Spring MVC Views and Thymeleaf

Three ways to carry data to the view with Model, ModelMap, and ModelAndView; Thymeleaf's "natural templating" philosophy; Thymeleaf variable/link/message expressions (@{...}, #{...}, and dollar-brace variable access); th:if, th:each, th:fragment; SpringEL selection expressions (.?[...] / #vars); and MVC vs. REST, with reference to this project's own layout.

Intermediate 40 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/spring-mvc-views-thymeleaf

---

Spring MVC Views and Thymeleaf

In Spring MVC Fundamentals we saw how `Model` carries data from the controller to the view, but never looked at the view itself -- the template file that actually turns that `Model` into HTML. Validation & Exception Handling stayed entirely on the `@RestController` side too: JSON bodies, `ResponseEntity`, `ProblemDetail`. This lesson turns to the other side of the coin, the one this project actually uses -- how the logical view name returned by `@Controller` becomes real HTML in this project's own `templates/topic.html` and `templates/fragments/layout.html` files. The technology that does that translation, brought in by `spring-boot-starter-thymeleaf`, is **Thymeleaf**.

What Is the View Layer in Spring MVC?

In "ViewResolver: From Logical View Name to HTML" (Spring MVC Fundamentals) we saw `ViewResolver` translate a view name like `"topic"` into `templates/topic.html`. The view layer is exactly that file's content -- a template that describes how the data placed into the `Model` should be turned into HTML:

```
// From DispatcherServlet's point of view, a "View" can be summarized as one method:
interface MinimalView {
    void render(java.util.Map<String, Object> model,
                jakarta.servlet.http.HttpServletResponse response) throws java.io.
IOException;
}
```

In real Spring MVC, that interface's actual name is `org.springframework.web.servlet.View`; the Thymeleaf integration provides a `ThymeleafView` that implements it -- its `render` method copies the `Model` into Thymeleaf's own `Context` and processes the template.

Why Does It Exist?

Without a view layer, every controller would have to build HTML by hand through Java string concatenation -- hard to read, prone to XSS (it's easy to forget to escape something by hand), and it makes it impossible for a designer and a developer to work on the same file. A template engine separates HTML structure (the designer's territory) from data (what the controller produces); Thymeleaf in particular does that separation with a philosophy it calls **"natural templating"** -- exactly the topic of the next section, What Is Thymeleaf? The "Natural Templating" Philosophy.

History

Thymeleaf 1.0 shipped in 2011 as an alternative to JSP, which was the common choice in the Spring world at the time -- instead of JSP's `<%...%>` scriptlets and its dedicated `.jsp` extension, it proposed an engine that works on plain `.html` files. Thymeleaf 2.0 (2013) matured the Spring integration (`thymeleaf-spring`). Thymeleaf 3.0 (2016) introduced a new processing engine that significantly improved performance (especially for large templates), and that's still the main release line in use today. Spring Boot has auto-configured Thymeleaf through `spring-boot-starter-thymeleaf` since 1.0 (2014) -- the path this project also takes; JSP fell out of favor in the Spring Boot world largely because it doesn't fit well with the embedded servlet container model (the subject of the Auto-Configuration lesson).

Model, ModelMap, and ModelAndView: Three Ways to Carry Data to the View

Spring MVC Fundamentals' "Model: Carrying Data from Controller to View" section covered `Model` -- but it's not the only way to get data from a controller to a view:

```
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}
    }
}
```

All three end up in the same place: a String-keyed data map the view reads from. `Model` is a narrow interface that extends `ModelMap`; `ModelMap` can also be used directly like a `java.util.Map`. `ModelAndView` bundles both (the data and the view name) into a single return value -- for a controller like this project's `TopicController.show`, where the view name is always the same (`"topic"`) but which sections get rendered depends on a flag (`contentAvailable`), the split between a `Model` parameter and a `String` return value is usually more readable; `ModelAndView` is more useful when the view name itself is what varies.

What Is Thymeleaf? The "Natural Templating" Philosophy

The idea that sets Thymeleaf apart from other template engines is that a template is meant to be **both** valid HTML **and** a processable template at the same time:

```
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="${message}"` is an HTML attribute -- a browser that doesn't recognize it simply ignores it and shows the plain text inside the tag ("This is placeholder text..."). On the server side, once Thymeleaf processes it, that text is replaced with the actual value of `${message}`. That's something JSP's `<% %>` scriptlets, or the `{{ }}` syntax of engines like Mustache, **can't** do -- a file containing them looks broken when opened directly in a browser or a design tool. This is exactly why this project's own `templates/topic.html` is valid HTML a designer (or you) can preview directly in a browser without ever running Thymeleaf.

Variable Expressions: Reading Model Data with ${...}

`${...}` is the basic way to read data placed into the `Model`:

```
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>
    }
}
```

Notice the parentheses in `${topic.title()}` -- since `Topic` is a `record`, its accessor isn't `getTitle()`, it's `title()` directly. That's not an arbitrary syntax choice: this project's own `fragments/layout.html` accesses the `CourseNav` / `CategoryNav` / `TopicNavItem` records the exact same way (`course.name()`, `category.slug()`, `topicItem.title()`) -- we'll see that in the actual file in "This Project's Own Layout: fragments/layout.html and the Sidebar Accordion." Index access like `${tags[0]}` also works directly on lists.

Link Expressions: Building URLs with @{...}

`@{...}` builds a URL -- you don't need separate string concatenation for path variables and query parameters:

```
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>
    }
}
```

Inside `@{/topics/{slug}(slug=${slug}, lang=${lang})}`, the parenthesized part plays two roles: if a path placeholder named `{slug}` already exists in the path, the matching parameter (`slug=${slug}`) is substituted into it; any remaining parameters (`lang=${lang}`) automatically become a query string like `?lang=en`. This is exactly how this project's own

`topic.html` used to build its links -- `th:href="@{/topics/{slug}(slug=${topic.slug}, lang=${otherLanguage.code})}"` -- before an SEO-driven redesign moved language into the URL path itself. Today the same line reads `th:href="@{/{lang}/topics/{slug}(lang=${otherLanguage.code}, slug=${topic.slug})}"` -- both `{lang}` and `{slug}` are now path placeholders, so there's nothing left over to spill into a query string; either way, it's the view-side counterpart of the same path/query distinction we read on the server side with `@PathVariable` / `@RequestParam` in Path Variables and Request Parameters.

Displaying Text: th:text vs. th:utext

`th:text` always **escapes** its output (encodes HTML special characters); `th:utext` ("unescaped text") writes it out verbatim:

```
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
    }
}
```

Escaping is the default **and** the safe behavior -- if a user-supplied string contains a `<script>` tag, `th:text` turns it into harmless plain text. This project's `topic.html` uses `th:utext="${contentHtml}"` -- so it does **not** escape -- but that's a deliberate exception: `contentHtml` isn't user input, it's trusted HTML that `MarkdownService` produces **on the**

server, from `.md` files in the repo (see the accompanying comment in `topic.html`). Any text that could come from a user (a future comment form, for example) should always be rendered with `th:text`.

Conditional Rendering: th:if and th:unless

`th:if` **removes the tag from the output entirely** when its condition is falsy -- it doesn't hide it like `display:none`, it never reaches the HTML at all; `th:unless` checks the opposite condition:

```
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>
    }
}
```

This project's `topic.html` uses exactly this pair: `th:if="${!contentAvailable}"` for the "not available in this language" warning, and `th:if="${contentAvailable}"` for the actual content block -- the two never render at the same time because the conditions are exact opposites of each other. A `null` check works through the same mechanism: `th:if="${previousTopic != null}"` keeps the "Previous" link from showing up at all on the first topic (the `th:if="${previousTopic != null}"` line in the navigation block).

Loops: Rendering Lists with th:each

`th:each` repeats the tag it's on once for every element in a collection:

```
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>
    }
}
```

The `iterStat` in `topic, iterStat : ${topics}` is an optional **status variable** -- it carries fields like `count`, `index`, `size`, `first`, `last`, `even`, `odd`. This project's sidebar (`fragments/layout.html`) never uses the status variable (`th:each="topicItem : ${category.topics()}"`) because it doesn't need to; but when you want to style the last item in a list differently (as

in this lesson's "Appendix: Mini Project — A Simple Blog Page"), `iterStat.last` is exactly what it's for.

Message Expressions: i18n with #{...}

`#{...}` is the Thymeleaf counterpart of the `messages*.properties` mechanism from the i18n lesson -- it turns a key into text resolved for the current locale:

```
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").
    }
}
```

In a real Thymeleaf template this is one line: `th:text="#{topic.unavailable (${languageName})}"`. What happens behind the scenes is exactly what `TopicController.buildUnavailableMessage` does by hand -- picking a bundle from a `MessageSource` based on locale and filling in `{0}`-style placeholders; the difference is that Thymeleaf does this automatically for every `#{...}` it sees. This project's `topic.html` already relies on this for UI text like `#{nav.previous}`, `#{breadcrumb.home}`, `#{toc.onThisPage}` -- the fact that `buildUnavailableMessage` is written by hand is a **special** case where the word order differs enough between Turkish and English that a single `{0}` placeholder isn't enough (see the corresponding Javadoc in `TopicController`).

Fragments: th:fragment, th:insert, and th:replace

A piece of markup that repeats across a site (a navbar, a footer, a card component) is defined once with `th:fragment` and pulled in wherever it's needed with `th:insert` / `th:replace`:

```
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
    }
}
```

The difference is exactly one thing: `th:insert` places the fragment **inside the host tag** (the host tag stays); `th:replace` **swaps the host tag out** (the fragment's own root tag takes its place). This project's `topic.html` always uses `th:replace`, as in `<div th:replace="~{fragments/layout :: navbar}"></div>` -- because that `<div>` itself has no reason to survive into the output, it's only there to mark the fragment's location. The `fragments/layout` in `~{fragments/layout :: navbar}` points at a separate file; the `::` in this example's `~{::badge(...)}` means "this same template."

SpringEL Selection Expressions: .?[...] and #vars

`.?[...]` filters a collection -- the condition inside the square brackets is evaluated once per element, with `#this` bound to that element:

```
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
    }
}
```

> Inside a selection expression, `#this` rebinds the **entire evaluation scope** to the current element -- so referring to an outer context variable by its bare name (`activeSlug`) inside the brackets no longer looks up that variable, it looks for a field with that name on the element instead, and fails. `#vars.activeSlug` bypasses that rebinding and reaches straight into the top-level context variables. This project's real `fragments/layout.html` sidebar hit exactly this trap while computing `categoryIsActive` -- `.?[#this.slug() == activeTopicSlug]` threw a `SpelEvaluationException`, and `.?[#this.slug() == #vars.activeTopicSlug]` fixed it (see "Known Constraints" in the project notes).

This Project's Own Layout: fragments/layout.html and the Sidebar Accordion

Every mechanism in this lesson can be seen in the project's own `templates/fragments/layout.html` and `templates/topic.html` files. `layout.html` defines three fragments: `navbar`, `sidebar`, `footer` -- `topic.html` and `index.html` pull them in with `th:replace` (exactly the mechanism from "Fragments: th:fragment, th:insert, and th:replace").

The sidebar's category accordion brings together almost every topic in this lesson in one place: `th:each="category : ${course.categories()}"` walks each category ("Loops: Rendering Lists with th:each"), `th:with` computes two local variables named `categoryId` and `categoryIsActive` (`categoryIsActive`, using exactly the `.?[...]` + `#vars` pattern from "SpringEL Selection Expressions: .?[...] and #vars"), `th:classappend="${categoryIsActive} ? 'show' : ''"` adds a conditional CSS class, and `th:each="topicItem : ${category.topics()}"` lists that category's topics. It works together with Bootstrap's own `data-bs-toggle="collapse"` mechanism -- Thymeleaf only computes the right `id`/`aria-expanded`/CSS classes, the expand/collapse animation itself is entirely Bootstrap's JavaScript.

Form Binding (A Quick Look): th:object and th:field

This project doesn't have a form yet -- every page is read-only content. But Thymeleaf's Spring-specific form dialect offers `th:object`/`th:field` for binding any future `@ModelAttribute`-backed form:

```
// 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` determines which object `*{...}` expressions (starred, unlike `${...}`) resolve against; `th:field="*{author}"` both reads that object's `author` field as the `value` and derives the `id` / `name` attributes from the field name -- that same `name` is what Spring's `DataBinder` writes back into when the form is `POST`ed. Whenever this project adds a form (a comment form, for instance), this mechanism becomes a direct counterpart to the `@RequestBody` / `HttpMessageConverter` pair from Request and Response Handling -- one binds a JSON body to an object, the other binds form fields.

MVC (Server-Side Rendering) vs. REST: Which One, When?

Spring MVC Fundamentals' "@Controller vs. @RestController: Which One, When?" section drew this line at the annotation level; now that we know the view layer too, we can compare the outcomes. Server-side rendering (`@Controller` + Thymeleaf, the path this project takes) sends the browser directly viewable HTML -- the first page load feels faster (there's no waiting on JavaScript to fetch data and build the DOM), SEO works naturally (a search engine already sees HTML), but every page transition needs a full page load (or at least a server round-trip). REST (`@RestController` + JSON, the kind of API a single-page-application consumes) sends the client only data -- the client side (React, Vue...) turns it into DOM; page transitions can feel smoother, but the first load is heavier and SEO needs extra effort (server-side rendering, prerendering). This project is a learning site -- content is largely static, SEO matters, and it doesn't need complex client-side interaction -- so server-side rendering is a deliberate choice (the same kind of reasoning as the blocking/non-blocking trade-off in Spring MVC Fundamentals' "Spring MVC vs. Spring WebFlux (A Quick Look)").

Best Practices

- **Only use `th:utext` for trusted, server-generated content** -- any text that could come from a user must be escaped with `th:text` (see "Displaying Text: th:text vs. th:utext"); this project's `contentHtml` exception is deliberate and documented, not the default.
- **Inside selection/projection expressions (`.?[...]` , `.^[...]` , `.![...]`), always reach outer variables with `#vars.`** -- forgetting that `#this` changes scope caused a real bug in this project's sidebar (see "SpringEL Selection Expressions: .?[...] and #vars").
- **Prefer `th:replace` for fragments unless the host tag actually needs to survive** -- unnecessary `<div>` wrappers can produce unexpected gaps in CSS, especially in flex/grid layouts.
- **Put only what rendering needs into the view, not business logic** -- none of the three mechanisms in "Model, ModelMap, and ModelAndView: Three Ways to Carry Data to the View" restrict how the view uses that data; that discipline is on the developer -- the same idea as Spring MVC Fundamentals' "keep controllers thin, push logic to the service layer," applied here to the view layer.

Common Mistakes

1. Making `th:utext` a habit instead of `th:text`. Reaching for `th:utext` because "it's not working" and forgetting to switch back opens the door to XSS in any field that can contain user input -- `th:text`'s escaping is almost always the behavior you **want** (see "Displaying Text: th:text vs. th:utext").

2. Referring to an outer variable by its bare name inside a `.?[...]`. Because `#this` changes scope, the lookup targets a field on the element instead of the variable you meant, usually resulting in a `SpelEvaluationException` (see "SpringEL Selection Expressions: .?[...] and #vars").

3. Accessing a record field in `${...}` with `.title` (no parentheses). On a bean-style class, a `getTitle()` getter corresponds to the property `title`, but on a record the accessor is a real method (`title()`) -- `${topic.title}` can silently return `null` or fail depending on the case; the safe form is always `${topic.title()}` (see "Variable Expressions: Reading Model Data with ${...}").

4. Mixing up `th:insert` and `th:replace`. Both pull in a fragment, but one keeps the host tag and the other takes its place -- picking the wrong one shows up as an unexpected extra wrapper tag in the output (`th:insert`) or a missing wrapper you were relying on (`th:replace`) (see "Fragments: th:fragment, th:insert, and th:replace").

5. Not noticing that a path placeholder in `@{...}` doesn't match the parameter name. Writing `@{/topics/{slug}(id=${slug})}`, the parameter name in the parentheses (`id`) doesn't match the placeholder in the path (`{slug}`), so the placeholder is never filled and `id` falls through to the query string -- the result is a broken URL like `/topics/{slug}?id=...` (see "Link Expressions: Building URLs with @{...}").

6. Assuming a `#{...}` key is defined in both languages (tr/en `messages*.properties`). A missing key silently shows up on the page as something like `??key??` -- it isn't caught at build time, only when someone actually visits that page in that language (see "Message Expressions: i18n with #{...}").

Summary, Cheat Sheet, and Glossary

Thymeleaf is Spring MVC's default view technology -- its "natural templating" philosophy lets a template be both valid HTML and a processable file at the same time. Key points:

- `Model` / `ModelMap` / `ModelAndView`: three equivalent ways to carry data from a controller to a view
- `${...}`: variable expression, reads data from the `Model` (parenthesized access on records: `topic.title()`)
- `@{...}`: link expression, automatically combines the context path with path variables and query parameters

- `#{...}`: message expression, returns text resolved from an i18n bundle (in this project, through Spring's `MessageSource`)
- `th:text` / `th:utext`: escaped and unescaped text output, respectively
- `th:if` / `th:unless`: conditional blocks that remove the tag from rendering entirely
- `th:each`: walks a collection and repeats the tag for every element; `iterStat` gives access to index/count/first/last
- `th:fragment` / `th:insert` / `th:replace`: defining and pulling in a reusable chunk of markup (`th:replace` takes the host tag's place)
- `.?[...]`: selection (filtering) expression; `#this` inside it is bound to each element, outer variables are reached with `#vars.`
- `th:object` / `th:field`: bind form fields to a Java object (not yet used in this project)

Quick reference:

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

<!-- conditional + loop -->
<div th:if="${!items.isEmpty()}">
    <p th:each="item, stat : ${items}" th:text="${stat.count} + '. ' + ${item.name()}">row<
/p>
</div>
<div th:unless="${!items.isEmpty()}">Empty.</div>

<!-- fragment definition and call -->
<div th:fragment="card(title)" class="card" th:text="${title}">card</div>
<div th:replace="~{::card(${topic.title()})}">placeholder</div>

<!-- safe vs. trusted content -->
<p th:text="${userComment}">from a user -- escaped</p>
<article th:utext="${serverRenderedMarkdown}">server-generated -- unescaped</article>
```

Glossary

Thymeleaf — The Java template engine Spring Boot auto-configures by default, built around the "natural templating" philosophy.

Natural templating — Thymeleaf's design principle that a template should look valid and meaningful in a browser or design tool even before it's processed.

Variable expression (`${...}`) — A Thymeleaf expression that reads a value from the model/context.

Link expression (`@{...}`) — A Thymeleaf expression that builds a URL by combining the context path with path variables and query parameters.

Message expression (`#{...}`) — A Thymeleaf expression that resolves an i18n key into text for the current locale.

`th:text` / **`th:utext`** — Attributes that write a tag's text content escaped and unescaped, respectively.

Fragment — A reusable chunk of template defined with `th:fragment` and pulled in elsewhere with `th:insert` / `th:replace`.

Selection expression (`.?[...]`) — An expression that filters a collection based on a condition where `#this` is bound to the current element.

`#vars` — A Thymeleaf basic object that reaches straight into the top-level context variables from inside scope-changing expressions like selections and projections.

`th:object` / **`th:field`** — Thymeleaf's form dialect attributes that bind a form field to a field on a Java object arriving via `@ModelAttribute`.

Appendix: Mini Project — A Simple Blog Page

Building a small page that brings this lesson's mechanisms together: a `th:fragment` (a single post card), `th:each` (the list of posts), and `th:if` / `th:unless` (the empty-list state), all in one template:

```
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>
    }
}
```

The `postCard` fragment only knows about one post -- a title and an excerpt. The `th:unless="${#lists.isEmpty(posts)}"` block kicks in when there's at least one post and uses `th:each` to `th:insert` `postCard` for each one; when the list is empty, `th:if="${#lists.isEmpty(posts)}"` shows "No posts yet." on its own. `BlogPageDemo` calls the same `render` method first with a populated list, then with an empty one, to show both branches working correctly.

Appendix: Mini Project — An i18n Product Card Template

The second mini project brings `${...}`, `@{...}`, and `th:if` together, using the same approach as "Message Expressions: i18n with #{...}" (resolving the message first, then feeding an already-resolved string into the template):

```
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` renders a product's name (`${product.name()}`), a link to its product page (`@{/products/{slug}(slug=${product.slug()})}`), and a badge that only shows up when it's discounted (`th:if="${product.discounted()}"`). `ProductCardDemo` resolves the "Add to Cart"/"Sepete Ekle" label per language ahead of time and runs the same template in both languages, for both a discounted and a regular-priced product -- in a real Thymeleaf setup that last step would collapse into `#{addToCart}` in one line, but keeping it separate lets the template itself be tested independently of the message-resolution infrastructure.