Testing in Spring MVC

Web layer tests with MockMvc and @WebMvcTest; faking dependencies with @MockitoBean; verifying with model(), view(), and jsonPath(); testing request bodies, path variables/query params, validation errors, and multipart uploads; real tests of this project's own HomeController and TopicController.

Advanced 45 min

Last updated: August 20, 2026
Read online: https://www.learnforgex.com/en/topics/spring-mvc-testing

---

Testing in Spring MVC

This is the last lesson in the Spring MVC category -- and in a way, it ties all the others together. So far we've written `@Controller` / `@RestController`s, `@RequestBody` / `ResponseEntity`, `@Valid` / `ProblemDetail`, Thymeleaf views, `HandlerInterceptor` /CORS/multipart, and DTO /pagination/ idempotency patterns -- but we never actually **ran and verified** any of it. This lesson covers how `MockMvc` and `@WebMvcTest` let us prove that code actually does what it claims, quickly and repeatably, without spinning up a real server.

What Are Test Layers in Spring MVC?

There's no single way to test a Spring MVC application -- there are a few layers, each with a different purpose:

```
// Three different tests, three different speed/realism trade-offs:
// 1) Pure unit test: new TopicController(...).show(...) -- no Spring at all.
// 2) Slice test: @WebMvcTest + MockMvc -- only the web layer is loaded.
// 3) Integration test: @SpringBootTest -- the real app, a real DB (or test container).
```

This lesson focuses mainly on the middle layer: `MockMvc` with `@WebMvcTest`. A pure unit test is very fast but never verifies HTTP itself (path matching, headers, serialization); `@SpringBootTest` is realistic but slow and needs a database. `@WebMvcTest` sits between the two -- it tests real HTTP request-handling mechanics without a real server or database.

Why Does It Exist?

Testing a controller by hand (with `curl` or a browser) is work that has to be repeated on every change, easy to forget, and doesn't scale to automation. Every controller we've

written since `spring-mvc-fundamentals` -- path matching, model attributes, JSON serialization, validation, error bodies -- should be automatically re-verifiable on every code change. `MockMvc` makes that possible without opening a real HTTP server (no socket, no port) -- which keeps tests both fast and reliable in a CI environment.

History

Spring Test MVC started out (around 2012) as a separate `spring-test-mvc` project, outside Spring Framework's main codebase; that's where `MockMvc` and its `andExpect` chaining API came from. Spring 3.2 moved that project into Spring Framework itself (the `spring-test` module). Spring Boot 1.4 (2016) introduced "slice test" annotations like `@WebMvcTest` and its sibling `@DataJpaTest` -- the goal being to load only the slice of the `ApplicationContext` a test actually needs, not the whole thing. `@MockBean` was, for a long time, the standard way to fill in missing dependencies in these slices; Spring Boot 3.4 (2024) deprecated it in favor of `@MockitoBean`, which moved into Spring Framework's own test infrastructure -- since this project runs Spring Boot 4.1.0, we use only `@MockitoBean` here.

Unit Test vs. Slice Test vs. Integration Test: Where Does @WebMvcTest Sit?

`@WebMvcTest` actually runs the real `HandlerMapping` / `HandlerAdapter` / `ViewResolver` machinery we saw in spring-mvc-fundamentals' "This Project's Own Controllers: A Real Spring MVC Example" section, together with DispatcherServlet's front controller pattern -- but it does NOT load the `@Service` / `@Repository` layer or a real database connection. That makes it a deliberate middle ground among the three options: more realistic than a pure unit test (real HTTP request/response mechanics run), faster than a full `@SpringBootTest` (no database, not every bean). The rest of this lesson focuses on exactly that middle ground: `@WebMvcTest` and `MockMvc`.

@WebMvcTest and MockMvc: Loading Only the Web Layer

Let's see concretely what `@WebMvcTest` does and doesn't load:

```
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)` loads `DispatcherServlet`, message converters, and the given controller (plus any `@ControllerAdvice` / `HandlerInterceptor` / `WebMvcConfigurer` beans) -- but if it had a `@Service` dependency, context startup would fail with "no qualifying bean". `MockMvc` is one of the few beans this narrowed-down context can inject automatically.

Your First MockMvc Test: perform, andExpect, status()

Let's see `MockMvc` at its simplest, without even needing a Spring context:

```
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(...)` wires the given controller(s) into a mini pipeline by hand, WITHOUT a Spring `ApplicationContext` -- that's why this example can run as a plain `main()`, following this project's run-with-`main()` convention. `perform(...)` sends a fake

request (no real socket opens), and `andExpect(...)` runs chainable assertions, throwing an `AssertionError` if one fails.

Faking Dependencies with @MockitoBean

We saw that `@WebMvcTest` doesn't load `@Service` / `@Repository` beans -- so what happens when a controller genuinely depends on one?

```
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` adds a Mockito mock of the given type to the context (or replaces a real bean if one exists) -- without it, `GreeterController`'s `GreetingService` dependency would fail context startup with "no qualifying bean". Note: instead of `@MockBean` (deprecated since Spring Boot 3.4, removed in the 4.1.0 this project runs), we use `@MockitoBean` exclusively here and throughout the rest of this lesson.

Testing This Project's Own HomeController: A Real Example

Not a made-up controller -- let's test the real `HomeController` we introduced in spring-mvc-fundamentals' "This Project's Own Controllers: A Real Spring MVC Example" section. `HomeController` now has two endpoints: `/{lang:en|tr}` renders the actual home page, while a bare `/` is a language "negotiator" that 302-redirects to `/en` by default -- it deliberately ignores the `Accept-Language` header, so every visitor lands on the English site first and switches languages themselves via the navbar:

```
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`'s only dependency is `NavigationService`, so a single `@MockitoBean` covers all three tests. We don't care about the actual list `buildNavigation(...)` returns -- what's being tested here isn't `NavigationService`'s behavior, it's whether `HomeController` calls it correctly and puts the right attributes into the model. The index test hits `/en` directly instead of relying on an ambient default locale -- the language is now an explicit URL segment, not implicit test-environment state; the second test confirms a bare `/` request with no `Accept-Language` header lands on `/en`; and the third test goes further, sending a `Accept-Language: tr-TR,tr;q=0.9` header and still asserting `/en` -- that's not a gap in the negotiator, it's the negotiator's actual contract, pinned down so it can't silently regress.

Verifying the Model and View Name: model(), view()

Two matchers that matter more than `content()` for a classic (non-JSON) `@Controller`:

```
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(...)` verifies the logical view name that's returned -- not whether the physical HTML file got rendered (`standaloneSetup` has no `ViewResolver`/template engine at all). `model().attribute(...)` verifies an attribute's value, while `model().attributeExists(...)` verifies only its presence.

Testing a @RestController: Verifying a JSON Body with jsonPath

`@RestController`s have no view/model -- the response is straight JSON, and the tool for verifying it is `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("$.field")` looks INSIDE the response body -- verifying fields one at a time, rather than comparing the whole body as a string (`content().json(...)`), is useful especially when you want to ignore part of the body (like a server-generated timestamp). `jsonPath(...).exists()` / `doesNotExist()` verify a field's presence without looking at its value at all.

Sending a Request Body: content() and contentType()

Sending a POST/PUT/PATCH body takes two pieces: the body itself, and its type:

```
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)` supplies the raw bytes/string, `contentType(...)` supplies the Content-Type header -- without a Content-Type, Spring can't tell which `HttpMessageConverter` to use, and the request can be rejected (415 Unsupported Media Type). Hand-serializing with `ObjectMapper` is usually pulled out into a small helper method in real projects, since it repeats in almost every write test.

Testing Path Variables and Query Parameters

Path variables live inside the URL itself; query parameters are added with `.param(...)`:

```
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.");
    }
}
```

The placeholder-filling in `get("/api/categories/{categorySlug}/topics", "spring-mvc")` matches `@PathVariable`; calls like `.param("page", "1")` build the actual query string (`?page=1&size=5`) for you. We also verify that an `@RequestParam(required = false)` parameter that isn't supplied reaches the controller as `null`, not as a 400.

Testing Validation Errors: 400 and ProblemDetail

`standaloneSetup(...)` sets up a default validator automatically, since Bean Validation is on the classpath -- but as we saw in Validation & Exception Handling's "Global Error Handling: @RestControllerAdvice" section, `@ControllerAdvice` classes aren't scanned automatically:

```
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.");
    }
}
```

Getting a proper error body requires adding the advice by hand with `.setControllerAdvice(...)`. The `ValidationAdvice` here catches `MethodArgumentNotValidException` and builds a `ProblemDetail` using the same pattern from that lesson's "ProblemDetail: A Standard Error Body with RFC 7807" section -- with one invalid and one valid request, we compare two different outcomes from the same controller plus the same advice.

Testing a Multipart File Upload: MockMultipartFile

Advanced Spring MVC's `MultipartUploadControllerExample`, from the "Multipart File Upload: Taking a MultipartFile with @RequestParam" section, hand-implemented `MultipartFile` because it lives in main scope -- here, being in test scope, we can use the real `MockMultipartFile` directly:

```
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` is a real spring-test class that comes with `spring-boot-starter-test` (test scope). `multipart(...)` is a special request builder that builds a `multipart/form-data` body instead of the usual `get()` / `post()` -- the file added with `.file(file)` matches the controller's `@RequestParam("file") MultipartFile` parameter. Scenarios like exceeding a size limit (see that same lesson's "Multipart Configuration and Size Limits" section) can be tested the same way as `ValidationErrorTestExample`, by adding an advice that catches the relevant exception.

Best Practices

- **Always use `@MockitoBean` instead of `@MockBean`** -- this project's Spring Boot 4.1.0 removed `@MockBean`; `@MockitoBean` does the same job and lives in Spring Framework's own test infrastructure (see "Faking Dependencies with @MockitoBean").
- **Scope `@WebMvcTest` to the specific controller you're testing** (`@WebMvcTest(HomeController.class)`, not left empty) -- leaving it empty loads every controller and slows the test down, while also making it unclear which dependency needs mocking (see "Testing This Project's Own HomeController: A Real Example").
- **Verify JSON responses field-by-field with `jsonPath(...)` instead of comparing the whole body as a string** -- this keeps the test from breaking when the body shape changes slightly, like when a new field is added (see "Testing a @RestController: Verifying a JSON Body with jsonPath").
- **Don't forget to add `@ControllerAdvice` by hand when using `standaloneSetup(...)`** -- otherwise error scenarios end up as a raw exception instead of the `ProblemDetail` you'd see in the real application (see "Testing Validation Errors: 400 and ProblemDetail").

Common Mistakes

1. Forgetting to mock a `@Service` dependency (`@MockitoBean`) with `@WebMvcTest`. The context fails at startup with "no qualifying bean" -- forgetting that `@WebMvcTest` never loads the `@Service` / `@Repository` layer is the single most common mistake in this lesson (see "@WebMvcTest and MockMvc: Loading Only the Web Layer").

2. Forgetting `contentType(...)` on POST/PUT requests. Even with a body supplied via `content(...)`, without a Content-Type header Spring can't tell which `HttpMessageConverter` to use, and the request can be rejected with 415 (see "Sending a Request Body: content() and contentType()").

3. Writing `jsonPath(...)` without checking whether the response is an array or an object. A list needs `$[0].title`, not `$.title` -- the wrong expression leads to a confusing failure where the field simply can't be found (see "Testing a @RestController: Verifying a JSON Body with jsonPath").

4. Assuming `@Valid` works with `standaloneSetup(...)` and skipping the `@ControllerAdvice`. The validator is set up by default and `MethodArgumentNotValidException`

does get thrown, but unless an advice that turns it into a proper `ProblemDetail` is added by hand, the test hits an unexpected 500 instead (see "Testing Validation Errors: 400 and ProblemDetail").

5. Trying to reach a real database inside a `@WebMvcTest`. This slice deliberately doesn't load `@Repository` beans -- a controller that needs a repository won't work unless that repository is mocked with `@MockitoBean` (see "Testing This Project's Own HomeController: A Real Example").

Summary, Cheat Sheet, and Glossary

Testing in Spring MVC starts with a deliberate choice between three layers -- a pure unit test, a `@WebMvcTest` slice test, or a full `@SpringBootTest` integration test. Key points:

- `@WebMvcTest`: a slice test annotation that loads only the web layer (DispatcherServlet, controllers, converters), excluding `@Service` / `@Repository`
- `MockMvc`: a test tool that sends fake requests without opening a real HTTP server
- `MockMvcBuilders.standaloneSetup(...)`: an alternative setup that wires a controller pipeline by hand, without a Spring context
- `@MockitoBean`: an annotation that adds a Mockito mock to the context, or replaces a real bean (the replacement for `@MockBean`)
- `perform()` / `andExpect()`: the MockMvc methods that send a request and run chainable assertions, respectively
- `status()` / `view()` / `model()` / `jsonPath()` / `content()` / `header()`: matcher families that verify different response aspects (status code, view name, model attributes, JSON fields, body, headers)
- `MockMultipartFile`: a real spring-test class for multipart/form-data tests (test scope)

Quick reference:

```
@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());
    }
}
```

Glossary

`@WebMvcTest` — A Spring Boot test slice annotation that loads only the Spring MVC web layer, excluding `@Service` / `@Repository` beans.

`MockMvc` — A test tool for sending fake HTTP requests and verifying responses without opening a real server or socket.

`standaloneSetup` — A setup method that wires given controllers into a `MockMvc` pipeline by hand, without a Spring `ApplicationContext`.

`@MockitoBean` — An annotation that adds a Mockito mock to the test context, or replaces a real bean; the replacement for `@MockBean`.

`jsonPath` — A matcher that verifies a specific field of a JSON response body using a JSONPath expression.

`MockMultipartFile` — A fake file class provided by the `spring-test` library, used for multipart file upload tests.

Appendix: Mini Project — A Comprehensive Test Suite for This Project's Own TopicController

Bringing every technique from this lesson together, on this project's real `TopicController` (all six dependencies mocked with `@MockitoBean`):

```
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` provides helper methods that build the real `Course` / `Category` / `Topic` / `TopicTranslation` entities (all using Lombok `@Builder`) as a consistent tree. `TopicControllerWebMvcTest` covers three scenarios: 404 for an unknown slug, a 301 redirect from the old query-parameter URL (`/topics/{slug}?lang=..`) to the new path-based one, and 200 through the real `topic.html` template for a fully published topic -- in that last scenario, every mocked value is the same type the controller receives from real services in production (a real `Topic`, a real `MarkdownService.MarkdownRenderResult`), so the template renders normally, as if it were handling a real request.

Appendix: Mini Project — Testing an Interceptor with MockMvc

The last mini project tests the lifecycle from Advanced Spring MVC's "The HandlerInterceptor Interface: preHandle, postHandle, afterCompletion" section, in isolation with `MockMvc`, without ever writing a configuration class like the one in "WebMvcConfigurer: Registering an Interceptor":

```
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` is a small, realistic `HandlerInterceptor` that adds an `X-Response-Time-Ms` header to every request. `TimingInterceptorMockMvcTest` attaches it directly to `MockMvc` with `standaloneSetup(...).addInterceptors(...)` -- verifying the interceptor ON ITS OWN, without ever bringing in the whole application's configuration (path patterns, other interceptors).