Request and Response Handling

Turning a request body into an object with @RequestBody, taking full control of a response with ResponseEntity, HTTP status codes, and content negotiation.

Intermediate 45 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/request-response-handling

---

Request and Response Handling

In Spring MVC Fundamentals we saw that `@ResponseBody` / `@RestController` automatically convert an object to JSON; in Mapping Annotations we met `consumes` / `produces`. In this lesson we go **behind** these mechanisms: how `@RequestBody` turns a JSON body into a Java object, how to take full control of a response with `ResponseEntity`, when to use each HTTP status code, and how content negotiation is an "agreement" between client and server.

What Are Request and Response Handling?

Beyond the path/query string/headers, an HTTP request and response also have a **body** -- usually JSON, sometimes XML or another format depending on the API. `@RequestBody` reads this body, `@ResponseBody` / `ResponseEntity` writes it:

```
@PostMapping("/users")
public ResponseEntity<User> create(@RequestBody CreateUserRequest request) {
    // request is automatically populated from the request's JSON body
    User created = ...;
    return ResponseEntity.status(HttpStatus.CREATED).body(created);
    // created will be automatically written to the response's JSON body
}
```

Why Does It Exist?

Path variables and query parameters (as we saw in the previous lesson) are ideal for carrying individual, named values -- but impractical for complex, nested structures (an address, an order with multiple fields); you'd need a separate `@RequestParam` for every single field. The body is the way to carry all of that data as **one structured document**.

Similarly, returning a value as-is (200 OK, JSON) is enough most of the time, but a real API needs to set its status code, headers (`Location`, custom headers), and content type based on the situation -- `ResponseEntity` is what provides that control.

History

As mentioned in Spring MVC Fundamentals' "History" section, `@RequestBody` and `@ResponseBody` arrived in Spring 3.0 (2009) alongside `@PathVariable` -- all three served the same goal: making REST-style, JSON-based APIs first-class citizens next to Spring MVC's original view-oriented (HTML-returning) model. `ResponseEntity` was added in the same era -- a wrapper that can carry not just the body but also the status code and headers in a single object.

@RequestBody: Turning the Request Body into an Object

`@RequestBody` reads the **entire** body of the request and converts it into a Java object -- unlike `@RequestParam` / `@PathVariable`, which each target one named value, this targets the whole body:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;

// @RequestBody reads the ENTIRE HTTP request body and deserializes it into a Java
// object -- unlike @RequestParam/@PathVariable, which each read one named value, this
// reads the whole body at once.
@Controller
class UserCreationController {

    record CreateUserRequest(String name, String email) {
    }

    @PostMapping("/users")
    @ResponseBody
    public String create(@RequestBody CreateUserRequest request) {
        return "Created user: " + request.name() + " <" + request.email() + ">";
    }
}
```

The `{"name": "...", "email": "..."}` JSON in the request body is automatically mapped onto `CreateUserRequest`'s fields by name matching. We'll see who performs this conversion in "HttpMessageConverter: The Mechanism Behind @RequestBody/@ResponseBody".

HttpMessageConverter: The Mechanism Behind @RequestBody/@ResponseBody

`@RequestBody` / `@ResponseBody` don't do the JSON conversion themselves -- they delegate to an `HttpMessageConverter`; for JSON, that converter is the very Jackson `ObjectMapper` used directly here:

```
import com.fasterxml.jackson.databind.ObjectMapper;

// @RequestBody and @ResponseBody don't do the JSON conversion themselves -- they
// delegate to an HttpMessageConverter, and for JSON that converter is backed by
// exactly the Jackson ObjectMapper used directly here. spring-boot-starter-web
// auto-configures one of these and registers it as a bean; this is what it does
// under the hood on every request/response.
class HttpMessageConverterExample {

    record Product(String name, double price) {
    }

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();

        // What happens to an incoming @RequestBody:
        String requestJson = "{\"name\":\"Keyboard\",\"price\":49.9}";
        Product product = mapper.readValue(requestJson, Product.class);
        System.out.println(product);
        // Product[name=Keyboard, price=49.9]

        // What happens to an outgoing @ResponseBody:
        String responseJson = mapper.writeValueAsString(product);
        System.out.println(responseJson);
        // {"name":"Keyboard","price":49.9}
    }
}
```

`spring-boot-starter-web` auto-configures this converter and registers it as a bean -- another example of the auto-configuration mechanism from Spring Boot Auto-Configuration & Properties. In a real request, you never call `mapper.readValue(...)` yourself; DispatcherServlet calls it on your behalf -- what you see here is exactly what happens behind every request and response.

Deserializing Nested Objects and Lists

`@RequestBody` isn't limited to flat objects -- Jackson recursively deserializes nested objects and lists too, as long as there's a matching Java type at every level:

```
import com.fasterxml.jackson.databind.ObjectMapper;

import java.util.List;

// @RequestBody isn't limited to flat objects -- Jackson recursively deserializes
// nested objects and lists, as long as every level has a matching Java type.
class NestedObjectDeserializationExample {

    record Address(String city, String country) {
    }

    record OrderRequest(String customerName, Address shippingAddress, List<String> items) {
    }

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();

        String json = """
                {
                  "customerName": "Ayse",
                  "shippingAddress": { "city": "Istanbul", "country": "Turkey" },
                  "items": ["Keyboard", "Mouse"]
                }
                """;

        OrderRequest order = mapper.readValue(json, OrderRequest.class);
        System.out.println(order);
        // OrderRequest[customerName=Ayse, shippingAddress=Address[city=Istanbul,
country=Turkey], items=[Keyboard, Mouse]]
    }
}
```

The `shippingAddress` (an `Address` object) and `items` (a `List<String>`) inside `OrderRequest` are fully populated in a single `readValue(...)` call, with no manual conversion code -- Jackson maps the JSON's structure onto the Java type's structure step by step.

Missing or Extra Fields: How Jackson Behaves

A field missing from the JSON and an extra field present in the JSON but with no counterpart on the Java side lead to two very different behaviors in Jackson:

```
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException;

// Two very different failure modes: a field the JSON is MISSING is usually silently
// null (harmless, unless a lower-level lesson like Validation adds a rule against
// it); a field the JSON has EXTRA that Java doesn't know about is rejected outright,
// by Jackson's default configuration.
class UnknownFieldsToleranceExample {

    record CreateUserRequest(String name, String email) {
    }

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();

        String missingField = "{\"name\":\"Ayse\"}";
        CreateUserRequest withMissingField = mapper.readValue(missingField,
CreateUserRequest.class);
        System.out.println(withMissingField);
        // CreateUserRequest[name=Ayse, email=null]

        String extraField = "{\"name\":\"Ayse\",\"email\":\"ayse@example.com\",\"age\":30}";
        try {
            mapper.readValue(extraField, CreateUserRequest.class);
        } catch (UnrecognizedPropertyException e) {
            System.out.println("Rejected unknown field: " + e.getPropertyName());
            // Rejected unknown field: age
        }
    }
}
```

When `email` is missing, it's silently assigned `null`, no error at all. When an unknown field like `age` shows up, an `UnrecognizedPropertyException` is thrown -- Jackson's default setting is to **reject** fields it doesn't recognize. There's no built-in "required" check for missing fields -- that's what Bean Validation provides, the subject of the next lesson (Validation & Exception Handling).

ResponseEntity: Taking Full Control of the Response

Returning a plain object always sends `200 OK`. `ResponseEntity` gives full control over the status code alongside the body:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.Map;

// Returning a plain object always sends 200 OK. ResponseEntity gives full control
// over the status code (and, as we'll see next, headers) alongside the body.
@Controller
class ProductLookupController {
    private final Map<Long, String> products = Map.of(1L, "Keyboard");

    @GetMapping("/products/{id}")
    @ResponseBody
    public ResponseEntity<String> getProduct(@PathVariable Long id) {
        String name = products.get(id);
        if (name == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
        }
        return ResponseEntity.ok(name);
    }
}
```

When the product is found, `ResponseEntity.ok(name)` returns `200`; when it isn't, `ResponseEntity.status(HttpStatus.NOT_FOUND).build()` returns `404` -- the same method can produce two different status codes depending on the condition, a flexibility a plain return value can't offer.

Adding Headers with ResponseEntity

`ResponseEntity`'s builder can add headers along with the status code -- the most common example being `Location`, telling the client where a newly created resource now lives:

```
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;

import java.net.URI;

// ResponseEntity.BodyBuilder can also set headers -- the most common case being
// Location, telling the client where the resource it just created now lives.
@Controller
class ArticleCreationController {

    record CreateArticleRequest(String title) {
    }

    @PostMapping("/articles")
    @ResponseBody
    public ResponseEntity<Void> create(@RequestBody CreateArticleRequest request) {
        long newId = 42; // pretend this came from a real save operation
        URI location = URI.create("/articles/" + newId);

        return ResponseEntity.created(location)
                .header("X-Created-By", "learning-platform")
                .build();
    }
}
```

`ResponseEntity.created(location)` sets both the `201 Created` status and the `Location` header in a single line; `.header(...)` can add extra, custom headers too. The client can read the new resource's URL straight from the `Location` header and go there directly.

HTTP Status Codes: Which One, When

Status codes aren't chosen at random -- each one communicates a specific, standardized meaning to the client:

- **2xx**: the request was processed successfully (see "2xx Success Codes: 200, 201, 204")
- **4xx**: something is wrong with the request -- the client needs to fix something (see "4xx Client Errors: 400, 401, 403, 404, 409")
- **5xx**: the request was valid, but something went wrong on the server side (see "5xx Server Errors: 500")

Telling these three categories apart matters: retrying an unchanged request after a 4xx is pointless (the same error will happen again); after a 5xx -- since the request itself was valid -- retrying after some time can be reasonable.

2xx Success Codes: 200, 201, 204

The three most common success codes correspond to different scenarios:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.ResponseBody;

// The three 2xx codes that come up constantly: 200 (a normal successful read/update
// with a body), 201 (a new resource was created, usually with a Location header, see
// "Adding Headers with ResponseEntity"), 204 (successful, but there's nothing to send
// back).
@Controller
class NoteController {

    @GetMapping("/notes/1")
    @ResponseBody
    public ResponseEntity<String> get() {
        return ResponseEntity.status(HttpStatus.OK).body("Buy milk");
    }

    @PostMapping("/notes")
    @ResponseBody
    public ResponseEntity<String> create() {
        return ResponseEntity.status(HttpStatus.CREATED).body("Note created");
    }

    @DeleteMapping("/notes/1")
    @ResponseBody
    public ResponseEntity<Void> delete() {
        return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
    }
}

class StatusCode2xxExample {
    public static void main(String[] args) {
        NoteController controller = new NoteController();

        System.out.println(controller.get().getStatusCode());
        // 200 OK
        System.out.println(controller.create().getStatusCode());
        // 201 CREATED
        System.out.println(controller.delete().getStatusCode());
        // 204 NO_CONTENT
    }
}
```

`200 OK`: a normal successful read/update (with a body). `201 Created`: a new resource was created (usually together with the `Location` header we saw in "Adding Headers with ResponseEntity"). `204 No Content`: the operation succeeded but there's no body to send back -- the same code we saw in Mapping Annotations' "DELETE and Idempotency" section.

4xx Client Errors: 400, 401, 403, 404, 409

Five common 4xx codes, thrown with `ResponseStatusException` -- the same class this project's own `TopicController` uses:

```
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.server.ResponseStatusException;

import java.util.Map;

// ResponseStatusException -- the same class this project's own TopicController uses
// (see "This Project's Own Mappings" in an earlier lesson) -- is the simplest way to
// signal a 4xx from anywhere in a controller: throw it, and DispatcherServlet turns
// it into the right HTTP response, no manual ResponseEntity needed.
@Controller
class AccountController {
    private final Map<Long, String> accounts = Map.of(1L, "checking");
    private final Map<Long, String> owners = Map.of(1L, "ayse");

    record TransferRequest(Long fromAccountId, Double amount) {
    }

    @PostMapping("/transfers")
    @ResponseBody
    public String transfer(@RequestBody TransferRequest request) {
        if (request.amount() == null || request.amount() <= 0) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "amount must be
positive"); // 400
        }
        return "Transfer accepted";
    }

    @GetMapping("/accounts/{id}")
    @ResponseBody
    public String getAccount(@PathVariable Long id, String currentUser) {
        if (currentUser == null) {
            throw new ResponseStatusException(HttpStatus.UNAUTHORIZED, "login required");
// 401
        }
        if (!owners.getOrDefault(id, "").equals(currentUser)) {
            throw new ResponseStatusException(HttpStatus.FORBIDDEN, "not your account"); //
403
        }
        String account = accounts.get(id);
        if (account == null) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "account not found");
// 404
        }
```

```
        return account;
    }

    @DeleteMapping("/accounts/{id}")
    @ResponseBody
    public String closeAccount(@PathVariable Long id) {
        String account = accounts.get(id);
        if ("checking".equals(account)) {
            // Business rule violated: checking accounts with a balance can't be
            // closed. The request is well-formed, but conflicts with server state.
            throw new ResponseStatusException(HttpStatus.CONFLICT, "account has a
balance"); // 409
        }
        return "Account closed";
    }
}
```

`400 Bad Request`: the body/parameter is invalid (`amount` is negative). `401 Unauthorized`: the client hasn't authenticated at all. `403 Forbidden`: the client's identity is known, but they don't have permission for this resource -- the difference from 401 is "we know who you are, and you still don't have access." `404 Not Found`: the resource doesn't exist. `409 Conflict`: the request is well-formed, but conflicts with the current state on the server (like trying to close an account that has a balance).

5xx Server Errors: 500

Unlike 4xx, a `500` is usually **not** returned on purpose -- it's Spring's default response to an exception nobody caught:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

// Unlike a 4xx, a 500 usually isn't something you return on purpose -- it's Spring's
// DEFAULT response when a controller method throws an exception nobody handled.
@Controller
class ReportController {

    @GetMapping("/reports/summary")
    @ResponseBody
    public String summary() {
        int result = 1 / computeDivisor(); // bug: divisor can be 0, throws
ArithmeticException
        return "Result: " + result;
        // With no ResponseStatusException and no @ExceptionHandler (the next lesson,
        // Validation & Exception Handling, covers those) to catch it,
        // DispatcherServlet's default error handling turns the uncaught
        // ArithmeticException into a generic 500 Internal Server Error -- the
        // exception's details are logged server-side but never exposed to the client.
    }

    private int computeDivisor() {
        return 0;
    }
}
```

The `ArithmeticException` is caught by neither a `ResponseStatusException` nor (as we'll see in the next lesson) an `@ExceptionHandler` -- DispatcherServlet's default error handler steps in and returns a generic `500 Internal Server Error` to the client; the exception's details stay in the server logs only, never leaking to the client.

Content Negotiation: Choosing a Representation with Accept

Content negotiation is the client (via the `Accept` header) and the server (via the `produces` attribute) agreeing on **which representation** of the same resource to exchange:

```
import org.springframework.http.MediaType;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

// Content negotiation is the client (via the Accept header) and the server (via
// `produces`) agreeing on which REPRESENTATION of the same resource to exchange.
// Both mappings below serve the same underlying data, in different formats.
@Controller
class ProductRepresentationController {

    @GetMapping(path = "/products/1", produces = MediaType.APPLICATION_JSON_VALUE)
    @ResponseBody
    public String asJson() {
        return "{\"name\":\"Keyboard\",\"price\":49.9}";
    }

    @GetMapping(path = "/products/1", produces = MediaType.APPLICATION_XML_VALUE)
    @ResponseBody
    public String asXml() {
        return "<product><name>Keyboard</name><price>49.9</price></product>";
    }
    // A request with "Accept: application/json" matches asJson(); "Accept:
    // application/xml" matches asXml(). A request with "Accept: text/csv" -- a
    // representation neither method produces -- matches neither, and DispatcherServlet
    // responds with 406 Not Acceptable before either method is ever called.
}
```

The same path (`/products/1`) is defined twice, with two different `produces` values -- a client sending `Accept: application/json` is routed to `asJson()` , one sending `Accept: application/xml` to `asXml()` . This extends the `consumes` / `produces` section from Mapping Annotations -- there, `consumes` decided the type of the incoming request; here, `produces` + `Accept` decide the type of the outgoing response.

When an Unsupported Representation Is Requested: 406 Not Acceptable

If a client asks for a representation **no mapping produces** (like `Accept: text/csv` in the previous section's example), DispatcherServlet returns not `404` but `406 Not Acceptable` -- the path exists, just not in the requested representation. This is another dimension of the logic behind Mapping Annotations' "When an Unsupported HTTP Method Is Requested: 405 Method Not Allowed": 404 (no path), 405 (path exists, wrong method), 406 (path and method exist, wrong representation) -- each answers "something's missing" with a different, specific reason.

This Project's Own Responses: A Real Example

You can see the mechanisms from this lesson in `TopicController` 's own code -- since the project is currently a read-only HTML site, it still doesn't use `@RequestBody` , but it now

genuinely uses both `ResponseStatusException` (from "4xx Client Errors: 400, 401, 403, 404, 409") and `ResponseEntity` (from "Adding Headers with ResponseEntity"), in two different methods:

```
Topic topic = topicRepository.findBySlugWithCategoryAndCourse(slug)
        .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Konu
bulunamadı: " + slug));
// ...
return ResponseEntity.status(HttpStatus.MOVED_PERMANENTLY)
        .location(URI.create("/" + language.getCode() + "/topics/" + slug))
        .build();
```

The first returns `404` when a resource identified by a path variable (recalling the previous lesson's "Path Variable or Query Parameter? Which One, When" distinction) can't be found. The second is a real `301 Moved Permanently` with a `Location` header, built entirely by hand with `ResponseEntity` -- it's how the old `/topics/{slug}?lang=..` URLs (from before language moved into the URL path) keep working today, redirecting to their new address instead of breaking outright.

Best Practices

- **Use `201` + a `Location` header when you actually create a new resource, don't settle for just `200`** -- as shown in "Adding Headers with ResponseEntity", this saves the client from having to construct the new resource's address by hand.
- **Don't confuse 401 with 403** -- as shown in "4xx Client Errors: 400, 401, 403, 404, 409", one means authentication never happened at all, the other means the identity is known but permission is missing; this distinction lets the client take the right action (log in for 401, try a different account for 403).
- **Never return 500 on purpose** -- as shown in "5xx Server Errors: 500", this code means "something unexpected happened"; every expected error condition should be handled with a `ResponseStatusException` (or, as we'll see in the next lesson, an `@ExceptionHandler`).
- **Never blindly trust data that arrives via `@RequestBody`** -- as shown in "Missing or Extra Fields: How Jackson Behaves", Jackson only validates **shape** (is the JSON valid, do the types match); validating business rules (a positive quantity, a non-blank name) is up to you -- the next lesson introduces `@Valid`, which automates that validation.

Common Mistakes

1. Assuming `@RequestBody` reads a single field, like `@RequestParam`. `@RequestBody` converts the **entire** body into one object -- a method can't have more than one `@RequestBody` parameter, because the body can only be read once (see "@RequestBody: Turning the Request Body into an Object").

2. Assuming Jackson silently ignores unknown JSON fields. The default behavior is the opposite -- an extra field rejects the entire request (see "Missing or Extra Fields: How Jackson Behaves"). Client code written under this assumption can run into unexpected `400`s the moment the server adds a new field.

3. Returning `200` for every success without thinking, never using `201` / `204`. The distinction shown in "2xx Success Codes: 200, 201, 204" lets a client (especially an automated one) interpret the response correctly -- whether a creation request returns `200` or `201` can change the client's behavior.

4. Confusing a business rule violation (e.g., closing an account that has a balance) with `400 Bad Request`. If the request is entirely well-formed but conflicts with the current state on the server, the correct code is `409 Conflict` -- `400` is for cases where the request **itself** is malformed (see "4xx Client Errors: 400, 401, 403, 404, 409").

5. Ignoring the `Accept` header and always returning the same format (e.g., only JSON), then not understanding why a client expecting XML gets a `406`. As shown in "Content Negotiation: Choosing a Representation with Accept" and "When an Unsupported Representation Is Requested: 406 Not Acceptable", a `406` means none of that path's `produces` values matched the requested `Accept`.

Summary, Cheat Sheet, and Glossary

Request and response handling covers reading/writing an HTTP request's/response's body (`@RequestBody` / `ResponseEntity`), which status code to use when, and the client-server agreement over representations (content negotiation). Key points:

- `@RequestBody`: converts the entire request body into a Java object, via an `HttpMessageConverter` (Jackson `ObjectMapper` for JSON)
- `ResponseEntity`: carries the status code + headers + body in one object; builder methods include `.ok()`, `.status(...)`, `.created(uri)`, `.noContent()`
- Jackson **rejects unknown JSON fields by default**, and silently assigns `null` to missing ones
- 2xx: success (200 read/update, 201 creation, 204 success with no body)
- 4xx: client error (400 invalid request, 401 unauthenticated, 403 unauthorized, 404 not found, 409 conflict)
- 5xx: server error, usually unintentional (500, the default for uncaught exceptions)
- 406 Not Acceptable: the path exists but there's no `produces` matching `Accept`

Quick reference:

```
@PostMapping("/resource")
ResponseEntity<Void> create(@RequestBody CreateRequest request) {
    // ... validation, saving ...
    return ResponseEntity.created(URI.create("/resource/" + id)).build();  // 201
}

@GetMapping("/resource/{id}")
ResponseEntity<Resource> getOne(@PathVariable Long id) {
    Resource found = ...;
    return found != null
        ? ResponseEntity.ok(found)                                          // 200
        : ResponseEntity.status(HttpStatus.NOT_FOUND).build();              // 404
}

@DeleteMapping("/resource/{id}")
ResponseEntity<Void> delete(@PathVariable Long id) {
    // ...
    return ResponseEntity.noContent().build();                              // 204
}

// Throwing a 4xx from anywhere:
throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "reason");
throw new ResponseStatusException(HttpStatus.CONFLICT, "reason");
```

Glossary

`@RequestBody` — An annotation that deserializes an HTTP request's entire body into a Java object.

`ResponseEntity` — A wrapper class carrying the status code, headers, and body in a single object, giving full control over the response.

`HttpMessageConverter` — The component `@RequestBody` / `@ResponseBody` delegate the actual conversion between the body and a Java object to (Jackson `ObjectMapper` - based for JSON).

`ResponseStatusException` — A class that, thrown from anywhere in a controller method, makes DispatcherServlet return a specific HTTP status code.

Content negotiation — The client (`Accept` header) and server (`produces`) agreeing on which representation of the same resource to exchange.

404 Not Found — The HTTP status code returned when the requested resource (the path itself) can't be found.

406 Not Acceptable — The HTTP status code returned when the path exists but there is no `produces` representation matching the client's `Accept` header.

409 Conflict — The HTTP status code returned when a request is well-formed but conflicts with the current state on the server.

500 Internal Server Error — The default HTTP status code Spring returns for an exception caught nowhere.

Appendix: Mini Project — An Order Creation API

We bring every mechanism from this lesson together in a realistic order creation/retrieval API:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.server.ResponseStatusException;

import java.net.URI;
import java.util.LinkedHashMap;
import java.util.Map;

// An order-creation endpoint using every mechanism from this lesson: @RequestBody to
// read the order, manual validation (Bean Validation arrives in the next lesson) that
// throws a ResponseStatusException for bad input, and a ResponseEntity with a
// Location header and 201 Created for success.
@Controller
class OrderApiController {
    private final Map<Long, String> orders = new LinkedHashMap<>();
    private long nextId = 1;

    record CreateOrderRequest(String item, Integer quantity) {
    }

    @PostMapping("/api/orders")
    @ResponseBody
    public ResponseEntity<Void> create(@RequestBody CreateOrderRequest request) {
        if (request.item() == null || request.item().isBlank()) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "item is required");
        }
        if (request.quantity() == null || request.quantity() <= 0) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "quantity must be
positive");
        }

        long id = nextId++;
        orders.put(id, request.quantity() + "x " + request.item());

        return ResponseEntity.created(URI.create("/api/orders/" + id)).build();
    }

    @GetMapping("/api/orders/{id}")
    @ResponseBody
    public ResponseEntity<String> getOne(@PathVariable Long id) {
        String order = orders.get(id);
        return order != null ? ResponseEntity.ok(order) : ResponseEntity.status(HttpStatus.
NOT_FOUND).build();
    }
}
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.server.ResponseStatusException;

class OrderApiDemo {
    public static void main(String[] args) {
        OrderApiController controller = new OrderApiController();

        ResponseEntity<Void> created = controller.create(new OrderApiController.
CreateOrderRequest("Keyboard", 2));
        System.out.println(created.getStatusCode() + " Location=" + created.getHeaders().
getLocation());
        // 201 CREATED Location=/api/orders/1

        System.out.println(controller.getOne(1L).getBody());
        // 2x Keyboard

        try {
            controller.create(new OrderApiController.CreateOrderRequest("Mouse", 0));
        } catch (ResponseStatusException e) {
            System.out.println(e.getStatusCode() + ": " + e.getReason());
            // 400 BAD_REQUEST: quantity must be positive
        }
    }
}
```

`create(...)` performs the manual validation (checking `item` / `quantity`) mentioned in "Missing or Extra Fields: How Jackson Behaves", throwing a `400` via the `ResponseStatusException` from "4xx Client Errors: 400, 401, 403, 404, 409" when invalid; when valid, it returns `201` + `Location` following the pattern in "Adding Headers with ResponseEntity". `OrderApiDemo` exercises both the success and error paths by calling the methods directly, without a real DispatcherServlet.

Appendix: Mini Project — A Hand-Written HttpMessageConverter Chain Simulation

The final mini project combines "HttpMessageConverter: The Mechanism Behind @RequestBody/@ResponseBody" and "Content Negotiation: Choosing a Representation with Accept" into a single mechanism -- picking among multiple converters based on `Accept`:

```
import com.fasterxml.jackson.databind.ObjectMapper;

import java.util.Map;
import java.util.function.Function;

// A miniature model of Spring's HttpMessageConverter chain: several converters, each
// claiming a media type, picked based on what the request/response needs -- the same
// idea behind both @RequestBody deserialization ("HttpMessageConverter: The Mechanism
// Behind @RequestBody/@ResponseBody") and content negotiation ("Content Negotiation:
// Choosing a Representation with Accept").
class MessageConverterSimulation {

    record Product(String name, double price) {
    }

    private final ObjectMapper jsonMapper = new ObjectMapper();

    private final Map<String, Function<Product, String>> writers = Map.of(
            "application/json", this::toJson,
            "application/xml", this::toXml
    );

    String write(Product product, String acceptHeader) {
        Function<Product, String> writer = writers.get(acceptHeader);
        if (writer == null) {
            return "406 Not Acceptable: " + acceptHeader;
        }
        return writer.apply(product);
    }

    Product read(String json) throws Exception {
        return jsonMapper.readValue(json, Product.class);
    }

    private String toJson(Product product) {
        try {
            return jsonMapper.writeValueAsString(product);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private String toXml(Product product) {
        return "<product><name>" + product.name() + "</name><price>" + product.price() + "<
/price></product>";
    }
}
```

```
class MessageConverterDemo {
    public static void main(String[] args) throws Exception {
        MessageConverterSimulation converters = new MessageConverterSimulation();

        MessageConverterSimulation.Product product = converters.read("{\"name\":\"
Keyboard\",\"price\":49.9}");
        System.out.println(product);
        // Product[name=Keyboard, price=49.9]

        System.out.println(converters.write(product, "application/json"));
        // {"name":"Keyboard","price":49.9}
        System.out.println(converters.write(product, "application/xml"));
        // <product><name>Keyboard</name><price>49.9</price></product>
        System.out.println(converters.write(product, "text/csv"));
        // 406 Not Acceptable: text/csv
    }
}
```

The `writers` map is a small model of real Spring's `List<HttpMessageConverter<?>>` -- each one "claims" a media type. When `write(...)` can't find a converter matching `acceptHeader`, it produces the code from "When an Unsupported Representation Is Requested: 406 Not Acceptable" -- a hand-written version of the choice the real DispatcherServlet makes.

> Notice that `read(...)` uses the exact same `ObjectMapper` usage as `HttpMessageConverterExample`, and `write(...)`'s JSON branch is identical to that same example -- this mini project doesn't invent a new mechanism, it brings together the pieces we've seen throughout the lesson into a single "converter selection" flow.