Path Variables & Request Parameters

Reading data from a request's URL and headers with @PathVariable, @RequestParam, and @RequestHeader; the difference between a path variable and a query parameter.

Intermediate 40 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/path-variables-request-parameters

---

Path Variables and Request Parameters

In Mapping Annotations and HTTP Methods we used `@PathVariable` only to keep an example realistic, without going into detail. In this lesson we cover every way of reading data from a request's **URL** (both the path itself and the query string) and its **headers** -- and we keep the promise we made in Spring MVC Fundamentals' "HandlerMapping and HandlerAdapter: What Happens Inside DispatcherServlet?" section: in the final mini project, we build with our own hands exactly how a real `HandlerAdapter` reads and populates every parameter type from the right part of the request.

What Are URL Mapping Patterns?

The path portion of a URL can carry two kinds of information: **literal segments** (`/products`) and **variable segments** (marked with curly braces, like `{id}`). Together, these form a URL mapping pattern:

```
/users              -- literal, no variables
/users/{id}         -- one variable segment
/users/search       -- literal, doesn't collide with {id} (see the previous
                       lesson's "Combining @RequestMapping at the Class and
                       Method Level")
```

The way to read the value in a variable segment is `@PathVariable` ; the way to read values from the part **outside** the path, after the `?` (the query string), is `@RequestParam` -- we'll cover both in detail in this lesson.

Why Does It Exist?

Without path variables and request parameters, you'd need a separate mapping for every different `id` -- three separate `@GetMapping` s for `/users/1` , `/users/2` , `/users/3` . Instead, a

placeholder like `{id}` binds an **unlimited** number of URLs to a single mapping and a single method; the actual value is obtained as a parameter when the method is called. The same logic applies to the query string: instead of writing separate mappings for `?page=1`, `?page=2`, `?page=3`, the `page` parameter is read in one single method.

History

As mentioned in Spring MVC Fundamentals' "History" section, Spring 3.0 (2009) standardized REST-style endpoints with `@PathVariable` and `@RequestBody` / `@ResponseBody`. `@RequestParam` goes back even further, to Spring 2.5 (2007) -- the same release as `@RequestMapping` / `@Controller` -- answering the needs of form-based web applications (HTML forms sending a query string or form-encoded body via `GET` / `POST`). `@RequestHeader` was added in the same era.

@PathVariable: Reading a Value from the URL

The most basic usage: binding a `{placeholder}` in the path to a method parameter of the same name:

```
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;

// @PathVariable pulls a value straight out of the URL's own path -- the {id} segment
// in the mapping and the `id` parameter are linked by name.
@Controller
class ProductController {

    @GetMapping("/products/{id}")
    @ResponseBody
    public String getProduct(@PathVariable Long id) {
        return "Product #" + id;
    }
}
```

The `id` in `getProduct(Long id)` is bound to the `{id}` in the mapping by name matching -- Spring automatically converts the `String` value it read from the path into a `Long`; we'll see how that conversion works and what happens when it fails in "Type Conversion and Bad Values: 400 Bad Request".

Multiple Path Variables

A path can carry more than one variable segment -- each one binds to its own method parameter:

```
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;

// A path can carry more than one variable -- each {placeholder} becomes its own
// method parameter, matched by name.
@Controller
class OrderItemController {

    @GetMapping("/users/{userId}/orders/{orderId}")
    @ResponseBody
    public String getOrder(@PathVariable Long userId, @PathVariable Long orderId) {
        return "Order #" + orderId + " belonging to user #" + userId;
    }
}
```

`{userId}` and `{orderId}` bind to the `userId` and `orderId` parameters respectively -- because the binding is done by name, the parameters don't have to appear in the method signature in the same order they appear in the path (though keeping the same order is still good practice for readability).

Mapping a Path Variable's Name: The value Attribute

A method parameter's name doesn't have to match the `{placeholder}` exactly -- `@PathVariable`'s `value` attribute explicitly states which placeholder it binds to:

```
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;

// When the method parameter's name doesn't match the {placeholder}, or when the code
// is compiled without the -parameters flag (so parameter names aren't available at
// runtime), @PathVariable's `value` tells Spring explicitly which placeholder to bind.
@Controller
class ArticleController {

    @GetMapping("/articles/{articleSlug}")
    @ResponseBody
    public String getArticle(@PathVariable("articleSlug") String slug) {
        return "Article: " + slug;
    }
}
```

The `slug` parameter is explicitly bound to the `{articleSlug}` placeholder with `@PathVariable("articleSlug")`. This isn't just a naming preference -- if the code is compiled without the `-parameters` compiler flag, method parameters' actual names aren't available at runtime at all; in that case `value` becomes **mandatory**.

Path Variable or Query Parameter? Which One, When

The difference between a path variable and a query parameter goes deeper than syntax -- a path variable **identifies** a resource (the request is meaningless without it), a query parameter **filters/narrows** a request that's already valid on its own:

```
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.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

// A path variable IDENTIFIES a resource -- without it, there's no request to make.
// A query parameter FILTERS/MODIFIES a request that's already valid on its own.
@Controller
class ArticleListController {

    // No {id} here -- this endpoint is valid with zero query parameters too.
    @GetMapping("/articles")
    @ResponseBody
    public String list(@RequestParam(required = false) String category) {
        return category == null ? "All articles" : "Articles in category: " + category;
    }

    // {id} is required -- there is no "get one article" without knowing which one.
    @GetMapping("/articles/{id}")
    @ResponseBody
    public String getOne(@PathVariable Long id) {
        return "Article #" + id;
    }
}
```

`/articles/{id}` -- a "get one article" request means nothing without an `id`, so a path variable. `/articles?category=...` -- "list all articles" is a valid request even without `category`, it just narrows the result, so a query parameter. Getting this distinction right keeps URLs readable and cacheable.

@RequestParam: Reading a Value from the Query String

`@RequestParam` applies the same name-matching logic as `@PathVariable`, but to the query string instead of the path:

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

// @RequestParam reads a value from the query string -- ?page=2 becomes the `page`
// parameter, matched by name just like @PathVariable.
@Controller
class UserListController {

    @GetMapping("/users")
    @ResponseBody
    public String list(@RequestParam int page) {
        return "Showing page " + page;
    }
}
```

A `?page=2` request binds `2` (automatically converted to `int`) to the `page` parameter. Unlike `@PathVariable`, `@RequestParam` is **required by default** -- if `page` is never sent, the controller method is never called, and the client gets a `400 Bad Request`.

Required, Optional, and Default-Valued Parameters

The "required by default" behavior from the previous section can be changed with `required` and `defaultValue`:

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

// By default, @RequestParam is REQUIRED -- a missing query parameter is a 400 Bad
// Request, not a null value. `required = false` and `defaultValue` change that.
@Controller
class SearchController {

    @GetMapping("/search")
    @ResponseBody
    public String search(
            @RequestParam String query,
            @RequestParam(required = false) String sortBy,
            @RequestParam(defaultValue = "20") int limit) {
        return "Searching \"" + query + "\", sortBy=" + sortBy + ", limit=" + limit;
    }
}
```

`query` is required (a `400` without `?query=`); `sortBy` is optional thanks to `required = false` (`null` if omitted); `limit` is both optional and has a meaningful value instead of `null` when omitted, thanks to `defaultValue = "20"`. When `defaultValue` is given, there's no need to also specify `required` -- a parameter with a default value is already implicitly optional.

Multi-Valued Parameters: List and Array

A query string can carry the same key more than once -- binding that to a `List` (or an array) lets a single parameter carry multiple values:

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

import java.util.List;

// A query string can repeat the same key (?tag=java&tag=spring) -- binding that to a
// List lets a single parameter carry multiple values.
@Controller
class ArticleFilterController {

    @GetMapping("/articles/by-tag")
    @ResponseBody
    public String filterByTags(@RequestParam List<String> tag) {
        return "Filtering by tags: " + tag;
    }
}
```

A `?tag=java&tag=spring` request binds the list `["java", "spring"]` to the `tag` parameter. This is a natural extension of the filtering scenario we saw in "Path Variable or Query Parameter? Which One, When" -- ideal for multi-select filters like "show only items with one of these tags".

Capturing All Query Parameters: Map<String, String>

Sometimes you can't know the parameter names in advance -- binding `@RequestParam` to a `Map` captures **every** query parameter present on the request, whatever its name:

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

import java.util.Map;

// Sometimes you don't know the query parameter names in advance -- binding to a Map
// captures every query parameter present on the request, whatever its name.
@Controller
class FlexibleFilterController {

    @GetMapping("/reports")
    @ResponseBody
    public String report(@RequestParam Map<String, String> allParams) {
        return "Received filters: " + allParams;
    }
}
```

`allParams` collects a request with an unknown-in-advance number and set of parameters, like `?status=active&region=eu`, into a single `Map<String, String>`. This flexibility comes at a cost: you can't check at compile time which parameters exist, and type conversion (everything arrives as `String`) has to be done manually.

@RequestHeader: Reading HTTP Headers

`@RequestHeader` does for HTTP headers what `@RequestParam` does for the query string:

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

// @RequestHeader reads a value from the HTTP request headers, the same way
// @RequestParam reads from the query string -- required by default, with the same
// `required`/`defaultValue` options.
@Controller
class ClientInfoController {

    @GetMapping("/whoami")
    @ResponseBody
    public String whoAmI(
            @RequestHeader("User-Agent") String userAgent,
            @RequestHeader(value = "X-Request-Id", required = false) String requestId) {
        return "User-Agent: " + userAgent + ", X-Request-Id: " + requestId;
    }
}
```

`User-Agent` is required (every browser/client sends it anyway); `X-Request-Id` is left optional with `required = false` -- a custom header may not be present on every client. Because header names (like `"User-Agent"`) usually contain hyphens, `value` is almost always required here -- Java method parameter names can't contain hyphens.

Type Conversion and Bad Values: 400 Bad Request

`@PathVariable` / `@RequestParam` / `@RequestHeader` all arrive from the HTTP request as a raw `String` -- converting to a type like `Long`, `int`, or `boolean` is done by Spring's `ConversionService`:

```
import org.springframework.core.convert.ConversionException;
import org.springframework.core.convert.support.DefaultConversionService;

// Every @PathVariable/@RequestParam value arrives as a String -- Spring converts it to
// the declared parameter type (Long, int, boolean...) using the same kind of
// ConversionService machinery shown here directly. When conversion fails, real Spring
// MVC turns it into a 400 Bad Request before your controller method is ever called.
class TypeConversionErrorExample {
    public static void main(String[] args) {
        DefaultConversionService conversionService = new DefaultConversionService();

        Long id = conversionService.convert("42", Long.class);
        System.out.println("Converted: " + id);
        // Converted: 42

        try {
            conversionService.convert("abc", Long.class);
        } catch (ConversionException e) {
            System.out.println("Conversion failed, just like a real request to /products
/abc would fail");
            // Conversion failed, just like a real request to /products/abc would fail
        }
    }
}
```

`"42"` converts to `Long` without issue. `"abc"` can't be converted and throws a `ConversionException` -- in a real Spring MVC request, this is exactly why a request like `GET /products/abc` (see "@PathVariable: Reading a Value from the URL") results in a `400 Bad Request`: the conversion fails **before** the controller method is called, at the DispatcherServlet layer -- the method itself never runs.

This Project's Own Path Variable and Query Parameter: A Real Example

You can see the mechanisms from this lesson in the project's own `TopicController`, and its real code is a good example of the distinction actually changing over time:

```
@GetMapping("/{lang:en|tr}/topics/{slug}")
public String show(@PathVariable String lang, @PathVariable String slug, Model model) {
    ...
}

@GetMapping("/topics/{slug}")
public ResponseEntity<Void> legacyRedirect(@PathVariable String slug,
                                           @RequestParam(required = false) String lang) {
    ...
}
```

`slug` is a direct example of the distinction from "Path Variable or Query Parameter? Which One, When" in both methods -- a "show this topic" request has no meaning without it, so it's

always a path variable. `lang` is more interesting: in `show(...)` it's now **also** a path variable, because for SEO reasons every page needs a stable, crawlable URL per language (`/en/topics/{slug}` and `/tr/topics/{slug}` are two different, independently indexable pages, not one page with an optional modifier) -- that makes `lang` part of the resource's identity too, not a filter on top of it. The one place `lang` is still a genuinely optional `@RequestParam` is `legacyRedirect(...)`, which exists only to 301-redirect the site's old `/topics/{slug}?lang=..` URLs to the new path -- there, a request is still perfectly meaningful without `lang` (it just falls back to English), which is exactly the "optional filter/modifier" case the distinction describes. `HomeController` mirrors the same split across its two methods: `index(...)` (the `/{lang:en|tr}` mapping that actually renders a page) takes only `@PathVariable String lang`, while `root(...)` (the bare `/` negotiator that 302-redirects to `/en` or `/tr`) takes the same optional `@RequestParam(required = false) String lang` as `legacyRedirect(...)`, for the same reason -- it's a courtesy for old `/?lang=..` bookmarks, not something the request strictly needs.

Best Practices

- **If a value is part of a resource's identity, make it a path variable; if it's an optional filter/modifier, make it a query parameter** -- applying the distinction from "Path Variable or Query Parameter? Which One, When" consistently keeps an API's URLs readable and cacheable.
- **Use `required = false` or `defaultValue` on every parameter that's genuinely optional** -- otherwise, as shown in "Required, Optional, and Default-Valued Parameters", every parameter a client forgets to send turns into a `400`.
- **Only use `@RequestParam Map<String, String>` for parameters that are genuinely dynamic/unknown in advance** -- declaring known parameters individually (see "Required, Optional, and Default-Valued Parameters") gives you type safety and readability.
- **Prefer keeping path variable names identical to their `{placeholder}`, and always write `value` explicitly when they differ** -- as shown in "Mapping a Path Variable's Name: The value Attribute", relying on the `-parameters` flag is a fragile assumption.

Common Mistakes

1. Assuming `@RequestParam` is optional by default, like `@PathVariable`. The opposite is true -- `@RequestParam` is **required** by default; being optional requires explicit `required = false` or `defaultValue` (see "@RequestParam: Reading a Value from the Query String").

2. Using a path variable and a query parameter interchangeably (e.g., writing `/articles?id=5` instead of `/articles/5`). Both may technically work, but this violates the semantic distinction in "Path Variable or Query Parameter? Which One, When" -- a value that identifies a resource belongs in the path.

3. Mistaking a type conversion error (`400 Bad Request`) for an application bug and trying to add a try/catch inside the controller. The conversion happens at the DispatcherServlet layer, before the controller method is ever called -- a try/catch inside the method will never catch this error (see "Type Conversion and Bad Values: 400 Bad Request").

4. Writing header names like a Java method parameter name and forgetting to specify `value` (e.g., `@RequestHeader String userAgent` instead of `"User-Agent"`). Hyphens in header names aren't valid in Java identifiers -- without `value`, Spring looks for a header literally named `userAgent`, doesn't find it, and (being required by default) returns `400` (see "@RequestHeader: Reading HTTP Headers").

5. Expecting `@RequestParam List<String>` to work with a client sending a single comma-separated parameter like `?tag=java,spring`. Spring's `List` binding expects the same key to be **repeated** (`?tag=java&tag=spring`), not a single value split by commas (see "Multi-Valued Parameters: List and Array").

Summary, Cheat Sheet, and Glossary

Path variables and request parameters are the annotation-based way of reading data from an HTTP request's URL (path and query string) and headers. Key points:

- `@PathVariable`: reads a `{placeholder}` from the path; used for values that identify a resource
- `@RequestParam`: reads a value from the query string; required by default (can be made optional with `required = false` / `defaultValue`)
- `@RequestHeader`: reads an HTTP header; follows the same required/optional rules
- `List` / `Map` binding: `@RequestParam List<String>` collects repeated keys, `@RequestParam Map<String, String>` collects an unknown number of parameters
- Type conversion happens via `ConversionService`, **before** the controller method is called -- failure means `400 Bad Request`
- Path variable = identifies a resource (required); query parameter = filters a request (usually optional)

Quick reference:

```
@GetMapping("/users/{id}")
String getOne(@PathVariable Long id) { ... }                    // path variable

@GetMapping("/articles/{articleSlug}")
String getArticle(@PathVariable("articleSlug") String slug) { ... }  // name mapping

@GetMapping("/search")
String search(
    @RequestParam String query,                                  // required
    @RequestParam(required = false) String sortBy,               // optional
    @RequestParam(defaultValue = "20") int limit,                // default value
    @RequestParam(required = false) List<String> tag,            // multi-valued
    @RequestParam Map<String, String> allParams,                 // all parameters
    @RequestHeader("User-Agent") String userAgent                // header
) { ... }
```

Glossary

Path variable — A value read from a `{placeholder}` segment of a URL path, identifying a resource.

Query parameter — A value read from the part of a URL after the `?` (the query string), filtering or modifying a request.

`@PathVariable` — An annotation that binds a method parameter to a `{placeholder}` in the path.

`@RequestParam` — An annotation that binds a method parameter to a value in the query string (or form body); required by default.

`@RequestHeader` — An annotation that binds a method parameter to an HTTP header value.

`ConversionService` — The Spring component that converts a `String` to the target Java type (Long, int, boolean...); used by `@PathVariable` / `@RequestParam` / `@RequestHeader` alike.

400 Bad Request — The HTTP status code returned when a required parameter is missing or type conversion fails.

Appendix: Mini Project — A Catalog Search API

We bring every mechanism from this lesson together in one realistic search endpoint:

```
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.RequestHeader;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.List;

// A realistic search endpoint over a single category (path variable, identifies the
// resource collection), narrowed by optional query parameters (filters) and aware of
// the requesting client (header) -- every mechanism from this lesson, together.
@Controller
class CatalogSearchController {

    @GetMapping("/catalog/{category}/search")
    @ResponseBody
    public String search(
            @PathVariable String category,
            @RequestParam String query,
            @RequestParam(required = false) List<String> tag,
            @RequestParam(defaultValue = "10") int limit,
            @RequestHeader(value = "Accept-Language", required = false) String language) {
        return "Searching \"" + query + "\" in category=" + category
                + ", tags=" + tag + ", limit=" + limit + ", language=" + language;
    }
}
```

```
import java.util.List;

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

        System.out.println(controller.search("books", "spring", List.of("java", "web"), 10,
"en"));
        // Searching "spring" in category=books, tags=[java, web], limit=10, language=en

        System.out.println(controller.search("electronics", "headphones", null, 10, null));
        // Searching "headphones" in category=electronics, tags=null, limit=10,
language=null
    }
}
```

`category` follows the distinction from "Path Variable or Query Parameter? Which One, When" and is a path variable (a search request has no context without a category); `query` / `tag` / `limit` are query parameters (the request itself is already valid within the same category, just narrowed differently). The `Accept-Language` header carries the client's language preference -- much like this project's own `lang` query parameter, but through HTTP's standard mechanism.

Appendix: Mini Project — A Hand-Written Argument Resolver Simulation

The final mini project keeps the promise from Spring MVC Fundamentals: we build by hand exactly how a real `HandlerAdapter` fills in each of a method's parameters based on its annotation, from the right part of the request:

```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestParam;

import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
import java.util.Map;

// Spring MVC Fundamentals' HandlerAdapter simulations could only invoke parameterless
// methods. This one does what a real HandlerAdapter's argument resolvers do: look at
// each parameter's annotation, pull the matching value out of the request, and pass it
// along when invoking the method.
class GreetingHandler {
    public String greet(@PathVariable("name") String name,
                        @RequestParam(defaultValue = "en") String lang,
                        @RequestHeader(value = "X-Client", required = false) String
client) {
        return "Hello " + name + " (lang=" + lang + ", client=" + client + ")";
    }
}

class RequestBinderSimulation {

    static Object invoke(Object handler, Method method,
                         Map<String, String> pathVariables,
                         Map<String, String> queryParams,
                         Map<String, String> headers) throws Exception {
        Parameter[] parameters = method.getParameters();
        Object[] args = new Object[parameters.length];

        for (int i = 0; i < parameters.length; i++) {
            Parameter parameter = parameters[i];

            PathVariable pathVar = parameter.getAnnotation(PathVariable.class);
            if (pathVar != null) {
                args[i] = pathVariables.get(pathVar.value());
                continue;
            }

            RequestParam requestParam = parameter.getAnnotation(RequestParam.class);
            if (requestParam != null) {
                String value = queryParams.get(parameter.getName());
                args[i] = value != null ? value : requestParam.defaultValue();
                continue;
            }

            RequestHeader requestHeader = parameter.getAnnotation(RequestHeader.class);
            if (requestHeader != null) {
                args[i] = headers.get(requestHeader.value());
            }
        }

        return method.invoke(handler, args);
    }
}
```

```
import java.lang.reflect.Method;
import java.util.Map;

class RequestBinderDemo {
    public static void main(String[] args) throws Exception {
        GreetingHandler handler = new GreetingHandler();
        Method method = GreetingHandler.class.getMethod("greet", String.class, String.
class, String.class);

        Object result = RequestBinderSimulation.invoke(
                handler,
                method,
                Map.of("name", "Ayse"),
                Map.of(),
                Map.of("X-Client", "web"));

        System.out.println(result);
        // Hello Ayse (lang=en, client=web)
    }
}
```

`RequestBinderSimulation.invoke(...)` walks every parameter of the `greet(...)` method via reflection -- if it's marked `@PathVariable`, it reads from the `pathVariables` map; if `@RequestParam`, from `queryParams` (or `defaultValue()` if missing); if `@RequestHeader`, from `headers` -- then calls the method with those values. Aside from the `ConversionService` step covered in "Type Conversion and Bad Values: 400 Bad Request", this is a full model of what real Spring does behind the scenes on every request.

> `RequestBinderSimulation` finds which query parameter `@RequestParam` binds to using `parameter.getName()` (Java's own reflection API) -- this is only reliable when the code is compiled with the `-parameters` compiler flag; otherwise parameter names come back as generic placeholders like `arg0`, `arg1`. As shown in "Mapping a Path Variable's Name: The value Attribute", real code should always write `value` explicitly instead of relying on this ambiguity.