Spring MVC'de Test Yazmak

MockMvc ve @WebMvcTest ile web katmanı testleri; @MockitoBean ile bağımlılıkları sahteleme; model(), view() ve jsonPath() ile doğrulama; request body, path variable /query param, validation hataları ve multipart dosya yükleme testleri; bu projenin gerçek HomeController'ı ve TopicController'ı için gerçek testler.

İleri 45 dk

Son güncelleme: 20 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/spring-mvc-testing

---

Spring MVC'de Test Yazmak

Bu, Spring MVC kategorisinin son dersi -- ve bir bakıma hepsini bir araya getiriyor. Şu ana kadar `@Controller` / `@RestController`, `@RequestBody` / `ResponseEntity`, `@Valid` / `ProblemDetail`, Thymeleaf view'ları, `HandlerInterceptor` /CORS/multipart, ve DTO/pagination/idempotency desenlerini yazdık -- ama hiçbirini gerçekten **çalıştırıp doğrulamadık**. Bu ders, `MockMvc` ve `@WebMvcTest` ile bu kodun gerçekten söylediğini yaptığını, gerçek bir sunucu ayağa kaldırmadan, hızlı ve tekrarlanabilir bir şekilde nasıl kanıtlayacağımızı ele alıyor.

Spring MVC'de Test Katmanları Nedir?

Bir Spring MVC uygulamasını test etmenin tek bir yolu yok -- amaca göre değişen birkaç katman var:

```
// Üç farklı test, üç farklı hız/gerçekçilik dengesi:
// 1) Saf birim testi: yeni TopicController(...).show(...) -- Spring hiç yok.
// 2) Slice testi: @WebMvcTest + MockMvc -- yalnızca web katmanı yüklü.
// 3) Entegrasyon testi: @SpringBootTest -- gerçek uygulama, gerçek DB (ya da test
container'ı).
```

Bu ders esas olarak ortadaki katmana odaklanıyor: `MockMvc` ile `@WebMvcTest`. Saf birim testi çok hızlı ama HTTP'nin kendisini (path matching, header'lar, serialization) hiç doğrulamaz; `@SpringBootTest` gerçekçi ama yavaş ve bir veritabanı gerektirir. `@WebMvcTest`, ikisi arasında -- gerçek HTTP isteği işleme mekaniğini, gerçek bir sunucu ya da veritabanı olmadan test eder.

Neden Var?

Bir controller'ı elle (`curl` ile ya da tarayıcıdan) test etmek, her değişiklikte tekrar tekrar yapılması gereken, unutulması kolay, otomasyona uygun olmayan bir iştir. `spring-mvc-fundamentals` dersinden bu yana yazdığımız her controller -- path matching, model attribute'ları, JSON serileştirme, validation, hata gövdeleri -- otomatik olarak, her kod değişikliğinde yeniden doğrulanabilir olmalı. `MockMvc`, bunu gerçek bir HTTP sunucusu açmadan (soket yok, port yok) yapmayı mümkün kılıyor -- bu da testleri hem hızlı hem de CI ortamında güvenilir kılıyor.

Tarihçe

Spring Test MVC, başlangıçta Spring Framework'ün ana gövdesinin dışında, ayrı bir `spring-test-mvc` projesi olarak (2012 civarı) başladı; `MockMvc` ve `andExpect` zincirleme API'si buradan geldi. Spring 3.2 ile bu proje Spring Framework'ün kendisine (`spring-test` modülüne) taşındı. Spring Boot 1.4 (2016), `@WebMvcTest` ve kardeşi `@DataJpaTest` gibi "slice test" annotation'larını tanıttı -- amaç, tüm `ApplicationContext`'i değil, testin gerçekten ihtiyaç duyduğu dilimi yüklemekti. `@MockBean`, uzun süre bu dilimlerdeki eksik bağımlılıkları doldurmanın standart yoluydu; Spring Boot 3.4 (2024) ile deprecated edildi ve yerini, Spring Framework'ün kendi test altyapısına taşınan `@MockitoBean`'e bıraktı -- bu proje Spring Boot 4.1.0 kullandığı için burada yalnızca `@MockitoBean` kullanıyoruz.

Unit Test vs Slice Test vs Integration Test: @WebMvcTest Nerede Durur?

`@WebMvcTest`, `spring-mvc-fundamentals` dersinin "Bu Projenin Kendi Controller'ları: Gerçek Bir Spring MVC Örneği" bölümünde gördüğümüz gerçek `HandlerMapping` / `HandlerAdapter` / `ViewResolver` mekanizmasını, DispatcherServlet'in front controller deseniyle birlikte gerçekten çalıştırır -- ama `@Service` / `@Repository` katmanını ve gerçek bir veritabanı bağlantısını YÜKLEMEZ. Bu, onu üç seçenek arasında bilinçli bir orta nokta yapar: saf birim testinden daha gerçekçi (gerçek HTTP request/response mekaniği çalışır), tam `@SpringBootTest`'ten daha hızlı (veritabanı yok, tüm bean'ler yok). Bu dersin geri kalanı, tam olarak bu orta noktaya -- `@WebMvcTest` ve `MockMvc`'ye -- odaklanıyor.

@WebMvcTest ve MockMvc: Yalnızca Web Katmanını Yüklemek

`@WebMvcTest`'in ne yüklediğini, ne yüklemediğini somut bir örnekle görelim:

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.stereotype.Controller;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.web.bind.annotation.GetMapping;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// @WebMvcTest loads a SLICE of the application, not the whole thing: DispatcherServlet,
// HandlerMapping/HandlerAdapter, message converters, the given controller (and other
// @Controller/@ControllerAdvice/@Converter/HandlerInterceptor/WebMvcConfigurer beans) --
// but NOT @Service/@Repository/@Component beans, and NOT a real database connection.
// This is a "run via `mvn test`" example, not a plain main() program -- @WebMvcTest
// needs JUnit's test runner and a Spring TestContext to do its work.
@WebMvcTest(WebMvcTestSliceExample.PingController.class)
class WebMvcTestSliceExample {

    // A tiny controller defined right here, just to keep this example self-contained --
    // in a real test this would be an existing @Controller/@RestController class.
    @Controller
    static class PingController {
        @GetMapping("/ping")
        @org.springframework.web.bind.annotation.ResponseBody
        String ping() {
            return "pong";
        }
    }

    private final MockMvc mockMvc;

    WebMvcTestSliceExample(MockMvc mockMvc) {
        // MockMvc is one of the few beans @WebMvcTest auto-configures and lets you
        // inject directly -- no manual setup needed.
        this.mockMvc = mockMvc;
    }

    @Test
    void pingReturnsPong() throws Exception {
        mockMvc.perform(get("/ping"))
                .andExpect(status().isOk());
        // If PingController depended on a @Service, this test would fail at context
        // startup with "no qualifying bean" -- @WebMvcTest deliberately does NOT wire
        // up @Service/@Repository beans; see "Mocking Dependencies with @MockitoBean"
        // for how to supply the ones a real controller needs.
    }
}
```

`@WebMvcTest(PingController.class)`, `DispatcherServlet`, mesaj converter'ları, ve belirtilen controller'ı (bir de varsa `@ControllerAdvice` / `HandlerInterceptor` / `WebMvcConfigurer` bean'lerini) yükler -- ama bir `@Service` bağımlılığı olsaydı, context başlatma anında "no qualifying bean" hatasıyla patlardı. `MockMvc`, bu daraltılmış context'in otomatik olarak inject edebildiği birkaç bean'den biri.

İlk MockMvc Testi: perform, andExpect, status()

`MockMvc`'yi, hatta bir Spring context'i bile beklemeden, en yalın haliyle görelim:

```
import org.springframework.stereotype.Controller;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.bind.annotation.GetMapping;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// MockMvcBuilders.standaloneSetup(...) builds MockMvc WITHOUT a Spring ApplicationContext
--
// it just wires the given controller(s) into a DispatcherServlet-like pipeline by hand.
// That's why this file, following the project's runnable-via-main() convention, can run
// as a plain main(); the real @WebMvcTest/@SpringBootTest tests (in the other examples
// below) require a JUnit runner + Spring TestContext.
public class FirstMockMvcTestExample {

    @Controller
    static class GreetingController {
        @GetMapping("/greeting")
        @org.springframework.web.bind.annotation.ResponseBody
        String greeting() {
            return "Hello, MockMvc!";
        }
    }

    public static void main(String[] args) throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new GreetingController()).build();

        // perform(): sends a fake HTTP request (no real socket is opened, no real port
        // is listened on -- everything runs inside the same JVM, using fake implementations
        // of the servlet API). andExpect(): chainable assertions; if one fails it throws
        // an AssertionError and the rest of the chain does not run.
        mockMvc.perform(get("/greeting"))
                .andExpect(status().isOk())
                .andExpect(content().string("Hello, MockMvc!"));

        System.out.println("All andExpect() assertions passed -- no real HTTP");
        System.out.println("server was ever opened.");

        // status(): verifies the HTTP status code -- via readable helper methods like
        // isOk() (200), isNotFound() (404), isBadRequest() (400).
        // content(): verifies the response body -- via string(), contentType(), json(),
etc.
    }
}
```

`MockMvcBuilders.standaloneSetup(...)`, verilen controller(lar)ı bir Spring `ApplicationContext` OLMADAN, elle bir mini pipeline'a bağlar -- bu yüzden bu örnek, projenin `main()` ile

çalıştırma kuralına uyarak plain `main()` ile çalışabiliyor. `perform(...)` sahte bir istek gönderir (gerçek soket açılmaz), `andExpect(...)` zincirlenebilir doğrulamalar yapar ve biri başarısız olursa `AssertionError` fırlatır.

@MockitoBean ile Bağımlılıkları Sahtelemek

`@WebMvcTest`'in `@Service`/`@Repository` bean'lerini yüklemediğini gördük -- peki controller bunlara gerçekten bağımlıysa ne olur?

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.stereotype.Controller;
import org.springframework.stereotype.Service;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// NOTE: @MockBean has been deprecated since Spring Boot 3.4 and was planned for removal
// in 4.0; since this project uses Spring Boot 4.1.0, ONLY @MockitoBean
// (org.springframework.test.context.bean.override.mockito.MockitoBean) is used here and
// in all subsequent examples. This is a JUnit test class -- it does not run via main(),
// it requires `mvn test`.
@WebMvcTest(MockitoBeanExample.GreeterController.class)
class MockitoBeanExample {

    // In a real application this would be a @Service; here it's defined inside the
    // file just to keep the example self-contained.
    interface GreetingService {
        String greetingFor(String name);
    }

    @Service
    static class RealGreetingService implements GreetingService {
        @Override
        public String greetingFor(String name) {
            throw new UnsupportedOperationException("The real implementation does not
matter here");
        }
    }

    @Controller
    static class GreeterController {
        private final GreetingService greetingService;

        GreeterController(GreetingService greetingService) {
            this.greetingService = greetingService;
        }
```

```
        @GetMapping("/greet")
        @org.springframework.web.bind.annotation.ResponseBody
        String greet() {
            return greetingService.greetingFor("Cem");
        }
    }

    @Autowired
    private MockMvc mockMvc;

    // @MockitoBean: adds a Mockito fake of type GreetingService to the context (or
    // replaces the real bean if one exists). Since @WebMvcTest doesn't load @Service
    // beans anyway, without this GreeterController's dependency would blow up at
    // context startup with a "no qualifying bean" error.
    @MockitoBean
    private GreetingService greetingService;

    @Test
    void greetUsesMockedService() throws Exception {
        when(greetingService.greetingFor("Cem")).thenReturn("Hello, Cem!");

        mockMvc.perform(get("/greet"))
                .andExpect(status().isOk())
                .andExpect(content().string("Hello, Cem!"));

        // RealGreetingService never ran -- only the value returned by the fake object
        // was used. This fully isolates the test from RealGreetingService's implementation
        // details (e.g. a database call).
    }
}
```

`@MockitoBean`, context'e ilgili türden bir Mockito sahtesi ekler (ya da varsa gerçek bean'in yerine geçirir) -- `GreeterController`'ın bağımlılığı olan `GreetingService`, bu olmadan context başlatma anında "no qualifying bean" hatasıyla patlardı. Not: `@MockBean` (Spring Boot 3.4'ten beri deprecated, bu projenin kullandığı 4.1.0'da kaldırıldı) yerine, burada ve bu dersin geri kalanında kesinlikle `@MockitoBean` kullanıyoruz.

Bu Projenin Kendi HomeController'ını Test Etmek: Gerçek Bir Örnek

Kurgu bir controller değil, `spring-mvc-fundamentals` dersinin "Bu Projenin Kendi Controller'ları: Gerçek Bir Spring MVC Örneği" bölümünde tanıttığımız gerçek `HomeController`'ı test edelim. `HomeController`'ın artık iki endpoint'i var: `/{lang:en|tr}` gerçek anasayfayı render ediyor, çıplak `/` ise varsayılan olarak `/en`'e 302 yönlendiren bir dil "negotiator"ı -- `Accept-Language` başlığını BİLİNÇLİ OLARAK dikkate almıyor, her ziyaretçi önce İngilizce siteyi görüyor, dil değişimini navbar'dan kendisi yapıyor:

```
import com.cdurgun.learning.controller.HomeController;
import com.cdurgun.learning.service.NavigationService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
```

```
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.test.web.servlet.MockMvc;

import java.util.List;

import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
redirectedUrl;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

// A real @WebMvcTest test for this project's ACTUAL HomeController -- not a made-up
// controller. HomeController's only dependency is NavigationService, so a single
// @MockitoBean is enough. This is a JUnit test class, run via `mvn test`. Since Phase 64,
// HomeController has had two endpoints: `/{lang:en|tr}` renders the real home page,
// while the bare `/` acts as a language "negotiator" -- it 302-redirects to `/en` by
// default (Phase 69: deliberately ignores the `Accept-Language` header, so a visitor
// with a Turkish browser/OS still lands on the English site first and switches
// manually via the navbar). The three tests below cover the index page, the
// no-header case, and the header-is-ignored case explicitly.
@WebMvcTest(HomeController.class)
class HomeControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockitoBean
    private NavigationService navigationService;

    @Test
    void indexReturnsIndexViewWithNavigationModel() throws Exception {
        // buildNavigation does not hit a real DB -- returning even an empty list is enough,
        // because what's being tested here is not NavigationService's BEHAVIOR, but
        // HOW HomeController CALLS it and puts the result into the model.
        when(navigationService.buildNavigation(org.mockito.ArgumentMatchers.any())).
thenReturn(List.of());

        // Language now depends directly on the URL path ({en|tr}), not on the request's
        // own default locale (usually Locale.getDefault() in a test environment) --
        // an `/en` request always produces the same result, in any environment.
        mockMvc.perform(get("/en"))
                .andExpect(status().isOk())
                .andExpect(view().name("index"))
                .andExpect(model().attributeExists("language"))
                .andExpect(model().attributeExists("nav"));

        verify(navigationService).buildNavigation(org.mockito.ArgumentMatchers.any());
    }

    @Test
    void bareRootRedirectsToDefaultLanguage() throws Exception {
        // A request with no Accept-Language header (like this test) redirects to EN --
        // see HomeController.resolveRootLanguage's default behavior. This method doesn't
        // touch any dependency, so there's no need to stub navigationService at all.
        mockMvc.perform(get("/"))
```

```
                .andExpect(status().isFound())
                .andExpect(redirectedUrl("/en"));
    }

    @Test
    void bareRootIgnoresAcceptLanguageHeader() throws Exception {
        // Phase 69: even a request that clearly prefers Turkish still lands on `/en` --
        // resolveRootLanguage no longer reads Accept-Language at all. This is a
        // deliberate product decision (default to English for everyone, let the
        // visitor switch languages themselves via the navbar's TR/EN buttons), not an
        // oversight -- this test exists specifically to pin that behavior down so a
        // future change doesn't silently reintroduce header-based negotiation.
        mockMvc.perform(get("/").header("Accept-Language", "tr-TR,tr;q=0.9"))
                .andExpect(status().isFound())
                .andExpect(redirectedUrl("/en"));
    }
}
```

`HomeController`'ın tek bağımlılığı `NavigationService` olduğu için tek bir `@MockitoBean` her üç testi de kapsıyor. `buildNavigation(...)`'ın döndürdüğü gerçek listeye (ya da içeriğine) hiç önem vermiyoruz -- burada test edilen şey `NavigationService`'in davranışı değil, `HomeController`'ın onu doğru çağırıp çağırmadığı ve model'e doğru attribute'ları koyup koymadığı. İlk test doğrudan `/en`'i hedefliyor -- dil artık ortamın varsayılan locale'ine değil, açık bir URL segmentine bağlı; ikinci test `Accept-Language` header'ı olmayan çıplak bir `/` isteğinin `/en`'de bittiğini doğruluyor; üçüncü test ise bir adım ileri gidip `Accept-Language: tr-TR,tr;q=0.9` header'ı GÖNDEREREK yine `/en` bekliyor -- bu bir eksiklik değil, negotiator'ın gerçek sözleşmesi; ileride sessizce geri dönmesin diye bilerek sabitlendi.

Model ve View Adını Doğrulamak: model(), view()

Klasik (JSON döndürmeyen) bir `@Controller` için `content()`'ten daha anlamlı olan iki matcher:

```
import org.springframework.stereotype.Controller;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

// model()/view(): looks not just at the HTTP status and body, but at which attributes
// the controller put into the Model and which view name it returned -- for a classic
// @Controller (rendered with Thymeleaf, not returning JSON), this is usually more
// meaningful than content(), because it verifies the controller's CONTRACT (which
// view, with what data) rather than the rendered HTML.
public class ModelAndViewAssertionExample {

    @Controller
    static class ProfileController {
        @GetMapping("/profile")
        String profile(Model model) {
            model.addAttribute("username", "cdurgun");
            model.addAttribute("topicCount", 25);
            return "profile";
        }
    }

    public static void main(String[] args) throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new ProfileController()).build();

        mockMvc.perform(get("/profile"))
                .andExpect(status().isOk())
                // view().name(): verifies the returned logical view name (NOT whether
                // the physical profile.html file was actually rendered --
                // standaloneSetup has no ViewResolver/template engine).
                .andExpect(view().name("profile"))
                // model().attribute(...): verifies the VALUE of an attribute.
                .andExpect(model().attribute("username", "cdurgun"))
                .andExpect(model().attribute("topicCount", 25))
                // model().attributeExists(...): verifies only its PRESENCE, not its value
--
                // useful when you don't care about testing the value itself.
                .andExpect(model().attributeExists("username", "topicCount"));

        System.out.println("Model and view assertions passed.");
    }
}
```

`view().name(...)`, dönen mantıksal view adını doğrular -- fiziksel HTML dosyasının render edilip edilmediğini değil (`standaloneSetup`'ta bir `ViewResolver`/template motoru yok). `model().attribute(...)` bir attribute'ın değerini, `model().attributeExists(...)` ise yalnızca varlığını doğrular.

@RestController Test Etmek: JSON Gövdesini jsonPath ile Doğrulamak

`@RestController`'larda view/model yok -- yanıt doğrudan JSON, ve onu doğrulamanın aracı `jsonPath(...)`:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// @RestControllers return JSON; view()/model() are meaningless here (there is no view).
// jsonPath(...) looks INSIDE the response body using a JSONPath expression -- it's
// useful for verifying individual fields rather than comparing the whole body as a
// string by hand (content().json(...)), especially when you want to ignore part of
// the body (e.g. a timestamp generated by the server).
public class JsonPathAssertionExample {

    record BookResponse(String title, String author, int pageCount, boolean available) {
    }

    @RestController
    static class BookController {
        @GetMapping("/books/{id}")
        BookResponse book(@PathVariable String id) {
            return new BookResponse("Effective Java", "Joshua Bloch", 412, true);
        }
    }

    public static void main(String[] args) throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new BookController()).build();

        mockMvc.perform(get("/books/1"))
                .andExpect(status().isOk())
                // "$.field": a field of the root object.
                .andExpect(jsonPath("$.title").value("Effective Java"))
                .andExpect(jsonPath("$.author").value("Joshua Bloch"))
                .andExpect(jsonPath("$.pageCount").value(412))
                .andExpect(jsonPath("$.available").value(true))
                // jsonPath(...).exists() / doesNotExist(): verifies the field's presence,
                // without looking at its value at all.
                .andExpect(jsonPath("$.isbn").doesNotExist());

        System.out.println("JSON body fields verified with jsonPath.");

        // Note: if a list were returned (e.g. List<BookResponse>), an array-indexing
        // expression like "$[0].title" could be used, and "$.length()" for the element
count.
    }
}
```

`jsonPath("$.alan")`, yanıt gövdesinin İÇİNE bakar -- tüm gövdeyi elle string karşılaştırmaya (`content().json(...)`) tercihen, tek tek alan doğrulamak, özellikle gövdenin bir kısmını (örn. sunucu tarafından üretilen bir zaman damgasını) görmezden gelmek istediğinizde kullanışlıdır. `jsonPath(...).exists()` / `doesNotExist()` ise bir alanın değerine hiç bakmadan varlığını doğrular.

Request Body Göndermek: content() ve contentType()

POST/PUT/PATCH gövdesi göndermek için iki parça gerekir: gövdenin kendisi ve tipi:

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// To send a POST/PUT/PATCH body, you need to provide the raw bytes/string via
// content(...), and the Content-Type header via contentType(...) -- if Content-Type
// isn't provided, Spring won't know which HttpMessageConverter to use and may
// reject the request (415 Unsupported Media Type).
public class RequestBodyTestExample {

    record CreateNoteRequest(String title, String body) {
    }

    record NoteResponse(long id, String title) {
    }

    @RestController
    static class NoteController {
        @PostMapping("/notes")
        NoteResponse create(@RequestBody CreateNoteRequest request) {
            return new NoteResponse(1L, request.title());
        }
    }

    public static void main(String[] args) throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new NoteController()).build();
        ObjectMapper objectMapper = new ObjectMapper();

        String requestJson = objectMapper.writeValueAsString(
                new CreateNoteRequest("Meeting Note", "Finish the Spring MVC testing
section"));

        mockMvc.perform(post("/notes")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(requestJson))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id").value(1))
                .andExpect(jsonPath("$.title").value("Meeting Note"));

        System.out.println("POST body sent and response verified.");

        // content(requestJson) is serialized here by hand with ObjectMapper -- in real
        // projects this is usually extracted into a small helper method (e.g.
        // asJsonString(Object)), since it repeats in almost every write test.
    }
}
```

`content(requestJson)` ham baytları/string'i, `contentType(...)` ise Content-Type header'ını verir -- Content-Type verilmezse Spring hangi `HttpMessageConverter`'ın kullanılacağını bilemez ve isteği reddedebilir (415 Unsupported Media Type). `ObjectMapper` ile elle serileştirme, gerçek projelerde genelde küçük bir yardımcı metoda çıkarılır çünkü hemen her yazma testinde tekrar eder.

Path Variable ve Query Parametrelerini Test Etmek

Path variable'lar URL'in içinde, query parametreleri ise `.param(...)` ile eklenir:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// Path variables are given inside the URL itself (like {id}); query parameters are
// added via .param(name, value) -- there's no need to build the actual ?page=0&size=10
// query string by hand, MockMvc sets it up for you.
public class PathVariableQueryParamTestExample {

    record TopicSummary(String slug, int page, int size, String difficulty) {
    }

    @RestController
    static class TopicSearchController {
        @GetMapping("/api/categories/{categorySlug}/topics")
        TopicSummary search(@PathVariable String categorySlug,
                            @RequestParam(defaultValue = "0") int page,
                            @RequestParam(defaultValue = "20") int size,
                            @RequestParam(required = false) String difficulty) {
            return new TopicSummary(categorySlug, page, size, difficulty);
        }
    }

    public static void main(String[] args) throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new TopicSearchController()).
build();

        mockMvc.perform(get("/api/categories/{categorySlug}/topics", "spring-mvc")
                        .param("page", "1")
                        .param("size", "5")
                        .param("difficulty", "ADVANCED"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.slug").value("spring-mvc"))
                .andExpect(jsonPath("$.page").value(1))
                .andExpect(jsonPath("$.size").value(5))
                .andExpect(jsonPath("$.difficulty").value("ADVANCED"));

        // Verify it also works without a difficulty -- since it's @RequestParam(required =
false),
        // it enters the controller as null instead of causing a 400.
        mockMvc.perform(get("/api/categories/{categorySlug}/topics", "spring-mvc"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.page").value(0))
                .andExpect(jsonPath("$.size").value(20))
                .andExpect(jsonPath("$.difficulty").doesNotExist());

        System.out.println("Path variable and query parameter tests passed.");
    }
}
```

`get("/api/categories/{categorySlug}/topics", "spring-mvc")` şeklindeki placeholder doldurma, `@PathVariable` ile eşleşir; `.param("page", "1")` gibi çağrılar ise `?page=1&size=5` formatındaki gerçek query string'i sizin için kurar. `@RequestParam(required = false)` olan bir parametre verilmediğinde, isteğin 400 değil, `null` ile controller'a girdiğini de ayrıca doğruluyoruz.

Validation Hatalarını Test Etmek: 400 ve ProblemDetail

`standaloneSetup(...)`, Bean Validation classpath'te olduğu için varsayılan bir validator kurar -- ama Validation & Exception Handling dersinin "Global Hata Yönetimi: @RestControllerAdvice" bölümünde gördüğümüz gibi, `@ControllerAdvice` sınıfları otomatik taranmaz:

```
import com.fasterxml.jackson.databind.ObjectMapper;
import jakarta.validation.Valid;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ProblemDetail;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import java.util.stream.Collectors;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// standaloneSetup(...) sets up a validator BY DEFAULT, because Bean Validation is on the
// classpath -- meaning @Valid works without an extra .setValidator(...) call. However,
// @ControllerAdvice classes are NOT scanned automatically: to get a proper error body
// (400 + ProblemDetail), you need to add the advice by hand via .setControllerAdvice(...)
// -- see the "@RestControllerAdvice: Global Error Handling" section in the "Validation
// and Exception Handling" lesson.
public class ValidationErrorTestExample {

    record CreateTopicRequest(@NotBlank String slug, @Min(1) int estimatedMinutes) {
    }

    @RestController
    static class TopicCreationController {
        @PostMapping("/api/topics")
        String create(@Valid @RequestBody CreateTopicRequest request) {
            return "created: " + request.slug();
        }
    }
```

```
    @RestControllerAdvice
    static class ValidationAdvice {
        @ExceptionHandler(MethodArgumentNotValidException.class)
        ProblemDetail handleValidation(MethodArgumentNotValidException e) {
            ProblemDetail problem = ProblemDetail.forStatusAndDetail(HttpStatus.
BAD_REQUEST, "Validation failed");
            problem.setProperty("errors", e.getBindingResult().getFieldErrors().stream()
                    .map(fe -> fe.getField() + ": " + fe.getDefaultMessage())
                    .collect(Collectors.toList()));
            return problem;
        }
    }

    public static void main(String[] args) throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new TopicCreationController())
                .setControllerAdvice(new ValidationAdvice())
                .build();
        ObjectMapper objectMapper = new ObjectMapper();

        // slug is empty AND estimatedMinutes is 0 -- both fields are in violation.
        String invalidJson = objectMapper.writeValueAsString(new CreateTopicRequest("", 0));

        mockMvc.perform(post("/api/topics")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(invalidJson))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.detail").value("Validation failed"))
                .andExpect(jsonPath("$.errors.length()").value(2));

        System.out.println("Invalid body was rejected with 400 + ProblemDetail.");

        // Compare with a valid request: same controller, same advice, different result.
        String validJson = objectMapper.writeValueAsString(new CreateTopicRequest("spring-
mvc-testing", 45));
        mockMvc.perform(post("/api/topics")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(validJson))
                .andExpect(status().isOk());

        System.out.println("A valid body, on the other hand, was accepted with 200.");
    }
}
```

Geçerli bir hata gövdesi almak için advice'ı `.setControllerAdvice(...)` ile elle eklemek gerekiyor. Buradaki `ValidationAdvice`, `MethodArgumentNotValidException`'ı yakalayıp aynı dersin "ProblemDetail: RFC 7807 ile Standart Hata Gövdesi" bölümündeki desenle bir `ProblemDetail` üretiyor -- geçersiz ve geçerli iki farklı istekle, aynı controller + aynı advice'ın iki farklı sonucunu karşılaştırıyoruz.

Multipart Dosya Yüklemeyi Test Etmek: MockMultipartFile

Advanced Spring MVC dersinin "Multipart File Upload: @RequestParam ile MultipartFile Almak" bölümündeki `MultipartUploadControllerExample`, main-scope olduğu için `MultipartFile`'ı elle implemente etmişti -- burada test scope'ta olduğumuz için gerçek `MockMultipartFile`'ı doğrudan kullanabiliyoruz:

```
import org.springframework.mock.web.MockMultipartFile;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.multipart;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// MockMultipartFile comes from spring-test itself (bundled with spring-boot-starter-test,
// test scope) -- unlike the MultipartUploadControllerExample in the "Advanced Spring MVC"
// lesson (which implemented MultipartFile by hand since that example was main-scope), here
// we're in test scope so we can use the real MockMultipartFile directly.
public class MultipartUploadTestExample {

    record UploadResult(String filename, long size, String contentType) {
    }

    @RestController
    static class UploadController {
        @PostMapping("/api/uploads")
        UploadResult upload(@RequestParam("file") MultipartFile file) {
            return new UploadResult(file.getOriginalFilename(), file.getSize(), file.
getContentType());
        }
    }

    public static void main(String[] args) throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new UploadController()).build();

        MockMultipartFile file = new MockMultipartFile(
                "file",                          // part name matching the @RequestParam
name
                "notes.txt",                      // original file name
                "text/plain",                     // content type
                "spring-mvc-testing notes".getBytes());

        // multipart(...): a special request builder that builds a multipart/form-data
        // body, instead of the usual get()/post().
        mockMvc.perform(multipart("/api/uploads").file(file))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.filename").value("notes.txt"))
                .andExpect(jsonPath("$.contentType").value("text/plain"))
                .andExpect(jsonPath("$.size").value(file.getSize()));

        System.out.println("Multipart file upload test passed.");

        // For scenarios like exceeding the size limit (see the "Multipart Size Limits"
        // section in the Advanced Spring MVC lesson), a @RestControllerAdvice that catches
        // MaxUploadSizeExceededException can be added, using the same pattern as in
        // ValidationErrorTestExample.
    }
}
```

`MockMultipartFile`, `spring-boot-starter-test` ile gelen (test scope) gerçek bir spring-test sınıfı. `multipart(...)`, normal `get()` / `post()` yerine, `multipart/form-data` gövdesi kuran özel bir request builder'dır -- `.file(file)` ile eklenen dosya, controller'daki `@RequestParam("file") MultipartFile` parametresiyle eşleşir. Boyut sınırı ihlali gibi senaryolar (bkz. aynı dersin "Multipart Yapılandırması ve Boyut Sınırları" bölümü), `ValidationErrorTestExample` 'daki desenin aynısıyla, ilgili exception'ı yakalayan bir advice eklenerek test edilebilir.

Best Practices

- **`@MockBean` yerine her zaman `@MockitoBean` kullan** -- bu projenin kullandığı Spring Boot 4.1.0'da `@MockBean` kaldırıldı; `@MockitoBean` aynı işi görür ve Spring Framework'ün kendi test altyapısının bir parçası (bkz. "@MockitoBean ile Bağımlılıkları Sahtelemek").
- **`@WebMvcTest`'i, gerçekten test etmek istediğin controller'a daralt** (`@WebMvcTest(HomeController.class)` gibi) -- boş bırakmak tüm controller'ları yükler ve testi yavaşlatır, ayrıca hangi bağımlılığın sahtelenmesi gerektiğini belirsizleştirir (bkz. "Bu Projenin Kendi HomeController'ını Test Etmek: Gerçek Bir Örnek").
- **JSON yanıtlarda tüm gövdeyi string karşılaştırmak yerine `jsonPath(...)` ile tek tek alan doğrula** -- gövde şekli küçük bir şekilde değiştiğinde (yeni bir alan eklendiğinde gibi) testin kırılmaz olmasını sağlar (bkz. "@RestController Test Etmek: JSON Gövdesini jsonPath ile Doğrulamak").
- **`standaloneSetup(...)` kullanırken `@ControllerAdvice`'ı elle eklemeyi unutma** -- aksi hâlde hata senaryoları, gerçek uygulamada göreceğiniz `ProblemDetail` yerine ham bir exception ile sonuçlanır (bkz. "Validation Hatalarını Test Etmek: 400 ve ProblemDetail").

Yaygın Hatalar

1. `@WebMvcTest` ile bir `@Service` bağımlılığını sahteleme (`@MockitoBean`) unutmak. Context, "no qualifying bean" hatasıyla başlatma anında patlar -- `@WebMvcTest`'in `@Service` / `@Repository` katmanını hiç yüklemediğini unutmak, bu dersteki en sık karşılaşılan hata (bkz. "@WebMvcTest ve MockMvc: Yalnızca Web Katmanını Yüklemek").

2. POST/PUT isteklerinde `contentType(...)` eklemeyi unutmak. Gövde `content(...)` ile verilse bile, Content-Type header'ı olmadan Spring hangi `HttpMessageConverter`'ın kullanılacağını bilemez ve istek 415 ile reddedilebilir (bkz. "Request Body Göndermek: content() ve contentType()").

3. `jsonPath(...)`'i, dizi mi nesne mi döndüğünü kontrol etmeden yazmak. Bir liste için `$.title` değil `$[0].title` gerekir -- yanlış ifade, alanın hiç bulunamamasıyla sonuçlanan kafa karıştırıcı bir hataya yol açar (bkz. "@RestController Test Etmek: JSON Gövdesini jsonPath ile Doğrulamak").

4. `standaloneSetup(...)` ile `@Valid`'in çalıştığını varsayıp, `@ControllerAdvice`'ı eklemeyi atlamak. Validator varsayılan olarak kurulur ve `MethodArgumentNotValidException` fırlatılır, ama bu exception'ı düzgün bir `ProblemDetail`'e çeviren advice elle eklenmediği sürece, test

beklenmedik bir 500 ile karşılaşır (bkz. "Validation Hatalarını Test Etmek: 400 ve ProblemDetail").

5. `@WebMvcTest`'te gerçek bir veritabanına erişmeye çalışmak. Bu dilim, kasıtlı olarak `@Repository` bean'lerini yüklemez -- bir repository'ye ihtiyaç duyan controller, o repository `@MockitoBean` ile sahtelenmediği sürece çalışmaz (bkz. "Bu Projenin Kendi HomeController'ını Test Etmek: Gerçek Bir Örnek").

Özet, Cheat Sheet ve Terimler Sözlüğü

Spring MVC'de test yazmak, üç katman arasında bilinçli bir seçim yapmakla başlıyor -- saf birim testi, `@WebMvcTest` slice testi, ya da tam `@SpringBootTest` entegrasyon testi. Öne çıkan noktalar:

- `@WebMvcTest`: yalnızca web katmanını (DispatcherServlet, controller'lar, converter'lar) yükleyen, `@Service` / `@Repository`'i hariç tutan bir slice test annotation'ı
- `MockMvc`: gerçek bir HTTP sunucusu açmadan sahte istekler gönderen test aracı
- `MockMvcBuilders.standaloneSetup(...)`: Spring context olmadan, elle bir controller pipeline'ı kuran alternatif kurulum
- `@MockitoBean`: context'e bir Mockito sahtesi ekleyen/gerçek bean'in yerine geçiren annotation (`@MockBean`'in yerini aldı)
- `perform()` / `andExpect()`: sırasıyla isteği gönderen ve zincirlenebilir doğrulamalar yapan MockMvc metotları
- `status()` / `view()` / `model()` / `jsonPath()` / `content()` / `header()`: farklı yanıt yönlerini (durum kodu, view adı, model attribute'ları, JSON alanları, gövde, header'lar) doğrulayan matcher aileleri
- `MockMultipartFile`: multipart/form-data testleri için gerçek bir spring-test sınıfı (test scope)

Hızlı referans:

```
@WebMvcTest(TopicController.class)
class TopicControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockitoBean
    private TopicRepository topicRepository;

    @Test
    void unknownSlugReturns404() throws Exception {
        when(topicRepository.findBySlugWithCategoryAndCourse("x"))
                .thenReturn(Optional.empty());

        mockMvc.perform(get("/en/topics/x"))
                .andExpect(status().isNotFound());
    }
}
```

Terimler Sözlüğü

`@WebMvcTest` — Yalnızca Spring MVC web katmanını yükleyen, `@Service` / `@Repository` bean'lerini hariç tutan bir Spring Boot test slice annotation'ı.

`MockMvc` — Gerçek bir sunucu/soket açmadan sahte HTTP istekleri gönderip yanıtları doğrulamayı sağlayan test aracı.

`standaloneSetup` — Bir Spring `ApplicationContext` olmadan, verilen controller'ları elle bir `MockMvc` pipeline'ına bağlayan kurulum yöntemi.

`@MockitoBean` — Test context'ine bir Mockito sahtesi ekleyen ya da gerçek bir bean'in yerine geçiren annotation; `@MockBean`'in yerini aldı.

`jsonPath` — Bir JSON yanıt gövdesinin belirli bir alanını, bir JSONPath ifadesiyle doğrulayan matcher.

`MockMultipartFile` — Multipart dosya yükleme testleri için kullanılan, `spring-test` kütüphanesinin sağladığı sahte dosya sınıfı.

Ek: Mini Proje — Bu Projenin Kendi TopicController'ı İçin Kapsamlı Bir Test Paketi

Bu dersteki tüm teknikleri, bu projenin gerçek `TopicController`'ı (altı bağımlılığın tamamı `@MockitoBean` ile sahtelenmiş) üzerinde birleştiriyoruz:

```
import com.cdurgun.learning.domain.Category;
import com.cdurgun.learning.domain.Course;
import com.cdurgun.learning.domain.Difficulty;
import com.cdurgun.learning.domain.Language;
import com.cdurgun.learning.domain.Topic;
import com.cdurgun.learning.domain.TopicTranslation;
```

```
// This project's real entities (Topic, Category, Course, TopicTranslation) use Lombok
// @Builder -- they can be used directly to produce readable "fixtures" (test data)
// without needing an extra test library. This class builds a SELF-CONSISTENT Topic
// tree, complete with a TR/EN translation and all its ManyToOne relationships -- the
// TopicControllerWebMvcTest below uses these helpers.
public class TopicTestFixtures {

    public static Course sampleCourse() {
        return Course.builder()
                .id(1L)
                .name("Java")
                .slug("java")
                .sortOrder(1)
                .build();
    }

    public static Category sampleCategory(Course course) {
        return Category.builder()
                .id(1L)
                .course(course)
                .name("Spring MVC")
                .slug("spring-mvc")
                .sortOrder(1)
                .build();
    }

    public static Topic sampleTopic(Category category) {
        return Topic.builder()
                .id(9L)
                .category(category)
                .slug("spring-mvc-testing")
                .difficulty(Difficulty.ADVANCED)
                .estimatedMinutes(40)
                .sortOrder(9)
                .build();
    }

    public static TopicTranslation sampleTranslation(Topic topic, Language language,
boolean published) {
        return TopicTranslation.builder()
                .id(language == Language.TR ? 91L : 92L)
                .topic(topic)
                .language(language)
                .title(language == Language.TR ? "Spring MVC'de Test Yazmak" : "Testing in
Spring MVC")
                .summary(language == Language.TR ? "MockMvc ve @WebMvcTest ile web katmanı
testleri." : "Web layer testing with MockMvc and @WebMvcTest.")
                .published(published)
                .build();
    }

    public static void main(String[] args) {
        Course course = sampleCourse();
        Category category = sampleCategory(course);
        Topic topic = sampleTopic(category);
        TopicTranslation translation = sampleTranslation(topic, Language.TR, true);
```

```
        // Verify that the chained relationship was actually established -- producing
        // entities with a builder doesn't eliminate the need to wire the references
        // between them BY HAND.
        System.out.println(translation.getTopic().getCategory().getCourse().getName());
        // Java
        System.out.println(translation.getTopic().getSlug() + " -> " + translation.
getTitle());
        // spring-mvc-testing -> Spring MVC'de Test Yazmak
    }
}
```

```
import com.cdurgun.learning.controller.TopicController;
import com.cdurgun.learning.domain.Category;
import com.cdurgun.learning.domain.Course;
import com.cdurgun.learning.domain.Language;
import com.cdurgun.learning.domain.Topic;
import com.cdurgun.learning.domain.TopicTranslation;
import com.cdurgun.learning.repository.TopicRepository;
import com.cdurgun.learning.repository.TopicTranslationRepository;
import com.cdurgun.learning.service.ContentResolver;
import com.cdurgun.learning.service.MarkdownService;
import com.cdurgun.learning.service.NavigationService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.context.MessageSource;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.test.web.servlet.MockMvc;

import java.util.List;
import java.util.Optional;

import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
redirectedUrl;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

// @WebMvcTest for this project's ACTUAL TopicController -- ALL 6 dependencies are
// mocked with @MockitoBean (see TopicController's constructor). TopicTestFixtures is
// used to produce this test's fixtures (Course/Category/Topic/TopicTranslation).
// Since Phase 64, the controller has had two mappings: `/{lang:en|tr}/topics/{slug}`
// (the real content) and `/topics/{slug}`, which permanently redirects legacy `?lang=`
// URLs (part of the move to a path-based URL structure for SEO reasons) -- since both
// live in the same controller, a single @WebMvcTest slice can test both.
@WebMvcTest(TopicController.class)
class TopicControllerWebMvcTest {

    @Autowired
    private MockMvc mockMvc;

    @MockitoBean
    private TopicRepository topicRepository;
```

```
    @MockitoBean
    private TopicTranslationRepository topicTranslationRepository;
    @MockitoBean
    private ContentResolver contentResolver;
    @MockitoBean
    private MarkdownService markdownService;
    @MockitoBean
    private NavigationService navigationService;
    @MockitoBean
    private MessageSource messageSource;

    @Test
    void unknownSlugReturns404() throws Exception {
        when(topicRepository.findBySlugWithCategoryAndCourse("does-not-exist"))
                .thenReturn(Optional.empty());

        // See in the controller code: ResponseStatusException(HttpStatus.NOT_FOUND, ...) --
        // even without a real @ControllerAdvice, Spring's default exception resolution
        // converts ResponseStatusException to the correct HTTP status.
        mockMvc.perform(get("/en/topics/does-not-exist"))
                .andExpect(status().isNotFound());
    }

    @Test
    void legacyQueryParamUrlRedirectsPermanentlyToPathBasedUrl() throws Exception {
        // Pre-Phase-64 `/topics/{slug}?lang=..` URLs no longer render, they now
        // 301 (permanently) redirect to the new `/{lang}/topics/{slug}` address -- since
        // this controller method doesn't touch any repository/service, there's no need
        // to set up a mock here. `redirectedUrl(...)` is a matcher that directly verifies
        // the Location header.
        mockMvc.perform(get("/topics/spring-mvc-testing").param("lang", "tr"))
                .andExpect(status().isMovedPermanently())
                .andExpect(redirectedUrl("/tr/topics/spring-mvc-testing"));
    }

    @Test
    void publishedTopicRendersWithContent() throws Exception {
        Course course = TopicTestFixtures.sampleCourse();
        Category category = TopicTestFixtures.sampleCategory(course);
        Topic topic = TopicTestFixtures.sampleTopic(category);
        TopicTranslation trTranslation = TopicTestFixtures.sampleTranslation(topic,
Language.TR, true);

        when(topicRepository.findBySlugWithCategoryAndCourse("spring-mvc-testing"))
                .thenReturn(Optional.of(topic));
        when(topicTranslationRepository.findByTopicIdAndLanguage(topic.getId(), Language.
TR))
                .thenReturn(Optional.of(trTranslation));
        when(topicTranslationRepository.findByTopicIdAndLanguage(topic.getId(), Language.
EN))
                .thenReturn(Optional.empty());
        when(contentResolver.resolve(eq("spring-mvc-testing"), eq(Language.TR)))
                .thenReturn(Optional.of("# Spring MVC'de Test Yazmak\n\nIcerik burada."));
        when(markdownService.render(any(), eq("spring-mvc-testing")))
                .thenReturn(new MarkdownService.MarkdownRenderResult("<h1>Spring MVC'de
Test Yazmak</h1>", List.of()));
        when(navigationService.buildNavigation(Language.TR)).thenReturn(List.of());
        when(navigationService.buildCourseSequence(course.getId(), Language.TR)).thenReturn
```

```
(List.of());

        // Language is now part of the URL path instead of `?lang=` -- a `/tr/...`
        // request targets Language.TR directly, independent of the environment/default
        // locale.
        mockMvc.perform(get("/tr/topics/spring-mvc-testing"))
                .andExpect(status().isOk())
                .andExpect(view().name("topic"))
                .andExpect(model().attribute("contentAvailable", true))
                .andExpect(model().attribute("otherLanguageAvailable", false));

        // Note: every value mocked here is of the same type that TopicController gets
        // from real services in PRODUCTION (a real Topic, a real MarkdownRenderResult
        // record) -- so templates/topic.html renders normally, as if it were handling
        // a real request.
    }
}
```

`TopicTestFixtures`, gerçek `Course` / `Category` / `Topic` / `TopicTranslation` entity'lerini (hepsi Lombok `@Builder` kullanıyor) tutarlı bir ağaç olarak kuran yardımcı metotlar sağlıyor. `TopicControllerWebMvcTest` üç senaryoyu kapsıyor: bilinmeyen bir slug için 404, eski query-parametreli URL'den (`/topics/{slug}?lang=..`) yeni path-bazlı adrese 301 yönlendirme, ve tam yayınlanmış bir konu için gerçek `topic.html` template'i üzerinden 200 -- son senaryoda mock'lanan her değer, controller'ın production'da gerçek servislerden aldığı değerlerle aynı tipte (gerçek `Topic`, gerçek `MarkdownService.MarkdownRenderResult`), bu yüzden template gerçek bir isteği işliyormuş gibi normal şekilde render ediliyor.

Ek: Mini Proje — Bir Interceptor'ı MockMvc ile Test Etmek

Son mini proje, Advanced Spring MVC dersinin "HandlerInterceptor Arayüzü: preHandle, postHandle, afterCompletion" bölümündeki yaşam döngüsünü, `WebMvcConfigurer: Interceptor'ı Kaydetmek` bölümündeki gibi bir konfigürasyon sınıfı hiç yazmadan, doğrudan `MockMvc` ile izole test ediyor:

```
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;

// A small, realistic HandlerInterceptor -- it uses the same lifecycle (preHandle/
// postHandle/afterCompletion) as HandlerInterceptorLifecycleExample in the "Advanced
// Spring MVC" lesson, but since the goal here is to test the interceptor on its own,
// it's kept as simple as possible: it adds an X-Response-Time-Ms header to every request.
public class TimingInterceptorForTest implements HandlerInterceptor {

    private static final String START_ATTRIBUTE = "requestStartNanos";

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) {
        request.setAttribute(START_ATTRIBUTE, System.nanoTime());
        return true;
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
                                Object handler, Exception ex) {
        Object startValue = request.getAttribute(START_ATTRIBUTE);
        if (startValue instanceof Long startNanos) {
            long elapsedMs = (System.nanoTime() - startNanos) / 1_000_000;
            response.setHeader("X-Response-Time-Ms", String.valueOf(elapsedMs));
        }
        // Note: adding the header in afterCompletion instead of postHandle is intentional
--
        // afterCompletion always runs, EVEN IF the handler throws an exception, whereas
        // postHandle does not (see the "HandlerInterceptor Lifecycle" section in the
        // "Advanced Spring MVC" lesson).
    }
}
```

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.header;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// standaloneSetup(...).addInterceptors(...): attaches an interceptor directly to
// MockMvc, without ever writing the WebMvcConfigurer that registers it (as in
// InterceptorRegistrationExample) -- ideal for testing the interceptor in ISOLATION,
// since it doesn't pull in the whole application's configuration (path patterns,
// other interceptors).
public class TimingInterceptorMockMvcTest {

    @RestController
    static class PingController {
        @GetMapping("/ping")
        String ping() {
            return "pong";
        }
    }

    public static void main(String[] args) throws Exception {
        MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new PingController())
                .addInterceptors(new TimingInterceptorForTest())
                .build();

        mockMvc.perform(get("/ping"))
                .andExpect(status().isOk())
                // header().exists(...): verifies that this header is present in the
response --
                // since its value will change on every run (the real elapsed time),
exists()
                // is a more accurate choice here than looking for an exact match with
                // string(...).
                .andExpect(header().exists("X-Response-Time-Ms"));

        System.out.println("Interceptor tested in isolation with standaloneSetup.");

        // addInterceptors(...) applies to ALL paths here -- if you want to restrict it to
        // a specific path pattern, as in the real application, MockMvcBuilders' standalone
        // API doesn't support that directly; in that case you'd need to test the
        // interceptor's own internal path check (if it has one).
    }
}
```

`TimingInterceptorForTest`, her isteğe `X-Response-Time-Ms` header'ı ekleyen küçük, gerçekçi bir `HandlerInterceptor`. `TimingInterceptorMockMvcTest`, `standaloneSetup(...).addInterceptors(...)` ile bu interceptor'ı doğrudan `MockMvc`'ye takıyor -- tüm uygulamanın konfigürasyonunu (path pattern'ler, diğer interceptor'lar) hiç devreye sokmadan, interceptor'ı KENDİ BAŞINA doğruluyor.