Spring MVC Views ve Thymeleaf

Model, ModelMap ve ModelAndView ile view'a veri taşımanın üç yolu; Thymeleaf'in "natural templating" felsefesi; Thymeleaf değişken/link/mesaj ifadeleri (@{...}, #{...} ve dolar-süslü-parantez erişimi); th:if, th:each, th:fragment; SpringEL seçim ifadeleri (.?[...] / #vars); ve projenin kendi layout'una referansla MVC vs REST karşılaştırması.

Orta 40 dk

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

---

Spring MVC Views ve Thymeleaf

Spring MVC Fundamentals dersinde `Model`'in controller'dan view'a nasıl veri taşıdığını gördük, ama view'ın kendisine -- o `Model`'i gerçekten HTML'e çeviren şablon dosyasına -- hiç girmedik. Validation & Exception Handling dersi de tamamen `@RestController` tarafında geçti: JSON gövdeler, `ResponseEntity`, `ProblemDetail`. Bu ders, madalyonun öbür yüzüne, bu projenin asıl kullandığı tarafa dönüyor -- `@Controller`'ın döndürdüğü o mantıksal view adının, projenin kendi `templates/topic.html` ve `templates/fragments/layout.html` dosyalarında gerçek HTML'e nasıl çevrildiğine. Bu çeviriyi yapan teknoloji, `spring-boot-starter-thymeleaf` ile projeye giren **Thymeleaf**.

Spring MVC'de View Katmanı Nedir?

"ViewResolver: Mantıksal View Adından HTML'e" bölümünde (Spring MVC Fundamentals) `ViewResolver`'ın `"topic"` gibi bir view adını `templates/topic.html` dosyasına çevirdiğini görmüştük. View katmanı, tam olarak o dosyanın içeriğidir -- `Model`'e konan verinin nasıl HTML'e döküleceğini tanımlayan şablon:

```
// DispatcherServlet'in görüş açısından bir "View", tek bir metotla özetlenebilir:
interface MinimalView {
    void render(java.util.Map<String, Object> model,
                jakarta.servlet.http.HttpServletResponse response) throws java.io.
IOException;
}
```

Gerçek Spring MVC'de bu arayüzün adı da tam olarak `org.springframework.web.servlet.View`; Thymeleaf entegrasyonu, bu arayüzü implemente eden bir `ThymeleafView` sağlar -- `render` metodu, `Model`'i Thymeleaf'in kendi `Context`'ine kopyalayıp şablonu işler.

Neden Var?

View katmanı olmadan, her controller HTML'i Java string birleştirmeyle elle üretmek zorunda kalırdı -- okunması zor, XSS'e açık (elle escape etmeyi unutmak kolaydır) ve tasarımcı ile geliştiricinin aynı dosya üzerinde çalışmasını imkansızlaştıran bir yaklaşım. Bir şablon motoru, HTML yapısını (tasarımcının alanı) ile veriyi (controller'ın ürettiği) ayırır; Thymeleaf özellikle, bu ayrımı **"natural templating"** dediği bir felsefeyle yapar -- bir sonraki bölüm, Thymeleaf Nedir? "Natural Templating" Felsefesi, tam olarak bunun konusu.

Tarihçe

Thymeleaf 1.0, 2011'de, o dönem Spring dünyasında yaygın olan JSP'ye bir alternatif olarak çıktı -- JSP'nin `<%...%>` scriptlet'leri ve özel `.jsp` uzantısı yerine, düz `.html` dosyaları üzerinde çalışan bir motor öneriyordu. Thymeleaf 2.0 (2013), Spring entegrasyonunu (`thymeleaf-spring`) olgunlaştırdı. Thymeleaf 3.0 (2016), performansı (özellikle büyük şablonlarda) önemli ölçüde artıran yeni bir işleme motoruyla geldi ve bugün hâlâ kullanılan ana sürüm hattı bu. Spring Boot, 1.0'dan (2014) itibaren `spring-boot-starter-thymeleaf` ile Thymeleaf'i otomatik yapılandırıyor -- bu projenin de kullandığı yol; JSP, Spring Boot'un embedded servlet container modeliyle (Auto-Configuration dersinin konusu) iyi uyuşmadığı için Spring Boot dünyasında büyük ölçüde terk edildi.

Model, ModelMap ve ModelAndView: Veriyi View'a Taşımanın Üç Yolu

Spring MVC Fundamentals'ın "Model: Controller'dan View'a Veri Taşımak" bölümünde `Model` 'i gördük -- ama controller'dan view'a veri taşımanın tek yolu bu değil:

```
import org.springframework.ui.ExtendedModelMap;
import org.springframework.ui.Model;
import org.springframework.ui.ModelMap;
import org.springframework.web.servlet.ModelAndView;

// Three ways to hand data to a view. All three end up as the same thing under the
// hood -- a String-keyed map the view engine reads from -- but they differ in how
// (and where) you populate that map.
class ModelVariantsExample {

    // 1) Model: the interface you see most often as a controller method parameter.
    //    DispatcherServlet creates and injects it automatically (see the Spring MVC
    //    Fundamentals lesson's "Model: Passing Data from Controller to View" section).
    static Model buildWithModel() {
        Model model = new ExtendedModelMap();
        model.addAttribute("title", "Spring MVC Views & Thymeleaf");
        model.addAttribute("readingMinutes", 20);
        return model;
    }

    // 2) ModelMap: Model actually extends ModelMap -- Model just narrows the API down
    //    to addAttribute(...). ModelMap also exposes plain java.util.Map methods.
    static ModelMap buildWithModelMap() {
        ModelMap modelMap = new ModelMap();
        modelMap.addAttribute("title", "Spring MVC Views & Thymeleaf");
        modelMap.put("readingMinutes", 20);
        return modelMap;
    }

    // 3) ModelAndView: bundles the model AND the view name into a single object --
    //    an alternative to returning a String view name and taking Model as a
    //    parameter. Useful when the view name itself depends on some computation
    //    that happens after the model is already partly built.
    static ModelAndView buildWithModelAndView() {
        ModelAndView mav = new ModelAndView("topic");
        mav.addObject("title", "Spring MVC Views & Thymeleaf");
        mav.addObject("readingMinutes", 20);
        return mav;
    }

    public static void main(String[] args) {
        Model model = buildWithModel();
        System.out.println(model.asMap());
        // {title=Spring MVC Views & Thymeleaf, readingMinutes=20}

        ModelMap modelMap = buildWithModelMap();
        System.out.println(modelMap);
        // {title=Spring MVC Views & Thymeleaf, readingMinutes=20}

        ModelAndView mav = buildWithModelAndView();
        System.out.println(mav.getViewName() + " -> " + mav.getModel());
        // topic -> {title=Spring MVC Views & Thymeleaf, readingMinutes=20}
    }
}
```

Üçü de sonunda aynı yere varıyor: view'ın okuyacağı, string anahtarlı bir veri haritası. `Model`, `ModelMap`'i genişleten dar bir arayüz; `ModelMap` ise doğrudan bir `java.util.Map` gibi de kullanılabilir. `ModelAndView`, ikisini (veri + view adı) tek bir dönüş değerinde birleştirir -- bu projenin `TopicController.show`'u gibi, view adının bir dizi koşula göre değiştiği (`"topic"` her zaman aynı olsa da, `contentAvailable` bayrağına göre farklı bölümler render edilir) durumlarda `Model` parametresi + `String` dönüş değeri ayrımı genelde daha okunaklı kalır; `ModelAndView` daha çok view adının kendisi de dinamik olduğunda tercih edilir.

Thymeleaf Nedir? "Natural Templating" Felsefesi

Thymeleaf'i diğer şablon motorlarından ayıran temel fikir, bir şablonun **hem** geçerli HTML **hem de** işlenebilir bir şablon olmasıdır:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

// "Natural templating" is Thymeleaf's signature idea: a template is valid HTML on
// its own -- a browser (or a designer opening the .html file directly, with no
// server involved) renders it and sees reasonable placeholder content, because
// th:* attributes sit alongside real HTML attributes/text instead of replacing them
// with a foreign template syntax (unlike, say, JSP's <% ... %> scriptlets).
class NaturalTemplatingExample {

    private static final String TEMPLATE = """
            <p th:text="${message}">This is placeholder text a designer can see directly.<
/p>
            """;

    public static void main(String[] args) {
        // Opened as a plain .html file, with no processing at all, a designer still
        // sees a sensible sentence -- th:text is just an extra attribute, ignored by
        // any browser that doesn't understand it.
        System.out.println("Raw file, exactly as a browser without Thymeleaf sees it:");
        System.out.println(TEMPLATE);

        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        context.setVariable("message", "Rendered by ThymeleafViewResolver on the server");

        String processed = engine.process(TEMPLATE, context);
        System.out.println("Same file, processed by Thymeleaf:");
        System.out.println(processed);
        // <p>Rendered by ThymeleafViewResolver on the server</p>
    }
}
```

`th:text="${message}"` bir HTML attribute'u -- tarayıcı bunu tanımasa bile göz ardı eder ve etiketin içindeki düz metni ("This is placeholder text...") gösterir. Sunucu tarafında Thymeleaf işlediğindeyse, o metnin yerini `${message}`'in gerçek değeri alır. Bu, JSP'nin `<% %>` scriptlet'lerinin ya da Mustache gibi motorların `{{ }}` sözdiziminin **yapamadığı** bir şey -- onları içeren bir dosya, tarayıcıda ya da bir tasarım aracında doğrudan açıldığında bozuk görünür. Bu projenin `templates/topic.html`'i de bu yüzden bir tasarımcının (ya da senin) Thymeleaf hiç çalıştırmadan tarayıcıda önizleyebileceği, geçerli bir HTML dosyası.

Değişken İfadeleri: ${...} ile Model Verisine Erişmek

`${...}`, `Model`'e konan veriyi okumanın temel yolu:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

import java.util.List;

// ${...} is a variable expression -- it reads from the model the controller
// populated (see "Model, ModelMap ve ModelAndView"). Note the explicit ()
// on record accessors below (topic.title(), not topic.title) -- this project's own
// fragments/layout.html sidebar does the exact same thing (course.name(),
// category.slug()...) because a record's accessor is a real method, not a
// getTitle()-style bean property.
class VariableExpressionExample {

    record Topic(String title, int estimatedMinutes) {
    }

    public static void main(String[] args) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        context.setVariable("topic", new Topic("Spring MVC Views & Thymeleaf", 20));
        context.setVariable("tags", List.of("spring", "thymeleaf", "mvc"));

        String template = """
                <h1 th:text="${topic.title()}">Title</h1>
                <span th:text="${topic.estimatedMinutes()} + ' min'">0 min</span>
                <span th:text="${tags[0]}">tag</span>
                """;

        System.out.println(engine.process(template, context));
        // <h1>Spring MVC Views &amp;amp; Thymeleaf</h1>
        // <span>20 min</span>
        // <span>spring</span>
    }
}
```

`${topic.title()}` ifadesindeki `.title()` parantezine dikkat -- `Topic` bir `record` olduğu için erişimci (accessor) `getTitle()` değil, doğrudan `title()`. Bu, tesadüfen seçilmiş bir sözdizimi değil: bu projenin kendi `fragments/layout.html`'i, `CourseNav` / `CategoryNav` / `TopicNavItem` record'larına tam olarak aynı şekilde erişiyor (`course.name()`, `category.slug()`, `topicItem.title()`) -- "Bu Projenin Kendi Layout'u: fragments/layout.html ve Sidebar Accordion" bölümünde bunu gerçek dosyada göreceğiz. `${tags[0]}` gibi indeks erişimi de listeler için doğrudan çalışır.

Link İfadeleri: @{...} ile URL Oluşturmak

`@{...}`, bir URL üretir -- path variable'lar ve query parametreleri için ayrı bir string birleştirme yapmana gerek kalmaz:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

// @{...} is a link expression -- it builds a URL, adding the application's context
// path automatically and turning named placeholders (path variables) and query
// parameters into the right syntax. This is exactly how this project's own
// topic.html built its links before an SEO-driven redesign moved language into
// the URL path itself -- back then it read
// th:href="@{/topics/{slug}(slug=${topic.slug}, lang=${language.code})}", with
// `lang` spilling over into a `?lang=..` query string exactly like below.
class LinkExpressionExample {

    public static void main(String[] args) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        context.setVariable("slug", "spring-mvc-views-thymeleaf");
        context.setVariable("lang", "tr");

        String template = """
                <a th:href="@{/topics/{slug}(slug=${slug}, lang=${lang})}">link with a path
variable</a>
                <a th:href="@{/(lang='en')}">link with only a query parameter</a>
                """;

        System.out.println(engine.process(template, context));
        // <a href="/topics/spring-mvc-views-thymeleaf?lang=tr">...</a>
        // <a href="/?lang=en">...</a>
    }
}
```

`@{/topics/{slug}(slug=${slug}, lang=${lang})}` ifadesindeki parantez içi, iki farklı role ayrılıyor: `{slug}` adlı bir path placeholder path'te zaten varsa, aynı isimli parametre (`slug=${slug}`) oraya yerleştirilir; kalan parametreler (`lang=${lang}`) otomatik olarak `?`

`lang=tr` gibi bir query string'e dönüşür. Bu projenin `topic.html`'i SEO odaklı bir yeniden tasarımla dili URL path'inin kendisine taşımadan önce linklerini TAM OLARAK böyle kuruyordu -- `th:href="@{/topics/{slug}(slug=${topic.slug}, lang=${otherLanguage.code})}"`. Bugün aynı satır `th:href="@{/{lang}/topics/{slug}(lang=${otherLanguage.code}, slug=${topic.slug})}"` şeklinde -- hem `{lang}` hem `{slug}` artık birer path placeholder, bu yüzden query string'e taşacak bir şey kalmıyor; her iki durumda da bu, Path Variable'lar ve Request Parametreleri dersinde `@PathVariable` / `@RequestParam` ile sunucu tarafında okuduğumuz aynı ayrımın view tarafındaki karşılığı.

Metin Görüntüleme: th:text vs th:utext

`th:text` her zaman çıktısını **escape eder** (HTML özel karakterlerini kodlar); `th:utext` ("unescaped text") ise olduğu gibi yazar:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

// th:text escapes its value (HTML-encodes it) before writing it out; th:utext
// ("unescaped text") writes it out verbatim. This is the exact same escape-by-default
// idea this project relies on for markdown content -- see topic.html's th:utext on
// contentHtml, and its comment about that content being trusted (repo-controlled
// CommonMark output), never raw user input.
class TextVsUtextExample {

    public static void main(String[] args) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        // What if this string came from an untrusted source, e.g. a comment form?
        context.setVariable("comment", "<script>alert('xss')</script> nice topic!");

        String template = """
                <p th:text="${comment}">escaped</p>
                <p th:utext="${comment}">not escaped</p>
                """;

        System.out.println(engine.process(template, context));
        // <p>&lt;script&gt;alert('xss')&lt;/script&gt; nice topic!</p>   -- safe to render
        // <p><script>alert('xss')</script> nice topic!</p>              -- the script tag
survives
    }
}
```

Escape etmek varsayılan **ve** güvenli davranış -- kullanıcıdan gelen bir metinde `<script>` etiketi varsa, `th:text` onu zararsız düz metne çevirir. Bu projenin `topic.html`'i `th:utext="${contentHtml}"` kullanıyor -- yani escape **etmiyor** -- ama bu bilinçli bir istisna:

`contentHtml`, kullanıcı girdisi değil, `MarkdownService`'in **sunucuda, repo'daki `.md` dosyalarından** ürettiği güvenilir HTML (bkz. `topic.html`'deki ilgili yorum). Kullanıcıdan gelebilecek herhangi bir metin (örneğin gelecekte bir yorum formu) her zaman `th:text` ile render edilmeli.

Koşullu Render: th:if ve th:unless

`th:if`, koşul `falsy` ise etiketi **çıktıdan tamamen çıkarır** -- `display:none` gibi gizlemez, HTML'e hiç yazmaz; `th:unless` tam tersi koşulu kontrol eder:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

// th:if removes the whole tag (not just hides it -- it never reaches the response)
// when its expression is falsy; th:unless is the mirror image. This project's own
// topic.html uses exactly this pair for the "content not available in this language"
// branch: th:if="${!contentAvailable}" vs. th:if="${contentAvailable}".
class ConditionalRenderExample {

    public static void main(String[] args) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        String template = """
                <div th:if="${contentAvailable}">Content: <span th:text="${title}">t</span><
/div>
                <div th:unless="${contentAvailable}">Not available in this language yet.<
/div>
                <div th:if="${previousTopic != null}">Previous: <span th:
text="${previousTopic}">p</span></div>
                """;

        Context available = new Context();
        available.setVariable("contentAvailable", true);
        available.setVariable("title", "Spring MVC Views & Thymeleaf");
        available.setVariable("previousTopic", null);

        System.out.println(engine.process(template, available));
        // <div>Content: <span>Spring MVC Views &amp;amp; Thymeleaf</span></div>
        // (the th:unless div and the previousTopic div are both dropped entirely)

        Context unavailable = new Context();
        unavailable.setVariable("contentAvailable", false);
        unavailable.setVariable("title", null);
        unavailable.setVariable("previousTopic", "Request ve Response Handling");

        System.out.println(engine.process(template, unavailable));
        // <div>Not available in this language yet.</div>
        // <div>Previous: <span>Request ve Response Handling</span></div>
    }
}
```

Bu projenin `topic.html`'i tam olarak bu ikiliyi kullanıyor: `th:if="${!contentAvailable}"` ile "bu dilde henüz yok" uyarısını, `th:if="${contentAvailable}"` ile asıl içerik bloğunu koşullu render ediyor -- ikisi aynı anda render edilmiyor çünkü koşullar birbirinin tam tersi. `null` kontrolü de aynı mekanizmayla çalışır: `th:if="${previousTopic != null}"`, ilk konuda "Önceki" linkinin hiç görünmemesini sağlıyor (navigasyon bölümünün `th:if="${previousTopic != null}"` satırı).

Döngüler: th:each ile Liste Render Etmek

`th:each`, bulunduğu etiketi koleksiyondaki her eleman için bir kez tekrarlar:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

import java.util.List;

// th:each repeats the tag it's on once per element, optionally exposing a second,
// "status" variable (iterStat below) with index/count/even/odd/first/last -- this
// project's own sidebar fragment uses the same mechanic (th:each="topicItem :
// ${category.topics()}") without needing the status variable at all.
class IterationExample {

    record TopicItem(String slug, String title) {
    }

    public static void main(String[] args) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        context.setVariable("topics", List.of(
                new TopicItem("spring-mvc-fundamentals", "Spring MVC Fundamentals"),
                new TopicItem("validation-exception-handling", "Validation & Exception
Handling"),
                new TopicItem("spring-mvc-views-thymeleaf", "Spring MVC Views &
Thymeleaf")));

        String template = """
                <ul>
                    <li th:each="topic, iterStat : ${topics}"
                        th:text="${iterStat.count} + '. ' + ${topic.title()} + (${iterStat.
last} ? ' (last)' : '')">
                        item
                    </li>
                </ul>
                """;

        System.out.println(engine.process(template, context));
        // <li>1. Spring MVC Fundamentals</li>
        // <li>2. Validation &amp;amp; Exception Handling</li>
        // <li>3. Spring MVC Views &amp;amp; Thymeleaf (last)</li>
    }
}
```

`topic, iterStat : ${topics}` sözdizimindeki `iterStat`, isteğe bağlı bir **durum değişkeni** -- `count`, `index`, `size`, `first`, `last`, `even`, `odd` gibi alanlar taşır. Bu projenin sidebar'ı (`fragments/layout.html`) durum değişkenini hiç kullanmıyor (`th:each="topicItem : ${category.topics()}"`) çünkü ihtiyacı yok; ama örneğin bir listedeki son elemana farklı bir

stil vermek istediğinde (bu dersin "Ek: Mini Proje — Basit Bir Blog Sayfası" bölümündeki gibi) `iterStat.last` tam olarak bunun için var.

Mesaj İfadeleri: #{...} ile i18n Entegrasyonu

`#{...}`, i18n dersinde gördüğümüz `messages*.properties` mekanizmasının Thymeleaf'teki karşılığı -- bir anahtarı, mevcut locale'e göre çözümlenmiş bir metne çevirir:

```
import java.text.MessageFormat;
import java.util.ListResourceBundle;
import java.util.Locale;
import java.util.ResourceBundle;

// #{...} is a message expression -- it looks up a key in a locale-specific bundle and
// (optionally) fills in {0}, {1}... placeholders, exactly like this project's own
// messages*.properties + MessageSource setup (see TopicController.buildUnavailableMessage,
// which does the same lookup + formatting by hand for a message that needs different
// word order in Turkish vs. English). Thymeleaf's #{...} is this same mechanism wired
// through an IMessageResolver that, in this project, ultimately delegates to Spring's
// MessageSource -- this example reproduces just the lookup/format part in plain Java,
// without a real Thymeleaf message resolver, to keep the demo focused.
class MessageExpressionExample {

    static class TrBundle extends ListResourceBundle {
        protected Object[][] getContents() {
            return new Object[][]{
                    {"topic.unavailable", "Bu içerik {0} dilinde henüz mevcut değil."}
            };
        }
    }

    static class EnBundle extends ListResourceBundle {
        protected Object[][] getContents() {
            return new Object[][]{
                    {"topic.unavailable", "This content is not yet available in {0}."}
            };
        }
    }

    static String resolve(String key, Locale locale, Object... params) {
        ResourceBundle bundle = locale.getLanguage().equals("tr") ? new TrBundle() : new
EnBundle();
        String pattern = bundle.getString(key);
        return MessageFormat.format(pattern, params);
    }

    public static void main(String[] args) {
        System.out.println(resolve("topic.unavailable", Locale.forLanguageTag("tr"),
"İngilizce"));
        // Bu içerik İngilizce dilinde henüz mevcut değil.

        System.out.println(resolve("topic.unavailable", Locale.forLanguageTag("en"),
"Turkish"));
        // This content is not yet available in Turkish.

        // In a Thymeleaf template, the same lookup is one attribute:
        //   <p th:text="#{topic.unavailable(${languageName})}">...</p>
        // -- no key found for the current locale falls back to "??key??" by default,
        // which is exactly the kind of silent-looking bug worth watching for
        // (see "Common Mistakes").
    }
}
```

Gerçek bir Thymeleaf şablonunda bu tek satırdır: `th:text="#{topic.unavailable (${languageName})}"`. Perde arkasında olan şey, tam olarak `TopicController.buildUnavailableMessage` metodunun elle yaptığı şey -- bir `MessageSource`'tan, locale'e göre bir bundle seçip `{0}` gibi parametre yer tutucularını doldurmak; farkı, Thymeleaf'in bunu her `#{...}` gördüğünde otomatik yapması. Bu projenin `topic.html`'i `#{nav.previous}`, `#{breadcrumb.home}`, `#{toc.onThisPage}` gibi UI metinleri için bunu zaten kullanıyor -- `buildUnavailableMessage`'ın elle yazılmış olması, "Türkçe'de dil adı cümlenin farklı bir yerinde geçiyor" gibi tek bir `{0}` yer tutucusunun yetmediği, karmaşık cümle yapısı gerektiren **özel** bir durum (bkz. `TopicController`'daki ilgili Javadoc).

Fragment'ler: th:fragment, th:insert ve th:replace

Bir sayfanın her yerinde tekrar eden parçaları (navbar, footer, bir kart bileşeni) `th:fragment` ile bir kez tanımlayıp, `th:insert` / `th:replace` ile istediğin yere çağırırsın:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

// th:fragment marks a reusable chunk of markup (optionally with parameters).
// th:insert and th:replace both pull that chunk in elsewhere -- the only difference
// is th:insert keeps the host tag, th:replace swaps the host tag out for the
// fragment's own root tag. This example references a fragment defined earlier in the
// SAME template string with ~{::selector} ("this template"); this project's real
// fragments/layout.html instead defines fragments in a separate file and topic.html
// pulls them in with th:replace="~{fragments/layout :: navbar}" (see "Bu Projenin
// Kendi Layout'u").
class FragmentExample {

    public static void main(String[] args) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        context.setVariable("badgeText", "INTERMEDIATE");

        String template = """
                <span th:fragment="badge(text)" class="badge" th:text="${text}">badge</span>

                <div>
                    <p>Inserted (keeps the surrounding div):</p>
                    <div th:insert="~{::badge(${badgeText})}">placeholder</div>
                </div>

                <div>
                    <p>Replaced (the div itself is swapped out for the span):</p>
                    <div th:replace="~{::badge(${badgeText})}">placeholder</div>
                </div>
                """;

        System.out.println(engine.process(template, context));
        // <div><span class="badge">INTERMEDIATE</span></div>  -- th:insert: div survives
        // <span class="badge">INTERMEDIATE</span>             -- th:replace: div is gone
    }
}
```

Aradaki fark tek bir şey: `th:insert`, fragment'i **konak etiketin içine** yerleştirir (konak etiket kalır); `th:replace`, konak etiketin **yerine geçer** (fragment kendi kök etiketiyle onun yerini alır). Bu projenin `topic.html`'i `<div th:replace="~{fragments/layout :: navbar}"></div>` gibi hep `th:replace` kullanıyor -- çünkü o `<div>`'in kendisinin çıktıda kalmasına gerek yok, yalnızca fragment'in konumunu işaretliyor. `~{fragments/layout :: navbar}` sözdizimindeki `fragments/layout`, ayrı bir dosyayı; bu örnekteki `~{::badge(...)}`'teki `::` ise "bu template'in kendisini" işaret ediyor.

SpringEL Seçim İfadeleri: .?[...] ve #vars

`.?[...]`, bir koleksiyonu filtreler -- köşeli parantez içindeki koşul, her eleman için `#this` o elemana bağlanmış olarak değerlendirilir:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

import java.util.List;

// .?[...] is a selection expression -- it filters a collection, evaluating the
// bracketed condition once per element with #this bound to that element. The catch:
// inside that bracket, #this rebinds the whole expression scope to the element, so a
// bare reference to an outer context variable no longer resolves the way it does
// everywhere else in the template. #vars.xxx reaches back to the top-level context
// variables explicitly, bypassing whatever #this currently means.
//
// This is not a toy problem -- this project's own fragments/layout.html sidebar hit
// exactly this while computing which category should default to expanded:
// "#vars.activeTopicSlug", never a bare "activeTopicSlug", inside a .?[...] selection
// (see CLAUDE.md's "Known Constraints" for the SpelEvaluationException this caused
// the first time around).
class SelectionExpressionExample {

    record TopicItem(String slug, String title) {
    }

    public static void main(String[] args) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        context.setVariable("activeSlug", "spring-mvc-views-thymeleaf");
        context.setVariable("topics", List.of(
                new TopicItem("spring-mvc-fundamentals", "Spring MVC Fundamentals"),
                new TopicItem("spring-mvc-views-thymeleaf", "Spring MVC Views &
Thymeleaf")));

        // #this here refers to each TopicItem in turn; #vars.activeSlug reaches past
        // that rebinding to the context variable set outside the selection.
        String template = """
                <p th:with="matches=${topics.?[#this.slug() == #vars.activeSlug]}"
                   th:text="${matches.size()} + ' match(es): ' + ${matches[0].title()}">
                    result
                </p>
                """;

        System.out.println(engine.process(template, context));
        // 1 match(es): Spring MVC Views &amp;amp; Thymeleaf
    }
}
```

> `#this`, seçim ifadesinin içinde **tüm değerlendirme kapsamını** o anki elemana çevirir -- bu yüzden köşeli parantez içinde dış bağlamdaki bir değişkene çıplak isimle (`activeSlug`) erişmeye çalışmak, o değişkeni değil elemanın kendi alanlarından birini aramaya çalışır ve başarısız olur. `#vars.activeSlug`, bu rebinding'i atlayıp doğrudan en dıştaki context değişkenlerine ulaşır. Bu projenin gerçek `fragments/layout.html` sidebar'ı, `categoryIsActive` hesabında tam olarak bu tuzağa düşmüştü -- `.?[#this.slug() == activeTopicSlug]` bir `SpelEvaluationException` fırlatıyordu, `.?[#this.slug() == #vars.activeTopicSlug]` ile düzeldi (bkz. proje notlarındaki "Bilinen Kısıtlar").

Bu Projenin Kendi Layout'u: fragments/layout.html ve Sidebar Accordion

Bu dersteki her mekanizmayı, projenin kendi `templates/fragments/layout.html` ve `templates/topic.html` dosyalarında görebilirsin. `layout.html`, üç fragment tanımlıyor: `navbar`, `sidebar`, `footer` -- `topic.html` ve `index.html` bunları `th:replace` ile içe alıyor ("Fragment'ler: th:fragment, th:insert ve th:replace" bölümündeki mekanizmanın ta kendisi).

Sidebar'daki kategori accordion'u, bu dersin neredeyse tüm konularını tek bir yerde birleştiriyor: `th:each="category : ${course.categories()}"` ile her kategoriyi dolaşıyor ("Döngüler: th:each ile Liste Render Etmek"), `th:with` ile `categoryId` ve `categoryIsActive` adında iki yerel değişken hesaplıyor (`categoryIsActive`, tam olarak "SpringEL Seçim İfadeleri: .?[...] ve #vars" bölümündeki `.?[...]` + `#vars` deseniyle), `th:classappend="${categoryIsActive} ? 'show' : ''"` ile koşullu bir CSS sınıfı ekliyor, ve `th:each="topicItem : ${category.topics()}"` ile o kategorinin konularını listeliyor. Bootstrap'in kendi `data-bs-toggle="collapse"` mekanizmasıyla birlikte çalışıyor -- Thymeleaf yalnızca doğru `id` / `aria-expanded` / CSS sınıflarını hesaplıyor, açılıp kapanma animasyonunun kendisi tamamen Bootstrap'in JavaScript'i.

Form Binding (Kısa Bakış): th:object ve th:field

Bu proje henüz bir form içermiyor -- her sayfa salt okunur içerik. Ama Thymeleaf'in Spring'e özel form dialect'i, `@ModelAttribute` ile gelecek herhangi bir formu bağlamak için `th:object` / `th:field` sunar:

```
// This project has no forms yet -- every page is read-only content. th:object/
// th:field belong to Thymeleaf's Spring-specific form dialect, which needs a real
// @ModelAttribute-backed BindingResult and Spring's RequestDataValueProcessor wired
// through an actual request -- infrastructure this focused example intentionally
// does NOT stand up (see CLAUDE.md's "Example Writing Principles": no unrelated
// infrastructure just to make something independently runnable). Instead, this shows
// what th:object/th:field expand into, so the mechanism is clear if this project
// ever adds a form (e.g. a future "Appendix: Mini Project" comment submission).
class FormBindingExample {

    // The @ModelAttribute-backed object a real controller would put on the Model,
    // e.g. model.addAttribute("commentForm", new CommentForm()).
    static class CommentForm {
        private String author = "";
        private String body = "";

        String getAuthor() {
            return author;
        }

        void setAuthor(String author) {
            this.author = author;
        }

        String getBody() {
            return body;
        }

        void setBody(String body) {
            this.body = body;
        }
    }

    public static void main(String[] args) {
        CommentForm form = new CommentForm();
        form.setAuthor("Ada");

        System.out.println("Template (what you'd write):");
        System.out.println("""
                <form th:object="${commentForm}" method="post">
                    <input type="text" th:field="*{author}"/>
                    <textarea th:field="*{body}"></textarea>
                </form>
                """);

        System.out.println("What th:field expands to for each bound property,");
        System.out.println("given commentForm.author = \"" + form.getAuthor() + "\":");
        System.out.println("""
                <input type="text" id="author" name="author" value="Ada"/>
                <textarea id="body" name="body"></textarea>
                """);
        // th:object="${commentForm}" sets the "current object" *{...} expressions
        // resolve against; th:field="*{author}" reads commentForm.getAuthor() for
        // the value AND derives id/name="author" from the property name -- the same
        // property Spring's DataBinder writes back into on form submission.
    }
}
```

`th:object`, `*{...}` ifadelerinin (yıldızlı, `${...}`'ten farklı) hangi nesneye göre çözüleceğini belirler; `th:field="*{author}"`, o nesnenin `author` alanını hem `value` olarak okur hem de `id` / `name` attribute'larını alan adından türetir -- aynı `name`, formun `POST` edilmesiyle Spring'in `DataBinder`'ının geri yazacağı alandır. Bu proje form eklediğinde (örneğin bir yorum formu), bu mekanizma Request ve Response Handling dersindeki `@RequestBody` / `HttpMessageConverter` ikilisine tam bir alternatif olur -- biri JSON gövdeyi bir nesneye bağlar, diğeri form alanlarını.

MVC (Sunucu Taraflı Render) vs REST: Ne Zaman Hangisi?

Spring MVC Fundamentals'ın "@Controller vs @RestController: Ne Zaman Hangisi?" bölümünde bu ayrımı annotation seviyesinde görmüştük; şimdi view katmanını da öğrendiğimize göre, sonuçlarını karşılaştırabiliriz. Sunucu taraflı render (`@Controller` + Thymeleaf, bu projenin kullandığı yol), tarayıcıya doğrudan görüntülenebilir HTML gönderir -- ilk sayfa yükü daha hızlı görünür (JavaScript'in veri çekip DOM kurmasını beklemez), SEO doğal olarak çalışır (arama motoru zaten HTML görür), ama her sayfa geçişi bir tam sayfa yüklemesi (ya da en azından bir sunucu round-trip'i) gerektirir. REST (`@RestController` + JSON, bir single-page-application'ın tükettiği API), istemciye yalnızca veri gönderir -- istemci tarafı (React, Vue...) bunu DOM'a çevirir; sayfa geçişleri daha akıcı olabilir ama ilk yük daha ağırdır ve SEO için ekstra çaba (server-side rendering, prerendering) gerekir. Bu proje bir öğrenim sitesi -- içerik büyük ölçüde statik, SEO önemli, karmaşık istemci-taraflı etkileşim gerekmiyor -- bu yüzden sunucu taraflı render, bilinçli bir tercih (Spring MVC Fundamentals'ın "Spring MVC vs Spring WebFlux (Kısa Bakış)" bölümündeki blocking/non-blocking değerlendirmesine benzer bir gerekçelendirme).

Best Practices

- **`th:utext` yalnızca güvenilir, sunucuda üretilmiş içerik için kullan** -- kullanıcıdan gelebilecek her metin `th:text` ile escape edilmeli (bkz. "Metin Görüntüleme: th:text vs th:utext"); bu projenin `contentHtml` istisnası bilinçli ve belgelenmiş, varsayılan değil.
- **Seçim/projection ifadelerinde (`.?[...]`, `.^[...]`, `.![...]`) dış değişkenlere her zaman `#vars.` ile eriş** -- `#this`'in kapsamı değiştirdiğini unutmak, bu projenin sidebar'ında gerçekten yaşanmış bir hataya yol açtı (bkz. "SpringEL Seçim İfadeleri: .?[...] ve #vars").
- **Fragment'leri `th:replace` ile kullan, konak etiketin çıktıda kalmasına gerek olmadıkça** -- gereksiz `<div>` sarmalayıcıları CSS'te (özellikle flex/grid düzenlerinde) beklenmedik boşluklara yol açabilir.
- **View'a yalnızca render için gereken veriyi koy, iş mantığını değil** -- "Model, ModelMap ve ModelAndView: Veriyi View'a Taşımanın Üç Yolu" bölümündeki üç mekanizmanın hiçbiri, view'ın o veriyi nasıl kullanacağını sınırlamaz; disiplini geliştirici sağlamak zorunda -- Spring MVC Fundamentals'ın "Controller'ları ince tut, iş mantığını service katmanına bırak" gerekçesiyle aynı fikir, burada view katmanı için geçerli.

Yaygın Hatalar

1. `th:text` yerine `th:utext` kullanmayı alışkanlık hâline getirmek. "Neden çalışmıyor?" diye `th:utext`'e geçip unutmak, kullanıcı girdisi içeren bir alanda XSS'e kapı açar -- `th:text`'in escape etmesi neredeyse her zaman **istenen** davranıştır (bkz. "Metin Görüntüleme: th:text vs th:utext").

2. `.?[...]` içinde dış değişkene çıplak isimle erişmeye çalışmak. `#this` kapsamı değiştirdiği için, beklediğin değişken yerine elemanın kendi bir alanı aranır ve genelde bir `SpelEvaluationException` ile sonuçlanır (bkz. "SpringEL Seçim İfadeleri: .?[...] ve #vars").

3. `${...}` içinde bir record alanına `.title` (parantezsiz) ile erişmeye çalışmak. Bean-tarzı bir sınıfta `getTitle()` property olarak `title`'a karşılık gelir, ama bir record'da erişimci gerçek bir metottur (`title()`) -- `${topic.title}` bazı durumlarda sessizce `null` dönebilir ya da hata verebilir; güvenli olan her zaman `${topic.title()}` yazmak (bkz. "Değişken İfadeleri: ${...} ile Model Verisine Erişmek").

4. `th:insert` ile `th:replace`'i birbirine karıştırmak. İkisi de fragment'i getirir ama biri konak etiketi bırakır, diğeri onun yerine geçer -- yanlış seçim, çıktıda beklenmedik fazladan bir sarmalayıcı etiket (`th:insert`) ya da beklenen sarmalayıcının kaybolması (`th:replace`) şeklinde ortaya çıkar (bkz. "Fragment'ler: th:fragment, th:insert ve th:replace").

5. `@{...}` içindeki path placeholder ile parametre adının eşleşmediğini fark etmemek. `@{/topics/{slug}(id=${slug})}` gibi bir yazımda, parantez içindeki parametre adı (`id`) path'teki placeholder'la (`{slug}`) eşleşmediği için placeholder hiç doldurulmaz ve `id` parametresi query string'e düşer -- sonuç `/topics/{slug}?id=...` gibi bozuk bir URL olur (bkz. "Link İfadeleri: @{...} ile URL Oluşturmak").

6. `#{...}` anahtarının her iki dilde de (tr/en `messages*.properties`) tanımlı olduğunu varsaymak. Eksik bir anahtar, sayfada sessizce `??key??` gibi bir metin olarak belirir -- derleme zamanında yakalanmaz, yalnızca o sayfayı o dilde ziyaret ettiğinde fark edilir (bkz. "Mesaj İfadeleri: #{...} ile i18n Entegrasyonu").

Özet, Cheat Sheet ve Terimler Sözlüğü

Thymeleaf, Spring MVC'nin varsayılan view teknolojisi -- "natural templating" felsefesiyle, bir şablonun hem geçerli HTML hem de işlenebilir bir dosya olmasını sağlıyor. Öne çıkan noktalar:

- `Model` / `ModelMap` / `ModelAndView`: controller'dan view'a veri taşımanın üç eşdeğer yolu
- `${...}`: değişken ifadesi, `Model`'deki veriyi okur (record'larda parantezli erişim: `topic.title()`)
- `@{...}`: link ifadesi, context path + path variable + query parametrelerini otomatik birleştirir
- `#{...}`: mesaj ifadesi, i18n bundle'ından (bu projede Spring'in `MessageSource`'u üzerinden) çözümlenmiş metin döner

- `th:text` / `th:utext`: sırasıyla escape'li ve escape'siz metin yazımı
- `th:if` / `th:unless`: etiketi tamamen render'dan çıkaran koşullu bloklar
- `th:each`: koleksiyonu dolaşıp etiketi her eleman için tekrarlar; `iterStat` ile index/count /first/last erişilebilir
- `th:fragment` / `th:insert` / `th:replace`: yeniden kullanılabilir parça tanımlama ve içe alma (`th:replace` konak etiketin yerine geçer)
- `.?[...]`: seçim (filtreleme) ifadesi; içinde `#this` her elemana bağlanır, dış değişkenlere `#vars.` ile erişilir
- `th:object` / `th:field`: form alanlarını bir Java nesnesine bağlar (bu projede henüz kullanılmıyor)

Hızlı referans:

```
<!-- değişken + link + mesaj -->
<a th:href="@{/{lang}/topics/{slug}(lang=${language.code}, slug=${topic.slug()})}" th:
text="${topic.title()}">Konu</a>
<span th:text="#{time.minutesShort}">dk</span>

<!-- koşul + döngü -->
<div th:if="${!items.isEmpty()}">
    <p th:each="item, stat : ${items}" th:text="${stat.count} + '. ' + ${item.name()}"
>satır</p>
</div>
<div th:unless="${!items.isEmpty()}">Boş.</div>

<!-- fragment tanımı ve çağrısı -->
<div th:fragment="card(title)" class="card" th:text="${title}">kart</div>
<div th:replace="~{::card(${topic.title()})}">yer tutucu</div>

<!-- güvenli vs güvenilir içerik -->
<p th:text="${userComment}">kullanıcıdan gelen -- escape'li</p>
<article th:utext="${serverRenderedMarkdown}">sunucuda üretilmiş -- escape'siz</article>
```

Terimler Sözlüğü

Thymeleaf — Spring Boot'un varsayılan olarak yapılandırdığı, "natural templating" felsefesine sahip Java şablon motoru.

Natural templating — Bir şablonun, işlenmeden önce de tarayıcıda/tasarım aracında geçerli ve anlamlı görünmesini sağlayan Thymeleaf tasarım ilkesi.

Değişken ifadesi (`${...}`) — Model/context'teki bir değeri okuyan Thymeleaf ifadesi.

Link ifadesi (`@{...}`) — Context path, path variable ve query parametrelerini birleştirerek bir URL üreten Thymeleaf ifadesi.

Mesaj ifadesi (`#{...}`) — Bir i18n anahtarını, mevcut locale'e göre çözümlenmiş metne çeviren Thymeleaf ifadesi.

`th:text` / **`th:utext`** — Bir etiketin metnini sırasıyla escape'li ve escape'siz yazan attribute'lar.

Fragment — `th:fragment` ile tanımlanan, `th:insert` / `th:replace` ile başka bir yerde yeniden kullanılan şablon parçası.

Seçim ifadesi (`.?[...]`) — Bir koleksiyonu, içindeki `#this` o anki elemana bağlı bir koşula göre filtreleyen ifade.

`#vars` — Seçim/projection gibi kapsam değiştiren ifadeler içinde, en dıştaki context değişkenlerine doğrudan erişmeyi sağlayan Thymeleaf temel nesnesi.

`th:object` / **`th:field`** — Bir form alanını, `@ModelAttribute` ile gelen bir Java nesnesinin alanına bağlayan Thymeleaf form dialect'i attribute'ları.

Ek: Mini Proje — Basit Bir Blog Sayfası

Bu dersteki mekanizmaların birlikte çalıştığı küçük bir sayfa kuruyoruz: bir `th:fragment` (tek bir yazı kartı), `th:each` (yazı listesi) ve `th:if` / `th:unless` (boş liste durumu) aynı şablonda:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

import java.util.List;

// Mini project, part 1/2: a tiny blog listing page that combines everything from this
// lesson -- a th:fragment for one post "card", th:each to repeat it, and th:if/
// th:unless for the empty-state message. See BlogPageDemo for how it's driven.
class BlogPageTemplateExample {

    record Post(String title, String excerpt) {
    }

    private static final String TEMPLATE = """
            <div th:fragment="postCard(post)" class="post-card">
                <h3 th:text="${post.title()}">title</h3>
                <p th:text="${post.excerpt()}">excerpt</p>
            </div>

            <section>
                <h2>Blog</h2>
                <div th:if="${#lists.isEmpty(posts)}">No posts yet.</div>
                <div th:unless="${#lists.isEmpty(posts)}">
                    <div th:each="post : ${posts}" th:insert="~{::postCard(${post})}"
>placeholder</div>
                </div>
            </section>
            """;

    static String render(List<Post> posts) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        context.setVariable("posts", posts);

        return engine.process(TEMPLATE, context);
    }
}
```

```
import java.util.List;

// Mini project, part 2/2: drives BlogPageTemplateExample with two different inputs --
// a populated list (th:each + the postCard fragment fire) and an empty one (the
// th:if empty-state message fires instead).
class BlogPageDemo {

    public static void main(String[] args) {
        List<BlogPageTemplateExample.Post> posts = List.of(
                new BlogPageTemplateExample.Post(
                        "Spring MVC Views & Thymeleaf",
                        "Model, ModelAndView, and Thymeleaf's basic syntax."),
                new BlogPageTemplateExample.Post(
                        "Validation & Exception Handling",
                        "Bean Validation and RFC 7807 ProblemDetail."));

        System.out.println("With posts:");
        System.out.println(BlogPageTemplateExample.render(posts));
        // <section><h2>Blog</h2><div><div class="post-card">...2 cards...</div></div><
/section>

        System.out.println("With no posts:");
        System.out.println(BlogPageTemplateExample.render(List.of()));
        // <section><h2>Blog</h2><div>No posts yet.</div></section>
    }
}
```

`postCard` fragment'i tek bir yazıyı biliyor -- başlık ve özet. `th:unless="${#lists.isEmpty(posts)}"` bloğu, listede en az bir yazı varsa devreye girip `th:each` ile her yazı için `postCard`'ı `th:insert` ediyor; liste boşsa `th:if="${#lists.isEmpty(posts)}"` bloğu tek başına "No posts yet." metnini gösteriyor. `BlogPageDemo`, aynı `render` metodunu önce dolu bir listeyle, sonra boş bir listeyle çağırarak iki dalın da doğru çalıştığını gösteriyor.

Ek: Mini Proje — i18n Destekli Ürün Kartı Şablonu

İkinci mini proje, `${...}`, `@{...}` ve `th:if`'i, "Mesaj İfadeleri: #{...} ile i18n Entegrasyonu" bölümündeki yaklaşımla (mesajı önceden çözüp şablona hazır bir string olarak beslemek) bir araya getiriyor:

```
import org.thymeleaf.TemplateEngine;
import org.thymeleaf.context.Context;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.StringTemplateResolver;

// Mini project, part 1/2: a product card template that combines ${...} (variables),
// @{...} (a link to the product page), and th:if (a conditional discount badge).
// The "Add to cart" label is passed in already resolved -- in a real Thymeleaf setup
// this would be a live #{...} lookup through Spring's MessageSource, but this example
// keeps the resolution step separate (see ProductCardDemo, which does it the same way
// MessageExpressionExample and TopicController.buildUnavailableMessage do) so this
// class only has to demonstrate the templating side.
class ProductCardTemplateExample {

    record Product(String slug, String name, String priceLabel, boolean discounted) {
    }

    private static final String TEMPLATE = """
            <div class="product-card">
                <a th:href="@{/products/{slug}(slug=${product.slug()})}" th:text="${product.
name()}">name</a>
                <span th:text="${product.priceLabel()}">price</span>
                <span th:if="${product.discounted()}" class="badge">%</span>
                <button th:text="${addToCartLabel}">Add to cart</button>
            </div>
            """;

    static String render(Product product, String addToCartLabel) {
        TemplateEngine engine = new TemplateEngine();
        StringTemplateResolver resolver = new StringTemplateResolver();
        resolver.setTemplateMode(TemplateMode.HTML);
        engine.setTemplateResolver(resolver);

        Context context = new Context();
        context.setVariable("product", product);
        context.setVariable("addToCartLabel", addToCartLabel);

        return engine.process(TEMPLATE, context);
    }
}
```

```
import java.util.List;
import java.util.Map;

// Mini project, part 2/2: renders the same product in Turkish and English by
// resolving "addToCartLabel" per locale first (a stand-in for a real #{...} lookup),
// then feeding the already-resolved string into ProductCardTemplateExample -- and
// renders a discounted vs. a regular-priced product to exercise the th:if badge.
class ProductCardDemo {

    private static final Map<String, String> ADD_TO_CART = Map.of("tr", "Sepete Ekle",
"en", "Add to Cart");

    public static void main(String[] args) {
        var mug = new ProductCardTemplateExample.Product("java-mug", "Java Mug", "$12.00",
false);
        var keyboard = new ProductCardTemplateExample.Product("mechanical-keyboard",
"Mechanical Keyboard", "$79.00", true);

        for (String lang : List.of("tr", "en")) {
            String addToCartLabel = ADD_TO_CART.get(lang);

            System.out.println("[" + lang + "] regular price:");
            System.out.println(ProductCardTemplateExample.render(mug, addToCartLabel));
            // no <span class="badge">%</span>

            System.out.println("[" + lang + "] discounted:");
            System.out.println(ProductCardTemplateExample.render(keyboard, addToCartLabel));
            // includes <span class="badge">%</span>
        }
    }
}
```

`ProductCardTemplateExample`, bir ürünü (`${product.name()}`), ürün sayfasına giden bir linki (`@{/products/{slug}(slug=${product.slug()})}`) ve indirimliyse görünen bir rozeti (`th:if="${product.discounted()}"`) render ediyor. `ProductCardDemo`, "Sepete Ekle"/"Add to Cart" metnini dil koduna göre önceden çözüp aynı şablonu iki dilde, hem indirimli hem normal fiyatlı bir ürün için çalıştırıyor -- gerçek bir Thymeleaf kurulumunda bu son adım `#{addToCart}` ile tek satıra iner, ama ayrık tutmak, şablonun kendisini mesaj çözümleme altyapısından bağımsız test edilebilir kılıyor.