerHttpRequest request, ServerHttpResponse response) {
        if(body instanceof ApiResponse<?>){
            return body;
        }

        return new ApiResponse<>(body);
    }
}

```

This is a global response wrapper using Spring's `ResponseBodyAdvice`.

It's an advanced but very professional pattern. Let me explain exactly what it does, why it's used, and how it works line-by-line.

---

## 🌸 Big picture (what this class does)

It automatically wraps every controller response into a standard `ApiResponse` object, so your API responses are consistent, without manually wrapping them in each controller.

Before this:

json

 Copy code

```
{
  "id": 1,
  "name": "Ram"
}
```

After this:

json

 Copy code

```
{
  "data": {
    "id": 1,
    "name": "Ram"
  }
}
```

And you did not modify any controller code.

## ★ Why use `ResponseBodyAdvice` ?

Without it, you'd have to write this everywhere:

```
java                                                                    Copy code

return new ApiResponse<>(employeeDTO);
```

With `ResponseBodyAdvice` :

- Controllers return **pure business data**
- A global layer formats the response
- Clean architecture
- No duplication
- Easy to change response format later

This is how **real production APIs** are designed.

---

## 🧩 Your code — explained line by line

---

### 1 `@RestControllerAdvice`

```
java                                                                    Copy code

@RestControllerAdvice
public class GlobalResponseHandler
```

This means:

- Applies to **all** `@RestController` s
- Runs **after controller execution**
- Before response is written to HTTP body

## 2 implements `ResponseBodyAdvice<Object>`

java

 Copy code

```
implements ResponseBodyAdvice<Object>
```

This tells Spring:

“Before sending ANY response body to the client, give me a chance to modify it.”

The generic `<Object>` means:

- It applies to all response types

---

## 3 `supports()` method

java

 Copy code

```
@Override
public boolean supports(MethodParameter returnType,
                        Class<?> extends HttpMessageConverter<?>> converterType) {
    return true;
}
```

This method decides:

“Should `beforeBodyWrite()` run for this response?”

Since you return `true`:

- The advice applies to all controller responses
- 👉 You can restrict this later (e.g., only JSON, only certain controllers).

#### beforeBodyWrite() method (MOST IMPORTANT)

java

 Copy code

```
@Override
public @Nullable Object beforeBodyWrite(
    @Nullable Object body,
    MethodParameter returnType,
    MediaType selectedContentType,
    Class<? extends HttpMessageConverter<?>> selectedConverterType,
    ServerHttpRequest request,
    ServerHttpResponse response)
```

This method:

- Receives the **actual response body**
- Lets you modify it
- Whatever you return here is sent to the client

## 5 Prevent double-wrapping (VERY IMPORTANT)

java

 Copy code

```
if (body instanceof ApiResponse<?>) {  
    return body;  
}
```

This check avoids this nightmare ❌:

json

 Copy code

```
{  
  "data": {  
    "data": {  
      "data": {...}  
    }  
  }  
}
```


So:

- If controller already returned `ApiResponse`
- Do NOT wrap again

- ✓ Correct
  - ✓ Professional
  - ✓ Mandatory
-

## 6 Wrap normal responses

java

 Copy code

```
return new ApiResponse<>(body);
```

If controller returned:

- EmployeeDTO
- List<EmployeeDTO>
- String
- null

They all get wrapped into:

java

 Copy code

```
ApiResponse<T>
```

## How Spring executes this (flow)

pgsql

 Copy code

Client

↓

Controller returns data

↓

ResponseBodyAdvice.beforeBodyWrite()

↓

Body wrapped into ApiResponse

↓

Jackson serializes JSON

↓

Response sent

## ★ What is ApiResponse (expected structure)

Typically:

java

 Copy code

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ApiResponse<T> {

    private T data;
    private String message;
    private boolean success;
}
```

Or minimal:

java

 Copy code

```
public record ApiResponse<T>(T data) {}
```

## ★ Why this is a GOOD design

Benefit	Explanation
Consistent responses	Same JSON structure everywhere
Thin controllers	No response formatting logic
Easy future changes	Modify response format in one place
Works with errors too	Complements <code>ApiError</code>
Industry standard	Used in large systems



## ⚠ Important things to watch out for

### 1 Do NOT wrap error responses

Usually handled automatically because:

- Exceptions return `ResponseEntity<ApiError>`
- Your check avoids double wrapping

If needed, you can explicitly skip:

```
java

if (body instanceof ApiError) {
    return body;
}
```

 Copy code

### 2 Handle `null` responses

If controller returns `void` or `null`:

```
java

return new ApiResponse<>(null);
```

 Copy code

This is fine, but you may want:

```
json

{
  "data": null
}
```

 Copy code

Or customize.

---

### 3 Media type control (optional)

If you only want JSON:

java

 Copy code

```
if (!MediaType.APPLICATION_JSON.equals(selectedContentType)) {  
    return body;  
}
```

---

### 🏆 Final verdict on your code

- ✅ Correct
- ✅ Professional
- ✅ Scalable
- ✅ Clean architecture
- ✅ Production-ready

You are now using:

- `@RestControllerAdvice` for errors
- `ResponseBodyAdvice` for responses

This is senior-level Spring Boot design.

---

### 🔑 One-line summary

`ResponseBodyAdvice` lets you globally intercept and wrap all controller responses into a standard API format without touching controller code.

then while making controller methods why we are defining the response entity then

Excellent question — this is exactly the **right confusion** to have when you introduce `ResponseBodyAdvice`. Let's resolve it **cleanly and precisely**.

---

### Short answer (core idea)

`ResponseEntity` is NOT about response body structure.  
It is about HTTP metadata (status, headers).

Even with `ResponseBodyAdvice`, you still need `ResponseEntity` when you want to control HTTP status or headers.

---

### Think of response in TWO parts

#### HTTP METADATA

- Status code (200, 201, 404, 400...)
- Headers (Location, Cache-Control, etc.)

#### RESPONSE BODY

- JSON content
- Wrapped by `ApiResponse` using `ResponseBodyAdvice`

These are **separate concerns**.

---

## What `ResponseBodyAdvice` actually does

It only touches:

java

 Copy code

```
@ResponseBody
```

👉 Only the body, not:

- Status code
- Headers

So this:

java

 Copy code

```
return new ApiResponse<>(data);
```

does NOT allow you to say:

- 201 Created
  - 404 Not Found
  - Custom headers
-

## ✅ Why controllers still use `ResponseEntity`

### Case 1: Default success (200 OK)

You can do:

java

 Copy code

```
@GetMapping("/{id}")
public EmployeeDTO getEmployee(@PathVariable Long id) {
    return employeeService.getEmployeeById(id);
}
```

Flow:

- Controller returns DTO
- `ResponseBodyAdvice` wraps it
- HTTP status defaults to 200 OK

✅ `ResponseEntity` NOT needed

## Case 2: You want 201 CREATED

java

 Copy code

```
@PostMapping
public ResponseEntity<EmployeeDTO> create(@RequestBody EmployeeDTO dto) {
    EmployeeDTO saved = service.create(dto);
    return ResponseEntity.status(HttpStatus.CREATED).body(saved);
}
```

Why `ResponseEntity` ?

- Because HTTP status MUST be 201
- `ResponseBodyAdvice` will still wrap the body

Final response:

arduino

 Copy code

```
HTTP 201 Created
{
  "data": { ... }
}
```

## Case 3: You want custom headers

java

 Copy code

```
return ResponseEntity
    .created(URI.create("/employees/" + id))
    .header("X-App-Version", "1.0")
    .body(employeeDTO);
```

Again:

- Only `ResponseEntity` can do this
- `ResponseBodyAdvice` will still wrap the body

## 🧠 How both work together (important)

sql

Copy code

Controller **returns** `ResponseEntity<T>`

↓

Spring extracts:

- HTTP status
- headers
- body

↓

`ResponseBodyAdvice.beforeBodyWrite()` runs **on BODY ONLY**

↓

**Final** response sent

So:

- `ResponseEntity` controls status & headers
- `ResponseBodyAdvice` controls body shape

They do not overlap.

## ❌ Common misunderstanding

"If I have `ResponseBodyAdvice`, I don't need `ResponseEntity`"

❌ Incorrect.

Correct understanding:

`ResponseBodyAdvice` replaces manual body wrapping, NOT HTTP control.

## Best practice guideline (very important)

Situation	Use ResponseEntity?
Simple GET (200 OK)	✗ Not required
POST create (201)	✓ Required
DELETE (204 No Content)	✓ Required
Error responses	✓ Required
Custom headers	✓ Required
Default success	Optional

## Mental model (remember this)

ResponseEntity = HTTP envelope

ResponseBodyAdvice = JSON body formatter



## ✅ Example (clean controller with your setup)


java

 Copy code

```
@PostMapping
public ResponseEntity<EmployeeDTO> create(@RequestBody EmployeeDTO dto) {
    EmployeeDTO saved = service.create(dto);
    return ResponseEntity.status(HttpStatus.CREATED).body(saved);
}
```

With `ResponseBodyAdvice`, final JSON:

json

 Copy code

```
{
  "data": {
    "id": 1,
    "name": "Ram"
  }
}
```

## 💡 Final one-line answer

We still use `ResponseEntity` in controllers to control HTTP status codes and headers; `ResponseBodyAdvice` only standardizes the response body, not the HTTP metadata.

in controller we return `ResponseEntity<EmployeeDTO>` then why we are sending the `EmployeeDTO` from controller ...is still `EmployeeDTO` is returning when we are using global response handler

This is an **excellent, subtle question**, and the confusion is very natural when you introduce a **global response handler**.

Let's clear it **precisely**.

### Short, correct answer

Yes — the controller still returns `EmployeeDTO`,  
but it is **NOT** the final response sent to the client.

The `EmployeeDTO` you return is an **intermediate body**.

Your `GlobalResponseHandler` (`ResponseBodyAdvice`) intercepts it and **wraps it before sending**.

### Think in TWO STAGES (very important)

#### • Stage 1: Controller return value (Java-level)

java

 Copy code

```
return ResponseEntity.ok(employeeDTO);
```

At this point:

- The controller returns `EmployeeDTO`
- Wrapped in `ResponseEntity`
- This is **inside Spring**, not yet sent to client

- **Stage 2: GlobalResponseHandler modifies the body (HTTP-level)**

Your `ResponseBodyAdvice` runs after controller execution:

```
java

@Override
public Object beforeBodyWrite(Object body, ...) {
    return new ApiResponse<>(body);
}
```

 Copy code

So:

Stage	Body value
Controller returns	EmployeeDTO
ResponseBodyAdvice receives	EmployeeDTO
Final HTTP response	ApiResponse<EmployeeDTO>

## Visual Flow (VERY IMPORTANT)

csharp

```
Controller
  ↓ returns
ResponseEntity<EmployeeDTO>
  ↓
Spring extracts body → EmployeeDTO
  ↓
ResponseBodyAdvice.beforeBodyWrite()
  ↓
new ApiResponse<>(EmployeeDTO)
  ↓
Jackson serializes JSON
  ↓
Client receives ApiResponse<EmployeeDTO>
```

 Copy code

## ? Then why is controller return type `ResponseEntity<EmployeeDTO>` ?

Because:

### ✓ `ResponseEntity<T>` controls HTTP metadata

- Status code
- Headers

### ✓ `T ( EmployeeDTO )` is just the raw body

- It will be wrapped later
- Controller does NOT need to know final format

## 🧠 Key insight (this removes all confusion)

Controller return type is NOT the same as the final response body.

It is just:

- The input to the response pipeline
- Not the output sent to the client

## ✓ Example (your exact setup)

### Controller

java

 Copy code

```
@GetMapping("/{id}")
public ResponseEntity<EmployeeDTO> getEmployee(@PathVariable Long id) {
    EmployeeDTO dto = service.getEmployeeById(id);
    return ResponseEntity.ok(dto);
}
```

## GlobalResponseHandler

```
java Copy code  
  
@Override  
public Object beforeBodyWrite(Object body, ...) {  
    return new ApiResponse<>(body);  
}
```

## Actual HTTP response

```
json Copy code  
  
{  
  "data": {  
    "id": 1,  
    "name": "Ram"  
  }  
}
```

- ✓ Controller returned `EmployeeDTO`
- ✓ Client received `ApiResponse<EmployeeDTO>`

## ✗ What does NOT happen

- ✗ `EmployeeDTO` is NOT directly sent to client
- ✗ Controller does NOT bypass `GlobalResponseHandler`
- ✗ `ResponseEntity` generic type does NOT define final JSON

## 🏆 Why this design is GOOD (professional reason)

Separation of concerns:

Layer	Responsibility
Controller	Business data + HTTP status
Service	Business logic
ResponseBodyAdvice	Response formatting
Jackson	JSON serialization

This makes your system:

- Clean
- Flexible
- Easy to refactor
- Easy to standardize responses

## 🔑 Mental model (remember this forever)

Controller returns raw data  
GlobalResponseHandler decides final response shape

Project ▾

SpringbootWebtutorial D:\SpringbootWebtutorial

src

main

java

com.codingshuttle.springboot

advice

ApiError

ApiResponse

GlobalExceptionHandler

GlobalResponseHandler

annotations

EmployeeRoleValidator

EmployeeRoleValidator

ValidAge

ValidAgeValidator

configs

MapperConfig

controllers

EmployeeController

EmployeeController

EmployeeController

EmployeeController

dto

EmployeeDTO

entities

ServiceLayer.java

SpringbootWebtutorialApplication.java

GlobalExceptionHandler.java

GlobalResponseHandler.java

ApiResponse.java

```
import com.fasterxml.jackson.annotation.JsonFormat;
import lombok.Data;

import java.time.LocalDateTime;

@Data
public class ApiResponse<T>{

    @JsonFormat(pattern = "hh:mm:ss dd-MM-yyyy")
    private LocalDateTime timestamp;

    private T data;

    private ApiError error;

    public ApiResponse() {
        this.timestamp=LocalDateTime.now();
    }

    public ApiResponse(T data){
        this();
        this.data=data;
    }

    public ApiResponse(ApiError error){
        this();
        this.error=error;
    }
}
```

Project ▾

- SpringbootWebtutorial D:\SpringbootWebtutorial
- .idea
- .mvn
- src
  - main
    - java
      - com.codingshuttle.springbootwebtutorial
        - advice
          - ApiError
          - ApiResponse
          - GlobalExceptionHandler
          - GlobalResponseHandler
        - annotations
          - EmployeeRoleValid
          - EmployeeRoleValid
          - ValidAge
          - ValidAgeValidator
        - configs
          - MapperConfig
        - controllers
          - EmployeeController
          - EmployeeController
          - EmployeeController
          - EmployeeController
        - dto
          - EmployeeDTO
        - entities

ServiceLayer.java SpringbootWebtutorialApplication.java GlobalExceptionHandler.java GlobalResponseHandler.java x ApiResponse.java

```
1 package com.codingshuttle.springbootwebtutorial.springbootwebtutorial.advice;  
2  
3 import org.jspecify.annotations.Nullable;  
4 import org.springframework.core.MethodParameter;  
5 import org.springframework.http.MediaType;  
6 import org.springframework.http.converter.HttpMessageConverter;  
7 import org.springframework.http.server.ServerHttpRequest;  
8 import org.springframework.http.server.ServerHttpResponse;  
9 import org.springframework.web.bind.annotation.RestControllerAdvice;  
10 import org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice;  
11  
12 @RestControllerAdvice new *  
13 public class GlobalResponseHandler implements ResponseBodyAdvice<Object> {  
14  
15     @Override new *  
16     public boolean supports(MethodParameter returnType, Class<?> extends HttpMessageConverter<?>> converterType) {  
17         return true;  
18     }  
19  
20     @Override no usages new *  
21     public @Nullable Object beforeBodyWrite(@Nullable Object body, MethodParameter returnType, MediaType selectedContentType, Class<?> responseBodyType, ServerHttpRequest request, ServerHttpResponse response) {  
22         if (body instanceof ApiResponse<?>){  
23             return body;  
24         }  
25  
26         return new ApiResponse<>(body);  
27     }  
28 }  
29
```



```
ServiceLayer.java  SpringbootWebtutorialApplication.java  GlobalExceptionHandler.java  GlobalResponseHandler.java  ApiRespo...

15  public class GlobalExceptionHandler {
35  //      return ResponseEntity.status(HttpStatus.NOT_FOUND).body(apiError);
36  //  }
37
38
39  @ExceptionHandler(ResourceNotFoundException.class) new *
40  @ public ResponseEntity<ApiResponse<?>> handleResourceNotFound(ResourceNotFoundException exception){
41      ApiError apiError=ApiError.builder()
42          .status(HttpStatus.NOT_FOUND)
43          .message(exception.getMessage())
44          .build();
45      return buildErrorResponseEntity(apiError);
46  }
47
48
```

```
public class GlobalExceptionHandler {  
    // }  
  
    @ExceptionHandler(Exception.class) new *  
    public ResponseEntity<ApiResponse<?>> handleInternalServerError(Exception exception){  
        ApiError apiError=ApiError.builder()  
            .status(HttpStatus.INTERNAL_SERVER_ERROR)  
            .message(exception.getMessage())  
            .build();  
        return buildErrorResponseEntity(apiError);  
    }  
}
```

```

public class GlobalExceptionHandler {
    //      return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(apiError);
    //  }

    @ExceptionHandler(MethodArgumentNotValidException.class) new *
    public ResponseEntity<ApiResponse<?>> handleInputValidationError(MethodArgumentNotValidException exception){
        List<String> errors=exception
            .getBindingResult() BindingResult
            .getAllErrors() List<ObjectError>
            .stream() Stream<ObjectError>
            .map( ObjectError error->error.getDefaultMessage()) Stream<String>
            .collect(Collectors.toList());
        ApiError apiError=ApiError.builder()
            .status(HttpStatus.BAD_REQUEST)
            .message("Input validation failed")
            .subErrors(errors)
            .build();
        return buildErrorResponseEntity(apiError);
    }

    private ResponseEntity<ApiResponse<?>> buildErrorResponseEntity(ApiError apiError) { 3 usages new *
        return new ResponseEntity<>(new ApiResponse<>(apiError),apiError.getStatus());
    }
}

```

```

package
com.codingshuttle.springbootwebtutorial.SpringbootWebtutorial.advic
es;

import com.fasterxml.jackson.annotation.JsonFormat;
import lombok.Data;

import java.time.LocalDateTime;

@Data
public class ApiResponse<T>{

    @JsonFormat(pattern = "hh:mm:ss dd-MM-yyyy")
    private LocalDateTime timestamp;

    private T data;

    private ApiError error;

    public ApiResponse() {
        this.timestamp=LocalDateTime.now();
    }

    public ApiResponse(T data){
        this();
        this.data=data;
    }

    public ApiResponse(ApiError error){
        this();
        this.error=error;
    }
}

```

the json format will come like {"timestamp": , "data":  
 ,"error": } but why error is not coming first in format before data  
 as timestamp will always come first as this() is always called first in both  
 the constructor



This is a very good question, and it reveals a common misconception about constructors vs JSON output.

👉 Constructor execution order has NOTHING to do with JSON field order.

Let's clear this precisely.

---

### 🔑 Short answer (core truth)

JSON field order is controlled by Jackson serialization,  
not by constructor execution order.

Calling `this()` first only initializes values,  
it does not decide the order of fields in JSON.

---

### 💡 Why your assumption feels logical (but is wrong)

You're thinking:

"timestamp is set first in constructor → so it must appear first  
then data → then error"

That would make sense if JSON were created by constructors.

But it's not.

---

### 🔍 What ACTUALLY decides JSON field order?

Jackson (the JSON library Spring uses) serializes objects using:

✓ Class field declaration order (by default)

OR

✓ Explicit annotations like `@JsonPropertyOrder`

It does NOT look at:

- Constructors
- `this()` calls
- Setter calls
- Runtime assignment order



## 🧩 Your class (important)

java

📄 Copy code

```
@Data
public class ApiResponse<T>{

    @JsonFormat(pattern = "hh:mm:ss dd-MM-yyyy")
    private LocalDateTime timestamp;

    private T data;

    private ApiError error;
}
```

Default JSON order will be:

json

📄 Copy code

```
{
  "timestamp": "...",
  "data": {...},
  "error": {...}
}
```

Because:

👉 That's the order of fields in the class.

## ✗ What does NOT affect JSON order

Thing	Affects JSON order?
Constructor order	✗ No
<code>this()</code> call	✗ No
Field assignment time	✗ No
Which constructor used	✗ No
Lombok @Data	✗ No

## ✅ How to control JSON field order (PROPER WAY)

### Option 1 Use `@JsonPropertyOrder` (BEST)

java

 Copy code

```
import com.fasterxml.jackson.annotation.JsonPropertyOrder;

@JsonPropertyOrder({"timestamp", "error", "data"})
@Data
public class ApiResponse<T> {

    @JsonFormat(pattern = "hh:mm:ss dd-MM-yyyy")
    private LocalDateTime timestamp;

    private T data;

    private ApiError error;

    public ApiResponse() {
        this.timestamp = LocalDateTime.now();
    }
}
```

Now JSON will be:

json

 Copy code

```
{
  "timestamp": "10:30:12 16-12-2025",
  "error": {...},
  "data": {...}
}
```


- ✓ Deterministic
- ✓ Explicit
- ✓ Professional





## Option 2 Reorder fields in class (not recommended long-term)

java

 Copy code

```
private LocalDateTime timestamp;  
private ApiError error;  
private T data;
```

Works, but fragile if someone reorders fields later.

## ★ Best practice recommendation

Always use `@JsonPropertyOrder` when order matters.

Because:

- It documents intent
- It's explicit
- It survives refactoring

## 🧠 Another important detail: null fields

If `error == null`, you probably don't want:

json

Copy code

```
"error": null
```

Add:

java

Copy code

```
@JsonInclude(JsonInclude.Include.NON_NULL)
```

java

Copy code

```
@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonPropertyOrder({"timestamp", "error", "data"})
@Data
public class ApiResponse<T> { ... }
```

Now:

- `error` appears only when present
- Order still respected

---

## 🔗 Final mental model (remember this)

Constructors initialize data.  
Jackson decides JSON shape.

They are completely independent.

---

## 🔑 One-line final answer

`error` does not appear first because JSON field order is determined by Jackson (field order or `@JsonPropertyOrder`), not by constructor execution or `this()` calls.

If it is false then the response returned by the controller will be the json output

Yes — exactly right. ✅

Your understanding is now spot-on.

### 🔑 Direct answer

If `supports()` returns `false`, then `beforeBodyWrite()` is NOT executed, and the controller's return value is serialized directly to JSON and sent to the client.

No transformation. No wrapping. No interception.

### 🧠 What happens internally (simple)

For every controller response, Spring does:

```
java
if (responseBodyAdvice.supports(...)) {
    body = responseBodyAdvice.beforeBodyWrite(...);
}
writeBodyAsJson(body);
```

📋 Copy code

So:

<code>supports()</code>	<code>beforeBodyWrite()</code>	Final response
<code>true</code>	Runs	Modified / wrapped body
<code>false</code>	Skipped	Controller output as-is

## 🔗 Example (your setup)

### Controller

java

 Copy code

```
@GetMapping("/employee")
public EmployeeDTO getEmployee() {
    return employeeDTO;
}
```

### Case 1 supports() → true

json


 Copy code

```
{
  "timestamp": "...",
  "data": {
    "id": 1,
    "name": "Ram"
  }
}
```

(GlobalResponseHandler wrapped it)

### Case 2 supports() → false

json

 Copy code

```
{
  "id": 1,
  "name": "Ram"
}
```

(Controller output serialized directly)

## Important clarification

The JSON conversion still happens in both cases.

What changes is:

- Whether your custom logic runs or not

Spring will always:

- Use `HttpMessageConverter`
- Serialize object → JSON

## Common real-world uses of `supports() == false`

✓ Skip wrapping for:

- Swagger endpoints
- Actuator endpoints
- Error responses
- File downloads
- Streaming responses

## ⌘ Mental model (final)

`supports()` = "Should I intercept?"

false → pass through untouched

true → intercept and modify

## One-line final answer

Yes. If `supports()` returns `false`, Spring skips `beforeBodyWrite()` and returns the controller's response as JSON without any transformation.