

Spring MVC Fundamentals

DispatcherServlet, the MVC pattern (Model/View/Controller), and the difference between @Controller and @RestController.

Intermediate 35 min

Generated on 5 October 2026

Read online: <https://www.learnforgex.com/en/topics/spring-mvc-fundamentals>

Spring MVC Fundamentals

In the Spring Core category we learned Spring's **container**: how beans are defined (Dependency Injection, Spring IoC Container), how they're found automatically (Component Scanning), what Spring Boot auto-configures and when (Auto-Configuration), and how data consistency is preserved (Transaction Management). In this lesson we move to how that container responds to **HTTP requests** -- exactly what this project's own `HomeController` and `TopicController` do on every page visit. Spring MVC turns this familiar bean/container mechanism into a web layer built around the front controller pattern.

What Is Spring MVC?

Spring MVC is a web framework built on top of the Jakarta Servlet API, implementing the **Model-View-Controller** pattern. The difference becomes clear next to a plain Servlet:

```
// A plain HttpServlet: a separate class per endpoint, manual request/response handling.
public class ProductServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        resp.getWriter().write("Product list");
    }
}

// Spring MVC: a single method, its parameters say what it needs, its return value
// says what it produces.
@Controller
class ProductController {
    @GetMapping("/products")
    public String list() {
        return "product-list";
    }
}
```

In the second version we never touch `HttpServletRequest / HttpServletResponse` directly -- Spring translates the request into a method call, and the return value into a response. Who performs that translation is the subject of "DispatcherServlet: The Front Controller Pattern" below.

Why Does It Exist?

With the plain Servlet API, two problems grow as an application grows: (1) every URL needs its own servlet class, registered in `web.xml` (or by hand) -- a new endpoint means a new class and a new registration; (2) every servlet repeats the same boilerplate -- reading parameters, converting types, writing the response. Spring MVC hands this responsibility to **one single servlet** (DispatcherServlet) -- everything else (which method gets called, how parameters get bound, how the response gets built) is declared with annotations, with no need for a separate class or registration per endpoint.

History

Spring MVC shipped with Spring Framework 1.0 (2004), but was entirely XML-based at first -- controllers were classes implementing the `Controller` interface, and URL mappings were done with `<bean>` definitions. Spring 2.5 (2007) introduced annotation-based controllers with `@Controller` and `@RequestMapping` -- the same release that brought component scanning and `@Autowired`, as mentioned in the Component Scanning lesson's "History" section. Spring 3.0 (2009) standardized REST-style endpoints with `@PathVariable` and `@RequestBody / @ResponseBody`. Spring 4.3 (2016) added the shortcut annotations `@GetMapping / @PostMapping` (aliases for `@RequestMapping(method=...)`). Spring Boot 1.0 (2014) then added the embedded servlet container, eliminating the need to deploy to an external application server entirely -- the approach this project itself uses.

The MVC Pattern: Model, View, Controller

Let's set the framework aside and see the pattern in plain Java:

```

import java.util.List;

// Plain Java simulation of the MVC pattern -- no Spring involved yet. The goal is to
// separate three concerns before we let a framework take over the "wiring" for us:
//  Model      -- the data itself, with no knowledge of how it will be displayed.
//  View       -- knows how to turn a Model into an output format (here, a plain
//               String standing in for an HTML page).
//  Controller -- receives a request, asks a service for data, puts that data into a
//               Model, and hands the Model to a View.
class MvcPatternExample {

    // --- Model -----
    record BookListModel(List<String> titles) {
    }

    // --- View -----
    // A real View (Thymeleaf, JSP...) turns a Model into HTML. Here we just build a
    // String, so the example runs without any templating engine.
    static class BookListView {
        String render(BookListModel model) {
            StringBuilder html = new StringBuilder("<ul>\n");
            for (String title : model.titles()) {
                html.append("  <li>").append(title).append("</li>\n");
            }
            html.append("</ul>");
            return html.toString();
        }
    }

    // --- Controller -----
    static class BookListController {
        private final BookListView view = new BookListView();

        String handle() {
            // In a real app this would come from a service/repository, not a literal.
            BookListModel model = new BookListModel(List.of("Effective Java", "Clean
Code"));
            return view.render(model);
        }
    }

    public static void main(String[] args) {
        BookListController controller = new BookListController();
        System.out.println(controller.handle());
        // <ul>
        //   <li>Effective Java</li>
        //   <li>Clean Code</li>
        // </ul>
    }
}

```

The three roles are cleanly separated: `BookListModel` only carries data and knows nothing about how it will be displayed; `BookListView` turns that data into an output format (here a

`String`, in a real app HTML); `BookListController` brings the two together. In Spring MVC, the framework creates the `Model` object for you and the `ViewResolver` finds the real View for you -- but the responsibilities of the three roles stay exactly the same.

DispatcherServlet: The Front Controller Pattern

The heart of Spring MVC is `DispatcherServlet` -- a single servlet that receives **every** incoming HTTP request and routes it to the right controller, a **front controller**:

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

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

// A hand-rolled, drastically simplified stand-in for what DispatcherServlet does at
// startup and on every request. Real Spring builds a much richer HandlerMapping (path
// variables, HTTP method matching, content negotiation...), but the core idea is the
// same: scan for @GetMapping-annotated methods, remember which path maps to which
// method, then invoke the right one when a request comes in.
class ProductPageHandlers {

    @GetMapping("/products")
    public String listProducts() {
        return "Showing all products";
    }

    @GetMapping("/products/featured")
    public String featuredProducts() {
        return "Showing featured products";
    }
}

class FrontControllerSimulationExample {

    // Stands in for DispatcherServlet's HandlerMapping: builds a path -> method registry
    // by reading @GetMapping off every method of the handler object.
    static Map<String, Method> buildHandlerMapping(Object handler) {
        Map<String, Method> mapping = new HashMap<>();
        for (Method method : handler.getClass().getDeclaredMethods()) {
            GetMapping annotation = method.getAnnotation(GetMapping.class);
            if (annotation != null) {
                mapping.put(annotation.value()[0], method);
            }
        }
        return mapping;
    }

    // Stands in for DispatcherServlet's dispatch loop: given an incoming path, find the
    // matching method (HandlerMapping's job) and invoke it (HandlerAdapter's job).
    static String dispatch(String path, Object handler, Map<String, Method> mapping) throws
    Exception {
        Method method = mapping.get(path);
        if (method == null) {
            return "404 Not Found: " + path;
        }
    }
}
```

```

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

public static void main(String[] args) throws Exception {
    ProductPageHandlers handler = new ProductPageHandlers();
    Map<String, Method> handlerMapping = buildHandlerMapping(handler);

    System.out.println(dispatch("/products", handler, handlerMapping));
    // Showing all products
    System.out.println(dispatch("/products/featured", handler, handlerMapping));
    // Showing featured products
    System.out.println(dispatch("/products/unknown", handler, handlerMapping));
    // 404 Not Found: /products/unknown
}
}

```

`buildHandlerMapping` is a tiny simulation of what the real `DispatcherServlet` does at startup: work out in advance which method corresponds to which path and keep that in a registry. `dispatch` simulates the loop that runs on every request: look up the path, find the right method, and call it. The real `DispatcherServlet` obviously does far more (HTTP method matching, path variables, content negotiation...) -- we'll go deeper into that distinction in "HandlerMapping and HandlerAdapter: What Happens Inside DispatcherServlet?".

The Journey of an HTTP Request: Request Lifecycle

The path a request takes from the browser to the response ties together every piece we've seen so far into one flow:



Each step corresponds to its own section in this lesson: the front controller step to "DispatcherServlet: The Front Controller Pattern", the handler-selection step to "HandlerMapping and HandlerAdapter: What Happens Inside DispatcherServlet?", the Model /response-body split to "@Controller vs. @RestController: Which One, When?", and the ViewResolver step to "ViewResolver: From Logical View Name to HTML".

This flow actually runs on every `/lang/topics/slug` request in this project: DispatcherServlet routes the request to `TopicController.show(...)`, which populates the `Model` and returns the view name `"topic"` -- we'll walk through this in detail in "This Project's Own Controllers: A Real Spring MVC Example".

Embedded Tomcat and spring-boot-starter-web

A traditional Servlet application is deployed as a compiled `.war` file to an external application server (Tomcat, Jetty...). Spring Boot reverses this: the `spring-boot-starter-web` dependency brings not just Spring MVC but an **embedded** Tomcat, packaged into the application itself, with no separate install step -- the application becomes a self-contained, runnable JAR that hosts its own server:

```
@SpringBootApplication
public class LearningPlatformApplication {
    public static void main(String[] args) {
        SpringApplication.run(LearningPlatformApplication.class, args);
    }
}
```

This is this project's real `LearningPlatformApplication` class -- the moment `SpringApplication.run(...)` is called, the embedded Tomcat starts listening on the `server.port: 8080` setting from `application.yml`, with no external server setup required.

`spring-boot-devtools` (a runtime / optional dependency in this project's `pom.xml`) automatically restarts the embedded Tomcat whenever it detects a classpath change -- no need to stop and start it by hand during development.

@Controller with Your First Endpoint

The most basic controller is a class marked `@Controller` with a method marked `@GetMapping`:

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

// A traditional Spring MVC controller. The String it returns is NOT the HTTP response
// body -- it is a LOGICAL VIEW NAME. DispatcherServlet hands this name to a
// ViewResolver, which turns it into an actual template (e.g. templates/home.html).
// This class can't run on its own -- it needs a running DispatcherServlet and a
// configured ViewResolver, exactly like this project's own HomeController (see "This
// Project's Own Controllers").
@Controller
class HomePageController {

    @GetMapping("/")
    public String home() {
        return "home"; // resolves to templates/home.html, not the literal text "home"
    }
}
```

The string `"home"` returned by `home()` is **not** the HTTP response body -- it is the **logical view name**. `DispatcherServlet` hands this name to a `ViewResolver`, which turns it into a real template; we'll see that translation in "ViewResolver: From Logical View Name to HTML".

Model: Carrying Data from Controller to View

A view is rarely static -- the controller usually needs to hand it data. That's done with the `Model` parameter:

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

import java.util.List;

// Model is how a controller hands data to a view without knowing anything about HTML.
// DispatcherServlet creates one Model per request and passes it into the controller
// method; whatever you put in it becomes available to the template under the same key.
@Controller
class ProductListController {

    @GetMapping("/products")
    public String list(Model model) {
        model.addAttribute("products", List.of("Keyboard", "Monitor", "Mouse"));
        model.addAttribute("count", 3);
        return "product-list";
        // The template can now read ${products} and ${count} -- exactly the mechanism
        // this project uses for th:each="topicItem : ${category.topics()}" in
        // fragments/layout.html.
    }
}
```

`DispatcherServlet` creates a fresh `Model` for every request and passes it into the method as a parameter; anything added with `addAttribute(...)` becomes available to the view under

the same key. This is the framework-managed version of the `BookListModel` we saw in "The MVC Pattern: Model, View, Controller".

@RestController: Turning Off the View

Not every endpoint needs to produce an HTML page -- often you just want to return data (JSON) directly. That's what `@RestController` is for:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

// @RestController writes its return value DIRECTLY to the HTTP response body -- there
// is no view name, no ViewResolver, no template. A plain String becomes the entire
// response body, with Content-Type: text/plain.
@RestController
class HelloRestController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}
```

Here, the returned `"Hello, World!"` is not a view name -- it is the **response body itself**; no `ViewResolver` is ever involved. This difference between `@Controller` and `@RestController` is exactly the subject of "`@Controller` vs. `@RestController`: Which One, When?".

@RestController and JSON Serialization

`@RestController` isn't limited to returning `String` -- return an object and Spring automatically converts it to JSON:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

// Returning anything other than a String (a record, a List, a Map...) makes Spring hand
// it to Jackson, which serializes it to JSON and sets Content-Type: application/json
// automatically -- no manual serialization code anywhere.
@RestController
class ProductRestController {

    record Product(String name, double price) {
    }

    @GetMapping("/api/products/1")
    public Product getProduct() {
        return new Product("Keyboard", 49.90);
        // {"name":"Keyboard","price":49.9}
    }
}
```

`Product` is a record; we didn't write any manual serialization code, yet the response comes out as `{"name":"Keyboard","price":49.9}`. What does the conversion is Jackson, another implicit dependency pulled in by `spring-boot-starter-web` -- it automatically converts every return value marked (explicitly or implicitly, via `@ResponseBody`, detailed in the next section) to JSON.

@Controller vs. @RestController: Which One, When?

`@RestController` isn't a separate mechanism at all -- it's a meta-annotation combining `@Controller` with `@ResponseBody`:

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

// @RestController is not a separate mechanism from @Controller -- it is a
// meta-annotation combining @Controller with @ResponseBody. The two controllers below
// behave identically for this one endpoint.
@Controller
class ManualResponseBodyController {

    @GetMapping("/status")
    @ResponseBody
    public String status() {
        return "OK"; // @ResponseBody on the method: written directly to the body
    }
}

@RestController
class EquivalentRestController {

    @GetMapping("/status")
    public String status() {
        return "OK"; // @RestController applies @ResponseBody to every method by default
    }
}
```

`ManualResponseBodyController` achieves the exact same result as `@RestController` by adding `@ResponseBody` explicitly to its method. The rule is simple: **producing an HTML page means `@Controller`, producing data (JSON/XML) means `@RestController`** -- mixing the two in one class (some methods returning view names, others returning data via `@ResponseBody`) is technically possible, but keeping a controller to a single job is usually preferred for readability.

HandlerMapping and HandlerAdapter: What Happens Inside DispatcherServlet?

Back to `FrontControllerSimulationExample` from "DispatcherServlet: The Front Controller Pattern" -- its two methods correspond to two separate real Spring components: `buildHandlerMapping` is a tiny model of what the real `RequestMappingHandlerMapping` does (working out which path goes to which method, computed once at startup); `dispatch` is a model of what `RequestMappingHandlerAdapter` does (calling the method it found with the right parameters -- `Model`, `@PathVariable`, `@RequestBody` ...). Our simulation can only call parameterless methods; the real `HandlerAdapter` can read and populate every parameter type -- including `@PathVariable` / `@RequestParam`, which we'll see in the next lesson (Mapping Annotations & HTTP Methods) -- from the right part of the request.

ViewResolver: From Logical View Name to HTML

When a `@Controller` returns a view name, the component that turns that name into a real file is the `ViewResolver`. Because this project uses Thymeleaf, `spring-boot-starter-thymeleaf` auto-configures a `ThymeleafViewResolver` -- by default it prepends `classpath:/templates/` to the view name and appends `.html`. So a controller returning "home" renders `templates/home.html`; one returning "topic" renders `templates/topic.html`.

This project's `topic.html` and `index.html` are full-page templates, but they pull in the `navbar` / `sidebar` / `footer` fragments from `fragments/layout.html` with `th:replace` (details like the sidebar's `.[...]` / `#vars` mechanism are a separate topic). The `ViewResolver`'s job is only to find which top-level template file gets rendered -- pulling in fragments is entirely Thymeleaf's own mechanism, not Spring MVC's.

Spring MVC vs. Spring WebFlux (A Quick Look)

Spring MVC is built on the Servlet API -- it is **blocking**: every request occupies a thread from the server's thread pool until it's done (the embedded Tomcat this project uses works exactly this way). Spring WebFlux is a **reactive, non-blocking** alternative, built on Project Reactor and (by default) Netty -- it can handle many concurrent, long-lived connections (streaming, WebSocket) with only a handful of threads. The two come from separate starters (`spring-boot-starter-web` vs. `spring-boot-starter-webflux`) and are generally not used together -- this project uses `spring-boot-starter-web` because its needs are the classic request/response cycle: each page view is a short-lived DB query plus a render.

This Project's Own Controllers: A Real Spring MVC Example

You can see every concept from this lesson in this project's own source code.

`HomeController` and `TopicController` are marked `@Controller`, just like we saw in the Component Scanning lesson's "This Project's Own Classes: A Real Component Scanning Example" (not `@RestController` -- both produce HTML pages, not JSON), and are found automatically thanks to `@SpringBootApplication`'s implicit component scanning.

`HomeController.index(@PathVariable String lang, Model model)` puts the navigation data it gets from `NavigationService` into the view with `model.addAttribute("nav", ...)` and returns the view name `"index"` -- exactly the mechanism we saw in "Model: Carrying Data from Controller to View". `TopicController` maps its real content page to `/lang:en|tr|topics/{slug}` -- we'll look in detail at how a controller's mapping shape evolves (and when a shared class-level prefix stops making sense) in the next lesson (Mapping Annotations & HTTP Methods). Both controllers use constructor injection (exactly the pattern we saw in the Dependency Injection lesson's "Constructor Injection" section) -- all six of `TopicController`'s dependencies live in `final` fields, all injected through its one constructor.

Best Practices

- **Keep a controller to one job: produce HTML (`@Controller`) or produce data (`@RestController`), not both** -- mixing the two makes it harder for a reader to answer "what does this endpoint return?" (see "@Controller vs. @RestController: Which One, When?").
- **Keep controllers thin, push business logic into the service layer** -- the Controller -> Service -> Repository order in "The Journey of an HTTP Request: Request Lifecycle" depends on each layer having exactly one responsibility.
- **Pick view names (like `"home"`, `"topic"`) that match their template file names exactly, not arbitrary strings** -- the `ViewResolver`'s prefix/suffix convention (see "ViewResolver: From Logical View Name to HTML") handles the rest automatically, but a naming mismatch produces errors that only show up at runtime.
- **Prefer constructor injection in controllers too** -- the reasons from the Dependency Injection lesson's "Why Is Constructor Injection Recommended?" section (testability, `final` fields) apply here as well; this project's own `TopicController` follows exactly this pattern (see "This Project's Own Controllers: A Real Spring MVC Example").

Common Mistakes

1. Expecting a response body from `@Controller` instead of a view name, then being confused by a blank or broken page. The string an `@Controller` method returns is a view name -- if you want the response body directly, add `@ResponseBody` or use `@RestController` instead (see "@Controller vs. @RestController: Which One, When?").

2. Trying to return an HTML page from `@RestController`. `@RestController` always applies `@ResponseBody` -- whatever you return (a string, an object) is written straight to the response body without ever going through a `ViewResolver`; putting HTML in that string does not render a template (see "`@RestController: Turning Off the View`").

3. Treating `DispatcherServlet` as an unquestionable black box. What it actually does, as shown in "HandlerMapping and HandlerAdapter: What Happens Inside `DispatcherServlet`?", is a clear two-step process: first find the method matching the path, then call that method with the right parameters -- a much richer version of the `buildHandlerMapping / dispatch` pair in `FrontControllerSimulationExample`.

4. Assuming embedded Tomcat is some Spring Boot-specific, different kind of server. The Tomcat brought in by `spring-boot-starter-web` is exactly the same Tomcat as an external installation -- the difference is that it's bundled into the JAR and started automatically by `SpringApplication.run(...)`, instead of being installed and deployed to manually (see "Embedded Tomcat and spring-boot-starter-web").

5. Assuming a view name will automatically match a template file name, then putting a file with the wrong name under `templates/`. The `ViewResolver`'s prefix/suffix rule only checks for an **exact string match** -- returning `"topic"` and creating `templates /Topic.html` (capital T) will 404 on case-sensitive file systems (see "`ViewResolver: From Logical View Name to HTML`").

6. Assuming Spring MVC and Spring WebFlux can be swapped in for each other freely. The two rest on separate starters (and separate server models -- servlet vs. reactive); if both are on the classpath, Spring Boot tries to pick one automatically, but this is generally confusing -- an application should commit to exactly one (see "Spring MVC vs. Spring WebFlux (A Quick Look)").

Summary, Cheat Sheet, and Glossary

Spring MVC is a web framework built on the Servlet API, organized around the front controller (`DispatcherServlet`) pattern. Key points:

- `DispatcherServlet`: the single entry point for every incoming request; uses `HandlerMapping` to find the right method and `HandlerAdapter` to call it
- `@Controller`: a method's return value is a **view name**, translated into a real template by a `ViewResolver`
- `@RestController`: `@Controller` + `@ResponseBody`; a method's return value is the **response body directly** (plain text for a String, JSON via Jackson for an object)
- `Model`: carries data from controller to view; created fresh by `DispatcherServlet` for every request
- `ViewResolver`: turns a logical view name (`"home"`) into a real file (`templates/home.html`)
- Embedded Tomcat: the server bundled inside the JAR by `spring-boot-starter-web` -- no external deployment needed

- Spring WebFlux: Spring MVC's reactive/non-blocking alternative, a separate starter

Quick reference:

```
@Controller                                // produces an HTML page
class PageController {
    @GetMapping("/page")
    String page(Model model) {
        model.addAttribute("key", "value");
        return "page-template";    // a view name, not the response body
    }
}

@RestController                            // produces data (JSON)
class ApiController {
    @GetMapping("/api/data")
    SomeRecord data() {
        return new SomeRecord(...); // response body directly, converted to JSON by Jackson
    }
}

@Controller
class MixedController {
    @GetMapping("/status")
    @ResponseBody                        // mimics @RestController for a single method
    String status() { return "OK"; }
}
```

Glossary

Spring MVC — A Spring web framework built on the Jakarta Servlet API, implementing the Model-View-Controller pattern.

DispatcherServlet — The single entry point that receives every incoming HTTP request and routes it to the right controller (a front controller).

@Controller — A controller annotation whose method return value is interpreted as a view name.

@RestController — The combination of **@Controller** and **@ResponseBody**; makes a method's return value the response body directly.

Model — The object used to carry data from controller to view; created fresh by DispatcherServlet for every request.

ViewResolver — The component that turns a logical view name into a real template file (in this project, **ThymeleafViewResolver**).

HandlerMapping — The component that determines which controller method a given HTTP request corresponds to.

HandlerAdapter — The component that calls the method found by **HandlerMapping** with the correct parameters.

Embedded servlet container — A server bundled inside the application itself, requiring no external install (embedded Tomcat, in this project).

Spring WebFlux — Spring MVC's reactive/non-blocking alternative, built on Project Reactor.

Appendix: Mini Project — Same Data, Two Controllers

In this project's own architecture, HTML pages and a (not yet written) JSON API could be different presentations of the same data. Let's build that at a small scale -- one "service", two controllers:

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

// Two controllers, one shared "service", two very different jobs: ProductPageController
// serves an HTML page (a view name + a Model), ProductApiController serves raw data (a
// List, serialized to JSON by Jackson). Neither controller knows the other exists --
// this is the same "MVC vs REST" split this project itself could make between
// TopicController's HTML page and a hypothetical JSON API over the same content.
class ProductCatalogService {
    record Product(String name, double price) {
    }

    List<Product> findAll() {
        return List.of(
            new Product("Keyboard", 49.90),
            new Product("Monitor", 199.00));
    }
}

@Controller
class ProductPageController {
    private final ProductCatalogService service = new ProductCatalogService();

    @GetMapping("/products")
    public String page(Model model) {
        model.addAttribute("products", service.findAll());
        return "products"; // would resolve to templates/products.html
    }
}

@RestController
class ProductApiController {
    private final ProductCatalogService service = new ProductCatalogService();

    @GetMapping("/api/products")
    public List<ProductCatalogService.Product> api() {
        return service.findAll(); // serialized straight to JSON by Jackson
    }
}
```

```

import org.springframework.ui.ExtendedModelMap;
import org.springframework.ui.Model;

import java.util.List;

// We can't start a real DispatcherServlet here, but both controllers are plain objects
// with plain methods -- nothing stops us from calling them directly to see exactly what
// they would hand off to a ViewResolver (the Model) or to Jackson (the return value).
class ProductCatalogDemo {
    public static void main(String[] args) {
        ProductPageController pageController = new ProductPageController();
        Model model = new ExtendedModelMap();
        String viewName = pageController.page(model);
        System.out.println("View name: " + viewName);
        // View name: products
        System.out.println("Model: " + model.asMap());
        // Model: {products=[Product[name=Keyboard, price=49.9], Product[name=Monitor,
price=199.0]]}

        ProductApiController apiController = new ProductApiController();
        List<ProductCatalogService.Product> products = apiController.api();
        System.out.println("API result: " + products);
        // API result: [Product[name=Keyboard, price=49.9], Product[name=Monitor, price=199.
0]]
    }
}

```

`ProductPageController` and `ProductApiController` both use the same `ProductCatalogService`, but one populates a `Model` (for the View), the other returns the data directly (for Jackson). `ProductCatalogDemo` runs both controllers by **calling their methods directly**, without a real `DispatcherServlet` -- we build `ExtendedModelMap`, a real implementation of `Model`, by hand and pass it into `page(model)`, exactly what `DispatcherServlet` does behind the scenes on every request.

Appendix: Mini Project — A Multi-Controller Request Routing Simulation

The final mini project takes `FrontControllerSimulationExample` from "DispatcherServlet: The Front Controller Pattern" a step further -- a real application has not one but **dozens** of `@Controller` beans; `DispatcherServlet` merges all of them into a single registry:

```

import org.springframework.web.bind.annotation.GetMapping;

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

// A slightly more complete front-controller simulation than
// FrontControllerSimulationExample: this one registers MULTIPLE handler objects (like a
// real app has many @Controller beans) and builds one shared registry across all of

```

```

// them -- exactly what DispatcherServlet's HandlerMapping does across every
// @Controller/@RestController bean the container finds.
class HomeHandlers {
    @GetMapping("/")
    public String home() {
        return "Welcome to the store";
    }
}

class CartHandlers {
    @GetMapping("/cart")
    public String cart() {
        return "Your cart is empty";
    }

    @GetMapping("/cart/checkout")
    public String checkout() {
        return "Redirecting to checkout";
    }
}

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

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

    // Runs once, across every registered handler -- like component scanning discovering
    // every @Controller and DispatcherServlet indexing its methods at startup.
    void register(Object handler) {
        for (Method method : handler.getClass().getDeclaredMethods()) {
            GetMapping mapping = method.getAnnotation(GetMapping.class);
            if (mapping != null) {
                registry.put(mapping.value()[0], new HandlerEntry(handler, method));
            }
        }
    }

    // Runs on every request -- like DispatcherServlet's dispatch loop.
    String dispatch(String path) {
        HandlerEntry entry = registry.get(path);
        if (entry == null) {
            return "404 Not Found: " + path;
        }
        try {
            return (String) entry.method().invoke(entry.instance());
        } catch (ReflectiveOperationException e) {
            throw new RuntimeException(e);
        }
    }

    List<String> registeredPaths() {
        return List.copyOf(registry.keySet());
    }
}

```

```

class RequestRouterDemo {
    public static void main(String[] args) {
        RequestRouterSimulation router = new RequestRouterSimulation();
        router.register(new HomeHandlers());
        router.register(new CartHandlers());

        System.out.println(router.dispatch("/"));
        // Welcome to the store
        System.out.println(router.dispatch("/cart"));
        // Your cart is empty
        System.out.println(router.dispatch("/cart/checkout"));
        // Redirecting to checkout
        System.out.println(router.dispatch("/unknown"));
        // 404 Not Found: /unknown

        System.out.println("Registered paths: " + router.registeredPaths().size());
        // Registered paths: 3
    }
}

```

Every call to `RequestRouterSimulation.register(...)` scans a new handler object's methods and adds them to the same shared registry -- `HomeHandlers` and `CartHandlers` know nothing about each other, but `dispatch(...)` can find either one from the same place. This is the HTTP-routing counterpart to the same idea we saw with component scanning gathering multiple `@Component` classes into a single container.

This simulation only does exact path matching -- the path-variable resolution a real `HandlerMapping` performs (like `/cart/{id}`), HTTP-method disambiguation (the same path mapped to both `GET` and `POST`), and content negotiation are all absent here. We'll see those with real Spring annotations in the next lesson (Mapping Annotations & HTTP Methods).