Spring MVC Temelleri

DispatcherServlet, MVC deseni (Model/View/Controller) ve @Controller ile @RestController arasındaki fark.

Orta 35 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/spring-mvc-fundamentals

---

Spring MVC Temelleri

Spring Core kategorisinde Spring'in **container'ını** öğrendik: bean'lerin nasıl tanımlandığını (Dependency Injection, Spring IoC Container), nasıl otomatik bulunduğunu (Component Scanning), Spring Boot'un neyi ne zaman otomatik yapılandırdığını (Auto-Configuration) ve veri tutarlılığının nasıl korunduğunu (Transaction Management). Bu derste, o container'ın **HTTP isteklerine** nasıl yanıt verdiğine geçiyoruz -- tam olarak bu projenin kendi `HomeController` ve `TopicController` sınıflarının her sayfa ziyaretinde yaptığı iş. Spring MVC, bu bilinen bean/container mekanizmasını, front controller deseniyle birleştirilmiş bir web katmanına dönüştürür.

Spring MVC Nedir?

Spring MVC, Jakarta Servlet API'sinin üzerine inşa edilmiş, **Model-View-Controller** desenini uygulayan bir web framework'üdür. Saf bir Servlet ile karşılaştırınca fark netleşir:

```
// Saf bir HttpServlet: her endpoint için ayrı bir sınıf, istek/yanıtla elle uğraşmak.
public class ProductServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        resp.getWriter().write("Product list");
    }
}

// Spring MVC: tek bir metot, imzasındaki parametrelerle ne istediğini,
// dönüş değeriyle de ne üreteceğini söylüyor.
@Controller
class ProductController {
    @GetMapping("/products")
    public String list() {
        return "product-list";
    }
}
```

İkinci versiyonda `HttpServletRequest` / `HttpServletResponse` ile elle uğraşmıyoruz -- Spring, isteği bir metot çağrısına, yanıtı da bir dönüş değerine çeviriyor. Bu çeviriyi kim yapıyor, aşağıdaki "DispatcherServlet: Front Controller Deseni" bölümünün konusu.

Neden Var?

Saf Servlet API'siyle bir uygulama büyüdükçe iki sorun büyür: (1) her URL için ayrı bir servlet sınıfı yazıp `web.xml`'de (ya da elle) eşlemek gerekir -- yeni bir endpoint, yeni bir sınıf ve yeni bir kayıt demektir; (2) her servlet, parametre okuma, tip dönüşümü, yanıt yazma gibi aynı boilerplate'i tekrar tekrar içerir. Spring MVC bu sorumluluğu **tek bir servlet'e** (DispatcherServlet) devrederek çözer -- geri kalan her şey (hangi metodun çağrılacağı, parametrelerin nasıl bağlanacağı, yanıtın nasıl üretileceği) annotation'larla deklaratif olarak belirtilir, ayrı ayrı sınıf/kayıt gerekmez.

Tarihçe

Spring MVC, Spring Framework 1.0 (2004) ile birlikte geldi, ama başlangıçta tamamen XML tabanlıydı -- controller'lar `Controller` arayüzünü implemente eden sınıflardı, URL eşlemeleri `<bean>` tanımlarıyla yapılırdı. Spring 2.5 (2007), `@Controller` ve `@RequestMapping` ile annotation tabanlı controller'ları getirdi -- Component Scanning dersinin "Tarihçe" bölümünde bahsettiğimiz component scanning ve `@Autowired` de aynı sürümle geldi. Spring 3.0 (2009), `@PathVariable` ve `@RequestBody` / `@ResponseBody` ile REST tarzı endpoint'leri standartlaştırdı. Spring 4.3 (2016), `@GetMapping` / `@PostMapping` gibi kısayol annotation'larını (`@RequestMapping(method=...)` için) ekledi. Spring Boot 1.0 (2014) ise embedded servlet container'ı getirerek harici bir uygulama sunucusuna (Tomcat, WildFly...) deploy etme ihtiyacını tamamen ortadan kaldırdı -- bu projenin de kullandığı yol.

MVC Deseni: Model, View, Controller

Framework'ü bir kenara bırakıp deseni saf Java ile görelim:

```
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>
    }
}
```

Üç rol birbirinden net ayrılmış: `BookListModel` yalnızca veriyi taşıyor, nasıl gösterileceğini bilmiyor; `BookListView` veriyi bir çıktı formatına (burada bir `String`, gerçek uygulamada

HTML) çeviriyor; `BookListController` ikisini bir araya getiriyor. Spring MVC'de framework, `Model` nesnesini senin için oluşturur ve `ViewResolver` gerçek View'ı senin için bulur -- ama üç rolün sorumluluğu birebir aynı kalır.

DispatcherServlet: Front Controller Deseni

Spring MVC'nin kalbi DispatcherServlet'tir -- gelen **her** HTTP isteğini karşılayan tek bir servlet, isteği doğru controller'a yönlendiren bir **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`, gerçek DispatcherServlet'in başlangıçta yaptığı işin minik bir simülasyonu: hangi metodun hangi path'e karşılık geldiğini önceden çıkarıp bir registry'de tutmak. `dispatch` ise her istekte çalışan asıl döngünün simülasyonu: path'e bakıp doğru metodu bulmak ve çağırmak. Gerçek DispatcherServlet elbette çok daha fazlasını yapar (HTTP metodu eşleşmesi, path variable'lar, content negotiation...) -- "HandlerMapping ve HandlerAdapter: DispatcherServlet'in İçinde Neler Oluyor?" bölümünde bu ayrımı derinleştireceğiz.

Bir HTTP İsteğinin Yolculuğu: Request Lifecycle

Bir isteğin tarayıcıdan yanıta kadar izlediği yol, gördüğümüz parçaları tek bir akışta birleştirir:

```
Browser
   |
   v
HTTP Request
   |
   v
DispatcherServlet          (front controller)
   |
   v
HandlerMapping              (hangi Controller, hangi metot?)
   |
   v
Controller -> Service -> Repository
   |
   v
Model + view adı   VEYA   doğrudan response body
   |
   v
ViewResolver (yalnızca @Controller için) -> HTML
   |
   v
HTTP Response
```

Adımların her biri, bu dersin ayrı bir bölümüne karşılık geliyor: front controller "DispatcherServlet: Front Controller Deseni" bölümünün, handler seçimi "HandlerMapping ve HandlerAdapter: DispatcherServlet'in İçinde Neler Oluyor?" bölümünün, Model/response body ayrımı "@Controller vs @RestController: Ne Zaman Hangisi?" bölümünün, ViewResolver adımı da "ViewResolver: Mantıksal View Adından HTML'e" bölümünün konusu.

Bu projede bu akış, her `/{lang}/topics/{slug}` isteğinde gerçekten çalışıyor: `DispatcherServlet`, isteği `TopicController.show(...)`'a yönlendiriyor, o da `Model`'i dolduruyor ve `"topic"` view adını döndürüyor -- "Bu Projenin Kendi Controller'ları: Gerçek Bir Spring MVC Örneği" bölümünde bunu ayrıntısıyla göreceğiz.

Embedded Tomcat ve spring-boot-starter-web

Geleneksel bir Servlet uygulaması, derlenmiş bir `.war` dosyası olarak harici bir uygulama sunucusuna (Tomcat, Jetty...) deploy edilir. Spring Boot bunu tersine çevirir: `spring-boot-starter-web` bağımlılığı, Spring MVC'nin yanında **embedded** bir Tomcat'i de (ayrı bir kurulum gerektirmeden) projenin kendisine taşır -- uygulama, kendi içinde bir sunucu barındıran, tek başına çalıştırılabilir bir JAR olur:

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

Bu, bu projenin gerçek `LearningPlatformApplication` sınıfı -- `SpringApplication.run(...)` çağrıldığı anda embedded Tomcat, `application.yml`'deki `server.port: 8080` ayarında dinlemeye başlar, hiçbir harici sunucu kurulumu gerekmez.

> `spring-boot-devtools` (bu projenin `pom.xml`'inde `runtime` / `optional` olarak tanımlı bir bağımlılık), classpath'te bir değişiklik algıladığında embedded Tomcat'i otomatik yeniden başlatır -- geliştirme sırasında elle durdurup başlatmaya gerek kalmaz.

@Controller ile İlk Endpoint

En temel controller, `@Controller` ile işaretlenmiş bir sınıf ve içinde `@GetMapping` ile işaretlenmiş bir metottur:

```
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"
    }
}
```

`home()` metodunun döndürdüğü `"home"` string'i, HTTP yanıtının **gövdesi değil** -- **mantıksal view adı**dır. DispatcherServlet bu adı bir `ViewResolver`'a teslim eder, o da adı gerçek bir template'e çevirir; bu çeviriyi "ViewResolver: Mantıksal View Adından HTML'e" bölümünde göreceğiz.

Model: Controller'dan View'a Veri Taşımak

Bir view çoğu zaman statik değildir -- controller'ın ona veri aktarması gerekir. Bu iş `Model` parametresiyle yapılır:

```
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, her istek için yeni bir `Model` nesnesi oluşturup metoda parametre olarak geçirir; `addAttribute(...)` ile eklenen her şey, aynı anahtarla view'a (Thymeleaf'te `${...}` ile) ulaşır. Bu, "MVC Deseni: Model, View, Controller" bölümünde gördüğümüz `BookListModel` 'in framework tarafından yönetilen hâlidir.

@RestController: View'ı Devre Dışı Bırakmak

Her endpoint bir HTML sayfası üretmek zorunda değil -- çoğu zaman doğrudan veri (JSON) döndürmek istenir. Bunun için `@RestController` kullanılır:

```
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!";
    }
}
```

Burada dönen `"Hello, World!"` bir view adı değil, HTTP yanıtının **doğrudan gövdesi**dir -- hiçbir `ViewResolver` devreye girmez. `@Controller` ile `@RestController` arasındaki bu fark, "@Controller vs @RestController: Ne Zaman Hangisi?" bölümünün tam olarak konusu.

@RestController ve JSON Serileştirme

`@RestController` yalnızca `String` döndürmekle sınırlı değil -- bir nesne döndürüldüğünde Spring, onu otomatik olarak JSON'a çevirir:

```
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` bir record; hiçbir elle serileştirme kodu yazmadık, ama yanıt `{"name":"Keyboard","price":49.9}` olarak dönüyor. Bunu yapan, classpath'teki Jackson kütüphanesi -- `spring-boot-starter-web`'in getirdiği bir diğer örtük bağımlılık, `@ResponseBody` (bir sonraki bölümde detaylandıracağımız) işaretli her dönüş değerini otomatik JSON'a çevirir.

@Controller vs @RestController: Ne Zaman Hangisi?

`@RestController` aslında ayrı bir mekanizma değil -- `@Controller` ile `@ResponseBody`'nin birleşimi olan bir meta-annotation:

```
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`, metoduna açıkça `@ResponseBody` ekleyerek `@RestController` ile aynı sonucu elde ediyor. Kural basit: **HTML sayfası üretiyorsan `@Controller`, veri (JSON/XML) üretiyorsan `@RestController`** -- bir sınıfta ikisini karıştırmak (bazı metotlarda view adı, bazılarında `@ResponseBody` ile veri döndürmek) da mümkündür, ama okunabilirlik için genelde bir controller'ın tek bir işi olması tercih edilir.

HandlerMapping ve HandlerAdapter: DispatcherServlet'in İçinde Neler Oluyor?

"DispatcherServlet: Front Controller Deseni" bölümündeki `FrontControllerSimulationExample` 'a dönelim -- oradaki iki metot, gerçek Spring'in iki ayrı bileşenine karşılık geliyor: `buildHandlerMapping`, gerçek `RequestMappingHandlerMapping`'in yaptığı işin (hangi path hangi metoda gidiyor, uygulama başlarken bir kez hesaplanır) minik bir modeli; `dispatch` ise `RequestMappingHandlerAdapter`'ın yaptığı işin (bulunan metodu, doğru parametrelerle -- `Model`, `@PathVariable`, `@RequestBody` ... -- çağırmak) modeli. Simülasyonumuz yalnızca parametresiz metotları çağırabiliyor; gerçek `HandlerAdapter`, bir sonraki konuda (Request Mapping & HTTP Methods) göreceğimiz `@PathVariable` / `@RequestParam` gibi her parametre türünü isteğin ilgili parçasından okuyup doldurabiliyor.

ViewResolver: Mantıksal View Adından HTML'e

`@Controller` bir view adı döndürdüğünde, o adı gerçek bir dosyaya çeviren bileşen `ViewResolver`'dır. Bu proje Thymeleaf kullandığı için `spring-boot-starter-thymeleaf`, `ThymeleafViewResolver`'ı otomatik yapılandırır -- varsayılan olarak view adının önüne `classpath:/templates/`, arkasına `.html` ekler. Yani `"home"` döndüren bir controller, `templates/home.html`'i render eder; `"topic"` döndüren bir controller `templates/topic.html`'i.

> Bu proje `topic.html` ve `index.html` gibi tam sayfa template'ler kullanıyor, ama bunlar `fragments/layout.html`'deki `navbar` / `sidebar` / `footer` fragment'lerini `th:replace` ile içe alıyor (bkz. sidebar'daki `.?[...]` / `#vars` mekanizması gibi Thymeleaf detayları, ayrı bir konu). `ViewResolver`'ın işi yalnızca hangi ana template dosyasının render edileceğini bulmak -- fragment içe alma tamamen Thymeleaf'in kendi mekanizması, Spring MVC'nin değil.

Spring MVC vs Spring WebFlux (Kısa Bakış)

Spring MVC, Servlet API üzerine kuruludur -- **blocking**'tir: her istek, sunucunun thread pool'undan bir thread'i, işlem bitene kadar meşgul eder (bu projenin kullandığı embedded Tomcat, bunu tam olarak bu şekilde yapar). Spring WebFlux ise Project Reactor ve (varsayılan olarak) Netty üzerine kurulu, **reactive** ve **non-blocking** bir alternatiftir -- az sayıda thread'le çok sayıda eşzamanlı, uzun süren bağlantıyı (streaming, WebSocket gibi) yönetebilir. İkisi ayrı starter'lar (`spring-boot-starter-web` vs `spring-boot-starter-webflux`) üzerinden gelir ve genelde bir arada kullanılmaz -- bu proje, ihtiyacının klasik istek/yanıt döngüsü olması (her sayfa görüntüleme, kısa süren bir DB sorgusu + render) nedeniyle `spring-boot-starter-web`'i kullanıyor.

Bu Projenin Kendi Controller'ları: Gerçek Bir Spring MVC Örneği

Bu dersteki her kavramı, bu projenin kendi kaynak kodunda görebilirsin. `HomeController` ve `TopicController`, Component Scanning dersinin "Bu Projenin Kendi Sınıfları: Gerçek Bir Component Scanning Örneği" bölümünde gördüğümüz gibi `@Controller` ile işaretli (`@RestController` değil -- ikisi de HTML sayfası üretiyor, JSON değil) ve `@SpringBootApplication`'ın örtük component scanning'i sayesinde otomatik bulunuyor.

`HomeController.index(@PathVariable String lang, Model model)`, `NavigationService`'ten aldığı navigasyon verisini `model.addAttribute("nav", ...)` ile View'a taşıyor ve `"index"` view adını döndürüyor -- tam olarak "Model: Controller'dan View'a Veri Taşımak" bölümünde gördüğümüz mekanizma. `TopicController`, gerçek içerik sayfasını `/{lang:en|tr}/topics/{slug}`'e map ediyor -- bir controller'ın mapping şeklinin nasıl evrildiğini (ve paylaşılan bir sınıf-seviyesi önekin ne zaman anlamını yitirdiğini) bir sonraki konuda (Request Mapping &

HTTP Methods) ayrıntısıyla göreceğiz. Her iki controller de constructor injection kullanıyor (Dependency Injection dersinin "Constructor Injection" bölümünde gördüğümüz desenin ta kendisi) -- `TopicController`'ın altı bağımlılığı, hepsi `final` alanlarda, hepsi tek constructor'da.

Best Practices

- **Bir controller'ın tek bir işi olsun: ya HTML üretsin (`@Controller`) ya da veri üretsin (`@RestController`)** -- ikisini karıştırmak, sınıfı okuyanın "bu endpoint ne döndürüyor?" sorusuna cevap vermesini zorlaştırır (bkz. "@Controller vs @RestController: Ne Zaman Hangisi?").
- **Controller'ları ince tut, iş mantığını service katmanına bırak** -- "Bir HTTP İsteğinin Yolculuğu: Request Lifecycle" akışındaki Controller -> Service -> Repository sırası, her katmanın tek bir sorumluluğu olmasına dayanır.
- **View adlarını (`"home"`, `"topic"` gibi) sabit string yerine anlamlı, template dosya adıyla birebir eşleşen isimler olarak seç** -- `ViewResolver`'ın prefix/suffix kuralı (bkz. "ViewResolver: Mantıksal View Adından HTML'e") bunu otomatik yapıyor, ama isim tutarsızlığı çalışma zamanına kadar fark edilmeyen hatalara yol açar.
- **Constructor injection'ı controller'larda da tercih et** -- Dependency Injection dersinin "Neden Constructor Injection Öneriliyor?" bölümündeki gerekçeler (test edilebilirlik, `final` alanlar) burada da geçerli; bu projenin `TopicController`'ı tam olarak bu deseni izliyor (bkz. "Bu Projenin Kendi Controller'ları: Gerçek Bir Spring MVC Örneği").

Yaygın Hatalar

1. `@Controller`'dan bir view adı yerine yanıt gövdesi bekleyip boş/hatalı sayfa görmek. `@Controller` metodunun döndürdüğü string bir view adıdır -- doğrudan yanıt gövdesi istiyorsan ya `@ResponseBody` eklemeli ya da `@RestController` kullanmalısın (bkz. "@Controller vs @RestController: Ne Zaman Hangisi?").

2. `@RestController`'dan bir HTML sayfası döndürmeye çalışmak. `@RestController` her zaman `@ResponseBody` uygular -- döndürdüğün string (ya da nesne), hiçbir `ViewResolver`'dan geçmeden doğrudan yanıt gövdesine yazılır; içine HTML koysan bile bir template render edilmez (bkz. "@RestController: View'ı Devre Dışı Bırakmak").

3. DispatcherServlet'in "otomatik" bir mucize olduğunu, ne yaptığını sorgulamamak. Aslında yaptığı, "HandlerMapping ve HandlerAdapter: DispatcherServlet'in İçinde Neler Oluyor?" bölümünde gördüğümüz gibi, açık bir iki adımlı süreç: önce path'e uyan metodu bul, sonra o metodu doğru parametrelerle çağır -- `FrontControllerSimulationExample`'daki `buildHandlerMapping` / `dispatch` çiftinin, çok daha kapsamlı bir versiyonu.

4. Embedded Tomcat'in Spring Boot'a özel, ayrı bir sunucu türü olduğunu sanmak. `spring-boot-starter-web`'in getirdiği Tomcat, tam olarak harici bir Tomcat kurulumuyla aynı

Tomcat'tir -- fark, onu manuel kurup deploy etmek yerine JAR'ın içine gömülmüş olarak, `SpringApplication.run(...)` ile otomatik başlatılmasıdır (bkz. "Embedded Tomcat ve spring-boot-starter-web").

5. View adının, template dosya adıyla otomatik eşleşeceğini varsayıp `templates/` klasörüne yanlış isimde bir dosya koymak. `ViewResolver`'ın prefix/suffix kuralı yalnızca **string birebir eşleşmesine** bakar -- `"topic"` döndürüp `templates/Topic.html` (büyük T ile) oluşturmak, dosya sisteminin case-sensitive olduğu ortamlarda 404'e yol açar (bkz. "ViewResolver: Mantıksal View Adından HTML'e").

6. Spring MVC ile Spring WebFlux'ün aynı anda, birbirinin yerine kullanılabileceğini sanmak. İkisi ayrı starter'lar (ve ayrı sunucu modelleri -- servlet vs reactive) üzerine kuruludur; classpath'te ikisi birden varsa Spring Boot hangisini kullanacağını otomatik seçmeye çalışır, ama bu genelde kafa karıştırıcıdır -- bir uygulama için ikisinden yalnızca biri tercih edilmelidir (bkz. "Spring MVC vs Spring WebFlux (Kısa Bakış)").

Özet, Cheat Sheet ve Terimler Sözlüğü

Spring MVC, Servlet API üzerine kurulu, front controller (DispatcherServlet) deseniyle çalışan bir web framework'ü. Öne çıkan noktalar:

- `DispatcherServlet`: gelen her isteği karşılayan tek giriş noktası; `HandlerMapping` ile doğru metodu bulur, `HandlerAdapter` ile o metodu çağırır
- `@Controller`: metodu dönüş değeri bir **view adı**dır, `ViewResolver` bunu gerçek bir template'e çevirir
- `@RestController`: `@Controller` + `@ResponseBody`; dönüş değeri **doğrudan yanıt gövdesi** dir (String ise düz metin, nesne ise Jackson ile JSON)
- `Model`: controller'dan view'a veri taşıyan, DispatcherServlet'in her istek için oluşturduğu nesne
- `ViewResolver`: mantıksal view adını (`"home"`) gerçek bir dosyaya (`templates/home.html`) çeviren bileşen
- Embedded Tomcat: `spring-boot-starter-web`'in getirdiği, JAR'ın içine gömülü sunucu -- harici deploy gerektirmez
- Spring WebFlux: Spring MVC'nin reactive/non-blocking alternatifi, ayrı bir starter

Hızlı referans:

```
@Controller                          // HTML sayfası üretir
class PageController {
    @GetMapping("/page")
    String page(Model model) {
        model.addAttribute("key", "value");
        return "page-template";     // view adı, yanıt gövdesi değil
    }
}

@RestController                      // veri (JSON) üretir
class ApiController {
    @GetMapping("/api/data")
    SomeRecord data() {
        return new SomeRecord(...); // doğrudan yanıt gövdesi, Jackson ile JSON'a çevrilir
    }
}

@Controller
class MixedController {
    @GetMapping("/status")
    @ResponseBody                    // tek bir metotta @RestController'ı taklit eder
    String status() { return "OK"; }
}
```

Terimler Sözlüğü

Spring MVC — Jakarta Servlet API üzerine kurulu, Model-View-Controller desenini uygulayan Spring web framework'ü.

DispatcherServlet — Gelen her HTTP isteğini karşılayan, doğru controller'a yönlendiren tek giriş noktası (front controller).

`@Controller` — Metodunun dönüş değerini bir view adı olarak yorumlatan controller annotation'ı.

`@RestController` — `@Controller` ve `@ResponseBody`'nin birleşimi; dönüş değerini doğrudan yanıt gövdesi yapar.

`Model` — Controller'dan view'a veri taşımak için kullanılan, DispatcherServlet'in her istek için oluşturduğu nesne.

`ViewResolver` — Mantıksal view adını gerçek bir template dosyasına çeviren bileşen (bu projede `ThymeleafViewResolver`).

HandlerMapping — Hangi HTTP isteğinin hangi controller metoduna karşılık geldiğini belirleyen bileşen.

HandlerAdapter — `HandlerMapping`'in bulduğu metodu, doğru parametrelerle çağıran bileşen.

Embedded servlet container — Uygulamanın kendi içine gömülü olarak taşıdığı, harici kurulum gerektirmeyen sunucu (bu projede embedded Tomcat).

Spring WebFlux — Spring MVC'nin reactive/non-blocking alternatifi; Project Reactor üzerine kuruludur.

Ek: Mini Proje — Aynı Veri, İki Controller

Bu projenin kendi mimarisinde HTML sayfaları ve (henüz yazılmamış) bir JSON API aynı verinin farklı sunumları olabilir. Bunu küçük ölçekte kuruyoruz -- tek bir "servis", iki controller:

```
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` ve `ProductApiController`, aynı `ProductCatalogService`'i kullanıyor ama biri `Model` dolduruyor (View'a), diğeri veriyi doğrudan döndürüyor (Jackson'a). `ProductCatalogDemo`, gerçek bir DispatcherServlet olmadan, her iki controller'ı **doğrudan metot çağrısıyla** çalıştırıyor -- `Model`'in gerçek bir implementasyonu olan `ExtendedModelMap`'i elle oluşturup `page(model)`'e geçiriyoruz, tıpkı DispatcherServlet'in her istekte arka planda yaptığı gibi.

Ek: Mini Proje — Çok Controller'lı İstek Yönlendirme Simülasyonu

Son mini proje, "DispatcherServlet: Front Controller Deseni" bölümündeki `FrontControllerSimulationExample`'ı bir adım ileri taşıyor -- gerçek bir uygulamada tek değil, **onlarca** `@Controller` bean'i vardır; DispatcherServlet hepsini tek bir registry'de birleştirir:

```
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
    }
}
```

`RequestRouterSimulation.register(...)`, her çağrıldığında yeni bir handler nesnesinin metotlarını tarayıp aynı `registry`'ye ekliyor -- `HomeHandlers` ve `CartHandlers` birbirinden habersiz, ama `dispatch(...)` ikisini de aynı yerden bulabiliyor. Bu, component scanning'in birden fazla `@Component` sınıfını tek bir container'da toplaması ile aynı fikrin, HTTP routing tarafındaki karşılığı.

> Bu simülasyon, yalnızca tam path eşleşmesi yapıyor -- gerçek `HandlerMapping`'in yaptığı path variable çözümlemesi (`/cart/{id}` gibi), HTTP metodu ayrımı (aynı path'e hem `GET` hem `POST` tanımlanabilmesi) ya da content negotiation burada yok. Bunları bir sonraki konuda (Request Mapping & HTTP Methods) gerçek Spring annotation'larıyla göreceğiz.