Mapping Annotations & HTTP Methods

@RequestMapping and its five shortcuts (@GetMapping/@PostMapping/@PutMapping /@PatchMapping/@DeleteMapping), consumes/produces, and the safe/idempotent semantics of HTTP methods.

Intermediate 35 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/mapping-annotations-http-methods

---

Mapping Annotations and HTTP Methods

In Spring MVC Fundamentals we only saw `@GetMapping` at a glance -- a single annotation binding a path to a method. In this lesson we go into the whole family of mapping annotations (`@RequestMapping` and its five shortcuts), which HTTP method each corresponds to, and the semantic rules HTTP methods themselves carry (safe, idempotent). In "Same Path, Different HTTP Methods" we'll also close the gap left by the warning in the Fundamentals lesson's `RequestRouterSimulation` mini project -- disambiguating by HTTP method.

What Are Mapping Annotations?

Mapping annotations declare **which HTTP request** a controller method responds to -- a path (`"/users"`), an optional HTTP method, and optionally other conditions (content type, headers):

```
@Controller
class UserController {
    @GetMapping("/users")       // path: /users, HTTP method: GET
    public String list() { ... }
}
```

This is exactly the information read by the `HandlerMapping` we saw in Spring MVC Fundamentals -- the mechanism our `buildHandlerMapping` simulation in "HandlerMapping and HandlerAdapter: What Happens Inside DispatcherServlet?" stood in for.

Why Does It Exist?

Without mapping annotations, DispatcherServlet would need this information -- "a request to this path, with this HTTP method, goes to this method" -- from somewhere else (XML, a hand-written routing table). That's exactly what the XML-based `<bean>` mappings from 2004 mentioned in Spring MVC Fundamentals' "History" section did. Annotations keep this information **on the method itself**, next to the code -- adding a new endpoint is just writing a new method and annotating it.

History

Spring MVC Fundamentals' "History" section already covered the general timeline for this family: `@RequestMapping` shipped with Spring 2.5 (2007); the shortcuts like `@GetMapping` arrived much later, with Spring 4.3 (2016). For the nine years in between, developers had to write `@RequestMapping(method = RequestMethod.GET)` -- repetitive, and easy to forget the `method` attribute on (which, when forgotten, makes the mapping accept **every** HTTP method, hiding the error we'll see in "When an Unsupported HTTP Method Is Requested: 405 Method Not Allowed"). Spring 4.3 removed that repetition by adding five shortcuts (`@GetMapping` , `@PostMapping` , `@PutMapping` , `@PatchMapping` , `@DeleteMapping`) for the five HTTP methods.

@RequestMapping: The Base Mapping Annotation

The root of the family tree is `@RequestMapping` -- its `method` attribute can match any HTTP method (or, left unset, all of them):

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

// @RequestMapping is the original, most general mapping annotation -- it can match
// any HTTP method (or several at once) via its `method` attribute. Every shortcut
// annotation we'll see next (@GetMapping, @PostMapping...) is built on top of this one.
@Controller
class RawRequestMappingController {

    @RequestMapping(path = "/ping", method = RequestMethod.GET)
    @ResponseBody
    public String ping() {
        return "pong";
    }

    // Without a `method`, @RequestMapping matches EVERY HTTP method on this path --
    // rarely what you want, but useful to know it's the default.
    @RequestMapping(path = "/any-method")
    @ResponseBody
    public String anyMethod() {
        return "matched regardless of HTTP method";
    }
}
```

Without `method`, `@RequestMapping` accepts **every** HTTP method on that path -- `anyMethod()` responds to `GET`, `POST`, and `DELETE` alike. This is rarely the behavior you want; as we'll see in "Mapping HTTP Methods to CRUD Operations", every HTTP method carries its own meaning, and leaving that ambiguous is usually a design mistake.

@GetMapping, @PostMapping, and the Other Shortcuts

The five shortcut annotations are meta-annotations built on top of `@RequestMapping` -- each one pre-fills the `method` attribute for you:

```
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.PatchMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.ResponseBody;

// Each shortcut is a meta-annotation: @GetMapping is exactly
// @RequestMapping(method = RequestMethod.GET), just shorter and more readable at a
// glance. A typical resource controller uses one of each, one per operation.
@Controller
@ResponseBody
class UserShortcutController {

    @GetMapping("/users")
    public String listUsers() {
        return "list of users";
    }

    @PostMapping("/users")
    public String createUser() {
        return "user created";
    }

    @PutMapping("/users/1")
    public String replaceUser() {
        return "user replaced";
    }

    @PatchMapping("/users/1")
    public String updateUser() {
        return "user partially updated";
    }

    @DeleteMapping("/users/1")
    public String deleteUser() {
        return "user deleted";
    }
}
```

`@GetMapping("/users")` is exactly the same as `@RequestMapping(path = "/users", method = RequestMethod.GET)` -- just shorter, and it makes the intent obvious at a glance. A typical resource controller carries one of each of these five -- list, create, full update, partial update, delete.

Combining @RequestMapping at the Class and Method Level

`@RequestMapping` at the class level defines a **common prefix** -- every method inside the class defines its own path as a continuation of that prefix:

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

// @RequestMapping at the class level sets a COMMON PREFIX for every method inside --
// exactly the pattern this project's own TopicController used to follow
// (@RequestMapping("/topics") on the class, @GetMapping("/{slug}") on the method),
// before it later grew a second mapping shape that no longer shared that prefix
// (see "This Project's Own Mappings: A Real Example" in the parent lesson).
// @PathVariable is used here just to keep the example realistic; we'll cover it in
// full in the next lesson (Path Variables & Request Parameters).
@Controller
@RequestMapping("/users")
class UserController {

    @GetMapping
    @ResponseBody
    public String list() {
        return "GET /users";
    }

    @GetMapping("/{id}")
    @ResponseBody
    public String getOne(@PathVariable Long id) {
        return "GET /users/" + id;
    }

    @GetMapping("/search")
    @ResponseBody
    public String search() {
        return "GET /users/search";
    }
}
```

This is exactly the pattern this project's own `TopicController` uses -- we're keeping the promise made in Spring MVC Fundamentals' "This Project's Own Controllers: A Real Spring MVC Example" section; we'll look at it in detail in "This Project's Own Mappings: A Real Example". The `search()` method's path (`/users/search`) doesn't collide with `getOne()` 's path-variable path (`/users/{id}`) -- Spring's path matching always treats **literal segments as more specific** than variable segments, regardless of declaration order.

Specifying Content Type: consumes and produces

A mapping can be narrowed not just by path and HTTP method, but also by **which content type it accepts/produces**:

```
import org.springframework.http.MediaType;
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;

// consumes/produces narrow a mapping to specific Content-Type/Accept headers -- the
// SAME path and HTTP method can be mapped twice, once per representation. @RequestBody
// is used here just to keep the example realistic; it's covered in full in a later
// lesson (Request & Response Handling).
@Controller
class ContentNegotiatingController {

    @PostMapping(path = "/orders", consumes = MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
    @ResponseBody
    public String createFromJson(@RequestBody String body) {
        return "{\"status\":\"created from JSON\"}";
    }

    @PostMapping(path = "/orders", consumes = MediaType.APPLICATION_XML_VALUE, produces =
MediaType.APPLICATION_XML_VALUE)
    @ResponseBody
    public String createFromXml(@RequestBody String body) {
        return "<status>created from XML</status>";
    }
}
```

The same path (`/orders`) and the same HTTP method (`POST`) are defined **twice** here -- they don't collide because `consumes` / `produces` let DispatcherServlet decide which one applies based on the request's `Content-Type` / `Accept` headers. `@RequestBody` is used here just to keep the example realistic; we'll cover it in full in a later lesson (Request & Response Handling).

HTTP Methods: The Safe and Idempotent Concepts

The HTTP specification attributes two important properties to every method: **safe** (must not change server state) and **idempotent** (calling it once or a hundred times must leave the same result). GET is **required** to be both:

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

// "Safe" means an HTTP method must not change server state; "idempotent" means
// calling it once or a hundred times leaves the server in the same state as calling
// it exactly once. GET is required to be both; POST is neither.
@Controller
class ViewCounterController {
    private int views = 0;

    @GetMapping("/article")
    @ResponseBody
    public String viewArticle() {
        // Safe: reading the article never changes `views`.
        return "Article content (viewed " + views + " times so far)";
    }

    @PostMapping("/article/views")
    @ResponseBody
    public String recordView() {
        // Not safe, not idempotent: every call increments the counter further.
        views++;
        return "Recorded. Total views: " + views;
    }
}

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

        System.out.println(controller.viewArticle());
        // Article content (viewed 0 times so far)
        System.out.println(controller.viewArticle());
        // Article content (viewed 0 times so far) -- safe: calling GET changed nothing

        System.out.println(controller.recordView());
        // Recorded. Total views: 1
        System.out.println(controller.recordView());
        // Recorded. Total views: 2 -- not idempotent: state changed again
    }
}
```

`viewArticle()` (GET) never changes `views`, no matter how many times it's called -- safe. `recordView()` (POST) changes state on every call -- neither safe nor idempotent. This distinction is our reference point when we look at each remaining method individually in "PUT vs. PATCH: Full Replacement vs. Partial Update" and "DELETE and Idempotency".

Same Path, Different HTTP Methods

Recall the warning at the end of Spring MVC Fundamentals' final mini project: `RequestRouterSimulation` only looked at the path -- it couldn't distinguish requests to the same path made with different HTTP methods. We fix that now:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMethod;

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

// The Fundamentals lesson's FrontControllerSimulationExample only kept a
// path -> method registry -- it couldn't tell a GET handler apart from a POST handler
// on the same path. This version adds the HTTP method into the registry key, exactly
// what real Spring's HandlerMapping does.
class ArticleHandlers {
    @GetMapping("/article")
    public String view() {
        return "Viewing article";
    }

    @PostMapping("/article")
    public String publish() {
        return "Publishing article";
    }
}

class HttpMethodDisambiguationExample {

    private record RouteKey(String path, RequestMethod method) {
    }

    static Map<RouteKey, Method> buildRegistry(Object handler) {
        Map<RouteKey, Method> registry = new HashMap<>();
        for (Method method : handler.getClass().getDeclaredMethods()) {
            GetMapping get = method.getAnnotation(GetMapping.class);
            if (get != null) {
                registry.put(new RouteKey(get.value()[0], RequestMethod.GET), method);
            }
            PostMapping post = method.getAnnotation(PostMapping.class);
            if (post != null) {
                registry.put(new RouteKey(post.value()[0], RequestMethod.POST), method);
            }
        }
        return registry;
    }

    static String dispatch(String path, RequestMethod httpMethod, Object handler,
Map<RouteKey, Method> registry) throws Exception {
        Method method = registry.get(new RouteKey(path, httpMethod));
        if (method == null) {
            return "405 Method Not Allowed: " + httpMethod + " " + path;
```

```
        }
        return (String) method.invoke(handler);
    }

    public static void main(String[] args) throws Exception {
        ArticleHandlers handler = new ArticleHandlers();
        Map<RouteKey, Method> registry = buildRegistry(handler);

        System.out.println(dispatch("/article", RequestMethod.GET, handler, registry));
        // Viewing article
        System.out.println(dispatch("/article", RequestMethod.POST, handler, registry));
        // Publishing article
        System.out.println(dispatch("/article", RequestMethod.DELETE, handler, registry));
        // 405 Method Not Allowed: DELETE /article
    }
}
```

`RouteKey` now uses the **pair** `(path, method)` as its key, not just the path -- a `GET` to `/article` goes to `view()`; a `POST` to the same path goes to `publish()`. When there's no matching `(path, method)` pair (like `DELETE /article`), what we return is exactly what real Spring would too: **405 Method Not Allowed** -- not 404, because the path itself exists, just not for that HTTP method.

PUT vs. PATCH: Full Replacement vs. Partial Update

Both mean "update", but with different contracts: `PUT` replaces the **entire** resource with the new representation (fields not sent are lost); `PATCH` updates only the fields that were sent:

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

import java.util.LinkedHashMap;
import java.util.Map;

// PUT replaces the ENTIRE resource -- every field must be supplied, missing fields
// are lost. PATCH updates only the fields that were actually sent, leaving the rest
// untouched.
@Controller
class UserProfileController {
    private final Map<String, String> profile = new LinkedHashMap<>();

    UserProfileController() {
        profile.put("name", "Ayse");
        profile.put("city", "Istanbul");
    }

    @PutMapping("/profile")
    @ResponseBody
    public Map<String, String> replace(@RequestBody Map<String, String> newProfile) {
        profile.clear();
        profile.putAll(newProfile); // anything not in newProfile is gone
        return new LinkedHashMap<>(profile); // a copy that keeps insertion order for
display
    }

    @PatchMapping("/profile")
    @ResponseBody
    public Map<String, String> update(@RequestBody Map<String, String> changes) {
        profile.putAll(changes); // only overwrites the given keys
        return new LinkedHashMap<>(profile); // a copy that keeps insertion order for
display
    }
}

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

        System.out.println(controller.update(Map.of("city", "Ankara")));
        // {name=Ayse, city=Ankara} -- PATCH: only "city" changed, "name" untouched

        System.out.println(controller.replace(Map.of("city", "Izmir")));
        // {city=Izmir} -- PUT: "name" is GONE, it wasn't in the replacement body
    }
}
```

`update()` (PATCH) only changes `city`, leaving `name` untouched. `replace()` (PUT) calls `profile.clear()` first, wiping everything, then puts back only the fields that were sent --

`name` is gone entirely because it wasn't sent. This confusion is one of the most common mistakes in API design: a client expecting PATCH semantics accidentally calling PUT and wiping out other fields.

DELETE and Idempotency

Per the definition in "HTTP Methods: The Safe and Idempotent Concepts", `DELETE` must be idempotent -- but what that actually means is subtler than it looks at first:

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

import java.util.HashMap;
import java.util.Map;

// DELETE is idempotent in the sense that matters: no matter how many times you call
// it, the END STATE is the same (the resource is gone) -- even though the HTTP status
// code of the second call differs from the first.
@Controller
class BookDeletionController {
    private final Map<Long, String> books = new HashMap<>(Map.of(1L, "Effective Java"));

    @DeleteMapping("/books/{id}")
    @ResponseBody
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        if (books.remove(id) != null) {
            return ResponseEntity.noContent().build(); // 204: it was there, now it's gone
        }
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build(); // 404: already gone
    }
}

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

        System.out.println(controller.delete(1L).getStatusCode());
        // 204 NO_CONTENT
        System.out.println(controller.delete(1L).getStatusCode());
        // 404 NOT_FOUND -- different status, but the end state (book 1 is gone) is
        // identical after either call, which is exactly what idempotency means.
    }
}
```

The first `delete(1L)` call returns `204 No Content` (the book was actually deleted); the second returns `404 Not Found` (the book is already gone) -- **two different HTTP status codes**. It's still idempotent, because idempotency is about the **final state on the server** staying the same, not the status code -- after either call, book 1 does not exist in the data store.

Mapping HTTP Methods to CRUD Operations

The five HTTP methods we've seen map to CRUD (Create/Read/Update/Delete) operations as follows:

- **GET** → Read (safe + idempotent) -- the `viewArticle()` we saw in "HTTP Methods: The Safe and Idempotent Concepts"
- **POST** → Create (neither safe nor idempotent) -- every call creates a new resource or otherwise changes state
- **PUT** → Update, full replacement (idempotent, not safe) -- sending the same `PUT` request twice leaves the resource in the same final state as sending it once
- **PATCH** → Update, partial (usually treated as idempotent, but the HTTP spec doesn't guarantee it -- a PATCH that means "increment this field by 1" isn't idempotent)
- **DELETE** → Delete (idempotent, not safe) -- as we saw in "DELETE and Idempotency", the status code can change but the final state stays the same

When an Unsupported HTTP Method Is Requested: 405 Method Not Allowed

The `"405 Method Not Allowed"` message our simulation produces in "Same Path, Different HTTP Methods" isn't made up -- it's exactly what the real DispatcherServlet does: if a path has **at least one** mapping but none matches the requested HTTP method, it returns not 404 (path doesn't exist) but **405** (path exists, just not for this method). This distinction matters -- a client that gets a 405 can tell it had the right path but the wrong HTTP method; with a 404 it loses that information.

This Project's Own Mappings: A Real Example

You can see the mechanisms from this lesson in the project's own source code. `HomeController` carries no class-level `@RequestMapping` -- it has two endpoints (`@GetMapping("/")` and `@GetMapping("/{lang:en|tr}")`) that don't share a common literal prefix, so there'd be nothing to factor out into a shared class-level path anyway. `TopicController` actually used to be a textbook case of "Combining @RequestMapping at the Class and Method Level" -- `@RequestMapping("/topics")` at the class level, `@GetMapping("/{slug}")` at the method level -- until it grew a second mapping shape (a legacy-URL redirect, `/topics/{slug}` , alongside the real `/{lang:en|tr}/topics/{slug}`) that no longer shares a single class-level prefix with the first. At that point the class-level `@RequestMapping` was dropped and each method declares its own full path instead -- a real, small example of when a shared prefix stops paying for itself. Both controllers respond only to `GET` requests -- since this project is currently a read-only content site, `POST` / `PUT` / `PATCH` / `DELETE` are never used; later lessons in this category (Request & Response Handling, REST API Design) will work through a JSON API scenario that needs these other methods.

Best Practices

- **Always use the most specific shortcut; reach for bare `@RequestMapping` only when you genuinely need to accept more than one HTTP method** -- a `@RequestMapping` left without `method`, as mentioned in "History", silently accepts every method, which is usually not what you want.
- **Stay true to what each HTTP method means: don't mutate state on GET, do stay idempotent on DELETE** -- an API that violates the rules in "HTTP Methods: The Safe and Idempotent Concepts" breaks assumptions built into HTTP infrastructure like caching and retries.
- **Don't use PUT and PATCH interchangeably** -- as shown in "PUT vs. PATCH: Full Replacement vs. Partial Update", picking the wrong one can silently lose data.
- **Use a class-level `@RequestMapping` for endpoints that share a common prefix** -- the way this project's own `TopicController` does (see "This Project's Own Mappings: A Real Example"), instead of repeating the prefix on every method.

Common Mistakes

1. Forgetting to set `method` on `@RequestMapping` and assuming the mapping only responds to the method you had in mind. Without `method`, **every** HTTP method is accepted -- this can accidentally let a `DELETE` request reach an endpoint meant to be "read-only" (see "@RequestMapping: The Base Mapping Annotation").

2. Assuming a literal path like `/users/search` will collide with `/users/{id}`, and trying to "fix" it by reordering the declarations. Spring always treats literal segments as more specific than variable segments -- declaration order doesn't matter at all (see "Combining @RequestMapping at the Class and Method Level").

3. Writing a GET endpoint that mutates data (like a clickable delete link meant to be "easy to test" from a browser). This violates the rule that GET must be safe -- a cache, a bot, or a browser's link-prefetching feature can trigger that GET request unexpectedly (see "HTTP Methods: The Safe and Idempotent Concepts").

4. Expecting PUT semantics when sending a PATCH request (assuming omitted fields are wiped rather than preserved), or the reverse. The two contracts are deliberately different -- which one you call determines the fate of the fields you didn't send (see "PUT vs. PATCH: Full Replacement vs. Partial Update").

5. Misreading DELETE's idempotency as "the second call returns the same status code too". Idempotency is about the **final state on the server** staying the same, not the status code -- the first call can return 204, the second 404, and both are still idempotent (see "DELETE and Idempotency").

6. Expecting a 404 for a request made with an unsupported HTTP method. If the path genuinely exists but no mapping matches that HTTP method, the correct response is 405 -- 404 is reserved for when the path itself can't be found at all (see "When an Unsupported HTTP Method Is Requested: 405 Method Not Allowed").

Summary, Cheat Sheet, and Glossary

Mapping annotations bind a controller method to a path + HTTP method (+ optional content type) combination; every HTTP method carries its own safe/idempotent rules. Key points:

- `@RequestMapping`: the base annotation; accepts every HTTP method if `method` isn't set
- `@GetMapping` / `@PostMapping` / `@PutMapping` / `@PatchMapping` / `@DeleteMapping`: shortcuts for the five HTTP methods, meta-annotations of `@RequestMapping(method=...)`
- Class-level `@RequestMapping`: a shared path prefix, combined with method-level paths
- `consumes` / `produces`: narrow the same path + HTTP method combination by content type
- Safe: doesn't change server state (required only for GET)
- Idempotent: calling N times produces the same **final state** as calling once (GET, PUT, DELETE required; not POST; PATCH not guaranteed)
- 405 Method Not Allowed: the path exists but there's no mapping for this HTTP method (different from 404)

Quick reference:

```
@RequestMapping(path = "/x", method = RequestMethod.GET)  // base form
@GetMapping("/x")           // its shortcut -- the two are equivalent

@RequestMapping("/users")   // class-level shared prefix
class UserController {
    @GetMapping                     // GET /users
    @GetMapping("/{id}")            // GET /users/{id}
    @PostMapping                    // POST /users
    @PutMapping("/{id}")            // PUT /users/{id}     -- full replacement
    @PatchMapping("/{id}")          // PATCH /users/{id}   -- partial update
    @DeleteMapping("/{id}")         // DELETE /users/{id}  -- idempotent
}

@PostMapping(path = "/orders", consumes = MediaType.APPLICATION_JSON_VALUE)
// only matches requests with Content-Type: application/json
```

Glossary

Mapping annotation — The family of annotations that binds a controller method to a path + HTTP method (+ optional other conditions) combination.

`@RequestMapping` — The family's base annotation; can match any HTTP method via its `method` attribute, or all of them if unset.

Meta-annotation — An annotation built on top of another one, pre-configuring it with a specific attribute value (`@GetMapping` is a meta-annotation of `@RequestMapping(method=GET)`).

Safe (HTTP method) — The property of an HTTP method not changing server state when called; required only for GET (and HEAD/OPTIONS).

Idempotent (HTTP method) — The property guaranteeing that calling an HTTP method N times leaves the server's final state the same as calling it once.

405 Method Not Allowed — The HTTP status code returned when a path exists but no mapping matches the requested HTTP method.

`consumes` / `produces` — Mapping annotation attributes that narrow a mapping based on the request's `Content-Type` / `Accept` headers.

Appendix: Mini Project — A Simple Book CRUD API

We bring every annotation from this lesson together in one controller, over one real resource:

```
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.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

// A small, complete CRUD controller -- every mapping annotation from this lesson in
// one place, the same shape this project would use for a real /api/books endpoint.
@Controller
@RequestMapping("/api/books")
class BookCrudController {
    private final Map<Long, String> books = new LinkedHashMap<>();
    private long nextId = 1;

    @GetMapping
    @ResponseBody
    public List<String> list() {
        return List.copyOf(books.values());
    }

    @GetMapping("/{id}")
    @ResponseBody
    public ResponseEntity<String> getOne(@PathVariable Long id) {
        String title = books.get(id);
```

```
        return title != null ? ResponseEntity.ok(title) : ResponseEntity.status(HttpStatus.
NOT_FOUND).build();
    }

    @PostMapping
    @ResponseBody
    public ResponseEntity<Long> create(@RequestBody String title) {
        long id = nextId++;
        books.put(id, title);
        return ResponseEntity.status(HttpStatus.CREATED).body(id);
    }

    @PutMapping("/{id}")
    @ResponseBody
    public ResponseEntity<Void> replace(@PathVariable Long id, @RequestBody String title) {
        if (!books.containsKey(id)) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
        }
        books.put(id, title);
        return ResponseEntity.noContent().build();
    }

    @DeleteMapping("/{id}")
    @ResponseBody
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        books.remove(id);
        return ResponseEntity.noContent().build(); // idempotent: same 204 whether or not
it existed
    }
}
```

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

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

        ResponseEntity<Long> created = controller.create("Effective Java");
        System.out.println(created.getStatusCode() + " id=" + created.getBody());
        // 201 CREATED id=1

        System.out.println(controller.list());
        // [Effective Java]

        System.out.println(controller.getOne(1L).getBody());
        // Effective Java

        controller.replace(1L, "Effective Java (3rd Edition)");
        System.out.println(controller.getOne(1L).getBody());
        // Effective Java (3rd Edition)

        System.out.println(controller.delete(1L).getStatusCode());
        // 204 NO_CONTENT
        System.out.println(controller.getOne(1L).getStatusCode());
        // 404 NOT_FOUND
    }
}
```

`BookCrudController` opens with the class-level prefix pattern (`@RequestMapping("/api/books")`) we saw in "Combining @RequestMapping at the Class and Method Level"; `list()` / `getOne()` / `create()` / `replace()` / `delete()` cover the five CRUD operations with `@GetMapping` / `@GetMapping("/{id}")` / `@PostMapping` / `@PutMapping("/{id}")` / `@DeleteMapping("/{id}")` respectively. `BookCrudDemo` , just like `ProductCatalogDemo` in the Spring MVC Fundamentals lesson, runs the whole flow (create → list → update → delete → query again) end to end by calling the controller's methods directly, without a real DispatcherServlet.

Appendix: Mini Project — An HTTP-Method-Aware Router Simulation

The final mini project merges Spring MVC Fundamentals' `RequestRouterSimulation` with the `(path, method)` key we introduced in "Same Path, Different HTTP Methods":

```
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.RequestMethod;

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

// Fundamentals' RequestRouterSimulation registered multiple handlers but could only
```

```
// key by path. This version keys by (path, HTTP method) across MULTIPLE handler
// classes -- the closest our hand-rolled simulations get to real Spring routing.
class ArticleApiHandlers {
    @GetMapping("/articles")
    public String list() {
        return "Listing articles";
    }

    @PostMapping("/articles")
    public String create() {
        return "Creating an article";
    }
}

class CommentApiHandlers {
    @GetMapping("/comments")
    public String list() {
        return "Listing comments";
    }

    @DeleteMapping("/comments")
    public String deleteAll() {
        return "Deleting all comments";
    }
}

class RouterWithMethodSimulation {
    private final Map<RouteKey, HandlerEntry> registry = new HashMap<>();

    private record RouteKey(String path, RequestMethod method) {
    }

    private record HandlerEntry(Object instance, Method method) {
    }

    void register(Object handler) {
        for (Method method : handler.getClass().getDeclaredMethods()) {
            GetMapping get = method.getAnnotation(GetMapping.class);
            if (get != null) {
                registry.put(new RouteKey(get.value()[0], RequestMethod.GET), new
HandlerEntry(handler, method));
            }
            PostMapping post = method.getAnnotation(PostMapping.class);
            if (post != null) {
                registry.put(new RouteKey(post.value()[0], RequestMethod.POST), new
HandlerEntry(handler, method));
            }
            DeleteMapping delete = method.getAnnotation(DeleteMapping.class);
            if (delete != null) {
                registry.put(new RouteKey(delete.value()[0], RequestMethod.DELETE), new
HandlerEntry(handler, method));
            }
        }
    }

    String dispatch(String path, RequestMethod httpMethod) {
        HandlerEntry entry = registry.get(new RouteKey(path, httpMethod));
        if (entry == null) {
```

```
            return "405 Method Not Allowed: " + httpMethod + " " + path;
        }
        try {
            return (String) entry.method().invoke(entry.instance());
        } catch (ReflectiveOperationException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
import org.springframework.web.bind.annotation.RequestMethod;

class RouterWithMethodDemo {
    public static void main(String[] args) {
        RouterWithMethodSimulation router = new RouterWithMethodSimulation();
        router.register(new ArticleApiHandlers());
        router.register(new CommentApiHandlers());

        System.out.println(router.dispatch("/articles", RequestMethod.GET));
        // Listing articles
        System.out.println(router.dispatch("/articles", RequestMethod.POST));
        // Creating an article
        System.out.println(router.dispatch("/articles", RequestMethod.DELETE));
        // 405 Method Not Allowed: DELETE /articles
        System.out.println(router.dispatch("/comments", RequestMethod.DELETE));
        // Deleting all comments
    }
}
```

`RouterWithMethodSimulation.register(...)` now reads `@GetMapping`, `@PostMapping`, and `@DeleteMapping` -- **all three** -- and adds them to the same registry, keyed by `RouteKey(path, method)`. `ArticleApiHandlers` and `CommentApiHandlers` are two separate "controllers" with no knowledge of each other, but `dispatch(...)` finds either one correctly from a single place, matching on both path and HTTP method -- the complete fix for the gap the Fundamentals lesson's mini project left behind.

> Notice that `dispatch("/articles", RequestMethod.DELETE)` returns `"405 Method Not Allowed"` -- the `/articles` path exists in the registry (for `GET` and `POST`), just not for `DELETE`. This shows that the 404/405 distinction from "When an Unsupported HTTP Method Is Requested: 405 Method Not Allowed" emerges naturally even in a hand-written simulation.