Thymeleaf es un motor de plantillas para aplicaciones Java, diseñado para trabajar en entornos web. Se integra bien con Spring Boot y facilita la generación de vistas en HTML que contienen datos dinámicos desde el servidor.

Instalación e integración con Spring Boot

Para poder trabajar con Thymeleaf hay que incluir las dependencias spring web y Thymeleaf en el asistente inicial en springboot.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Las plantillas Thymeleaf deben ubicarse en: **src/main/resources/templates**

En un proyecto Spring Boot, los recursos estáticos (imágenes, CSS, JavaScript) deben estar ubicados en el directorio **src/main/resources/static**

1.- Crear un proyecto con estas dependencias.

2.- En main/resources/templates creamos un archivo html: index.html

Para poder añadir páginas HTML tenemos que hacer lo siguiente: en eclipse, Help/Install New Software: Escribimos en el buscador HTML y marcamos Web, XML, Java EE … e instalamos. Reiniciamos eclipse STS.

Sobre la carpeta main/resources/templates, botón derecho, new Other, html file

3.- En el index.html escribimos nuestro nombre en <h1>

4.- A continuación, escribir una etiqueta <p> y poner un texto dentro de color rojo.

5.- Lanzar la aplicación en https://localhost:8080 y …

6.- Necesitaremos crear un controlador, tal y como se explica en el apartado siguiente.

Configuración de un controlador básico:

Necesitamos crear un controlador. Es una clase Java, que estará dentro de un paquete

controller. Debe tener la etiqueta **@Controller:**

```
@GetMapping("/")
  public String mostrarInicio() {
    return "index"; // busca templates/index.html
  }
```

¿Cómo pasar información desde el controlador a la vista? Usamos la interfaz **Model:**

La clase Model se usa para pasar datos desde el controlador a la vista html. Model utiliza el método addAttribute para agregar datos al modelo que se envía a la vista.

model.**addAttribute**("clave", valor);

·"**clave**": Es el nombre con el que se puede acceder al atributo en la vista.

·**valor**: Es el objeto o valor que se desea pasar (puede ser cualquier tipo de dato, como un string, un número, una lista, un mapa, etc.).

Creamos un paquete en el proyecto con nombre controller y dentro, una clase HomeController con el siguiente código. Un controlador en Spring envía datos a las plantillas Thymeleaf usando el objeto **Model y el método addAttribute.** Por ejemplo:

```
@Controller
public class HomeController {

    @GetMapping("/")
    public String home(Model model) {
        model.addAttribute("mensaje", "Bienvenido a Thymeleaf");
        return "home";
    }
}
```

Cuando el controlador devuelve una cadena, Spring lo interpreta como el **nombre de la vista** que debe renderizar. En el ejemplo, return “home”, Spring buscará una plantilla llamada home.html (en src/tmain/resources/templates y la renderizará.

Creación de una plantilla básica: <html xmlns:th="http://www.thymeleaf.org">

Crea un archivo home.html en src/main/resources/templates:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Página de Inicio</title>
</head>
<body>
    <h1 th:text="${mensaje}">Texto por defecto</h1>
</body>
</html>
```

Al abrir la aplicación en un navegador, verás el texto: "Bienvenido a Thymeleaf".

Sintaxis básica de Thymeleaf

Atributos básicos:

- **th:text**: Cambia el texto de un elemento (de forma segura, no interpreta etiquetas html)

  <p **th:text**="'Hola, ' + ${nombre}">Texto de ejemplo</p>

- **th:utext**: útil cuando necesitas que el contenido dinámico incluya HTML que debe interpretarse en lugar de mostrarse como texto literal.
  En Thymeleaf, el símbolo **${}** se utiliza para acceder a expresiones de contexto, como variables o datos del modelo, y generalmente se usa en atributos como th:text, th:value, th:href, etc.., cuando necesitas insertar datos dinámicos.

- **th:href:**Construyen URLs de forma dinámica a partir del contexto. Suele ir en <a>, <link>,<script>...

  <a **th:href**="@{/contacto}">Contacto

Donde **/contacto es una ruta definida en el controlador.** El controlador procesa la solicitud y devuelve la vista que devuelve el método.

Ejemplo:

```
@GetMapping("/contacto")
 public String contacto() {
                return "contacto"; // Retorna la plantilla Thymeleaf "contacto.html"
}
```

- **th:src:** Cambia rutas de imágenes.

```
<img th:src="@{/imagenes/logo.png}" alt="Logo">
```

Practica:

- En el método home() del controlador HomeController, pasar un nuevo atributo nombre1 (con vuestro nombre) a la vista y pintarlo en **home.html** en un párrafo `que ponga:` Hola, nombre.

- Pasar otro atributo llamado nombre2, cuyo valor debe ser una cadena con vuestro nombre pero con la etiqueta <b>. Pintar en la vista dos veces dicho atributo, uno escapado y otro sin escapar.

- Añadir una imagen a la carpeta src/main/resources/static/imagenes y verla en la vista home.html.

- A continuación, crear un enlace en la vista. Al pulsarlo, cargar la vista index.html. (Usar la ruta /index). Crear la vista index.html con un h1.

Estructuras de control:

✓ **Condicionales**:

- **th:if:** Muestra contenido si la condición es verdadera.
- **th:unless:** Muestra contenido si la condición es falsa.

```
<p th:if="${activo}">El usuario está activo</p>
<p th:unless="${activo}">El usuario no está activo</p>
```

✓ **Bucles**:

- **th:each:** Se usa para iterar sobre listas:

```
<ul>
  <li th:each="producto : ${productos}" th:text="${producto.nombre}"></li>
</ul>
```

Practica:

- En el método home del controlador HomeController, pasarle otro atributo a la vista: role con el valor 'admin'. En la vista mostrar un párrafo indicando si es o no administrador.

- Crear a continuación en el método home., un array de cadenas que contenga 5 nombres propios. PaRar el array a la vista home.html y escribir una lista con los nombres del array.
- Crear un enlace en la vista. Al pulsarlo, cargar la vista form.html. (Usar la ruta /form). Crear la vista form.html con un h1.

Formularios: los formularios HTML se enlazan con objetos del **modelo** de Spring usando el atributo **th:object.** Ejemplo:

```
@GetMapping("/formulario")
public String mostrarFormulario(Model model) {
    Usuario usuario = new Usuario(); // Creamos objeto vacío
    model.addAttribute("usuario", usuario); // Lo añadimos al modelo
    return "formulario"; // vista Thymeleaf
}
```

```
<form th:action="@{/guardar}" th:object="${usuario}" method="post">
    <input type="text" th:field="*{nombre}" />
    <input type="email" th:field="*{email}" />
    <button type="submit">Enviar</button>
</form>
```

th:action="@{/guardar}" → define la URL a la que se enviará el formulario.
th:object="${usuario}" → enlaza el formulario con el objeto usuario del modelo.
th:field="*{nombre}" → conecta el campo con la propiedad nombre del objeto usuario.
El *{} indica que es relativo al objeto definido en th:object.

Si no se enlaza con un objeto model (sin etiqueta th:object):
IMPORTANTE: hay que poner los atribute name manual.

```
@GetMapping("/formulario")
public String mostrarFormulario(Model model) {
    Usuario usuario = new Usuario(); // Creamos objeto vacío
    model.addAttribute("usuario", usuario); // Lo añadimos al modelo
    return "formulario"; // vista Thymeleaf
}
```

```
<form th:action="@{/guardar2}" method="post">
    <label>Nombre:</label>
    <input type="text" name="nombre" th:value="${usuario.nombre}" />

    <label>Email:</label>
    <input type="email" name="email" th:value="${usuario.email}" />

    <button type="submit">Guardar</button>
</form>
```

Etiquetas y atributos:

Atributo	Función
`th:action`	Define la URL donde se enviará el formulario.
`th:object`	Asocia el formulario con un objeto del modelo.
`th:field`	Vincula un campo del formulario con una propiedad del objeto del modelo.
`th:value`	Establece el valor de un input, select o textarea.
`th:checked`	Marca un checkbox como seleccionado según una condición.
`th:selected`	Selecciona una opción en un `<select>` según una condición.
`th:each`	Itera sobre colecciones, útil para generar opciones en `<select>`.
`th:if` / `th:unless`	Condicional para mostrar u ocultar elementos.

Ejemplos:

- th:action. El formulario se enviará a la URL que se indique en action:

  Puede ser estática: Ej: /guardar
  <form **th:action="@{/guardar}**" method="post"> ... </form>

  O Puede necesitar parámetros y construye dinámicamente: por ejemplo: /editar/id
  <form **th:action="@{/clientes/{id}(id=${cliente.id})}**" method="post">

- th:method: Define el método HTTP que se usará para enviar el formulario (`GET`, `POST`, etc.)

- th:field: Vincula un campo del formulario a una propiedad de un objeto del modelo (generalmente usado con un objeto gestionado por Spring). Este campo está vinculado a la propiedad `nombre` de un objeto del modelo:

  <input type="text" th:field="*{nombre}">

- th:object: Asocia un formulario completo con un objeto del modelo. Esto permite que todos los campos dentro del formulario se vinculen automáticamente a las propiedades de ese objeto. Ej: <form **th:object="${cliente}"** th:action="@{/form}" method="post">

- th:value: Establece el valor inicial de un campo del formulario.

  <input type="text" th:field="*{nombre}" **th:value**="${producto.nombre}">

- th:checked: Usado con checkboxes para establecer si están seleccionados o no.
  <input type="checkbox" th:field="*{activo}" th:checked="*{activo}">

- th:selected: Usado en listas desplegables (<select>), establece qué opción debe estar seleccionada según el valor del modelo

  <select th:field="*{categoria}">
  <option value="1" th:selected="${producto.categoria == 1}">Categoría 1</option>
  <option value="2" th:selected="${producto.categoria == 2}">Categoría 2</option>
  </select>

- th:placeholder: Define un texto de ayuda que aparece dentro de un campo de entrada cuando está vacío
  <input type="text" th:field="*{nombre}" th:placeholder="Nombre del producto"">

- th:errors: Muestra mensajes de error relacionados con un campo específico cuando se produce una validación fallida

  Error en el nombre

Cuando queremos enviar datos del formulario se utiliza: @ModelAttribute. Spring MVC los convierta automáticamente en un objeto Java. Es especialmente útil cuando quieres recibir un objeto completo en lugar de parámetros individuales.

```
<form th:action="@{/guardar}" th:object="${usuario}" method="post">
    <input type="text" th:field="*{nombre}" />
    <input type="email" th:field="*{email}" />
    <button type="submit">Enviar</button>
</form>
```

```
@PostMapping("/guardar")
public String guardarUsuario(@ModelAttribute Usuario usuario) {
    System.out.println(usuario.getNombre());
    return "resultado";
}
```

Paso de parámetros por URL

@PathVariable: Se usa para mapear partes de la URL de una solicitud a parámetros de un controlador de Spring. Los parámetros dinámicos que vienen por la URL los usamos en nuestro endpoint. Ejemplo de controlador para el el endopoint: /hola/Pepe

```
@GetMapping("/hola/{name}")
public String helloName(@PathVariable String name){
    return "Hello " + name;
}
```

@RequestParam: en **Spring MVC** se usa para **obtener valores de parámetros de la URL (después de ?) o de un formulario**, generalmente **campos individuales**, en lugar de mapear todo a un objeto como hace @ModelAttribute. Es decir, mientras @ModelAttribute convierte varios campos en un **objeto completo**, @RequestParam obtiene **valores sueltos**.

Ejemplo de parámetros por URL**: /buscar?nombre=pan&categoria=2**

```
GetMapping("/buscar")
public String buscar(
      @RequestParam String nombre, @RequestParam Long categoria,
      Model model) {

   // Ejemplo: llamar al servicio con los filtros
   List<Ingrediente> resultados = ingredienteService.buscar(nombre, categoria);

   model.addAttribute("resultados", resultados);
   model.addAttribute("nombre", nombre);
   model.addAttribute("categoria", categoria);

   return "ingredientes/lista";
}
```

La llamada en Thymeleaf sería:

```
<a th:href="@{/buscar(nombre=${nombre}, categoria=${categoria})}">
      Buscar
</a>
```

Formulario HTML:

```html
<form th:action="@{/guardar}" method="post">
    <label>Nombre:</label>
    <input type="text" name="nombre" />

    <label>Email:</label>
    <input type="email" name="email" />

    <button type="submit">Guardar</button>
</form>
```

Controlador usando `@RequestParam`:

```java
@PostMapping("/guardar")
public String guardarUsuario(
        @RequestParam("nombre") String nombre,
        @RequestParam("email") String email) {

    System.out.println("Nombre: " + nombre);
    System.out.println("Email: " + email);

    return "resultado";
}
```

Flujo Básico de un Formulario en Thymeleaf y el Controlador

1. **Cargar la Página del Formulario** (GET):
   - El controlador maneja una solicitud GET para cargar la página del formulario.
   - Se inicializa un objeto vacío en el modelo para que Thymeleaf pueda vincular los campos del formulario: (th:object y th:field).
     th:field se utiliza para enlazar un campo del formulario directamente con un objeto del modelo, lo que permite la vinculación bidireccional. No solo muestra datos del modelo, sino que también permite capturar los datos ingresados por el usuario en el formulario y devolverlos al modelo.

2. **Enviar el Formulario** (POST):
   - Cuando el usuario envía el formulario, el controlador maneja la solicitud POST.
   - Spring Boot automáticamente vincula los datos del formulario a un objeto Java (**@ModelAttribute** para mapear automáticamente los datos del formulario al objeto con los datos del formulario)
   - El controlador puede procesar los datos y devolver una respuesta adecuada.

3. **@RequestParam:** se utiliza cuando se quiere capturar valores pasados en la URL para usarlos dentro de tu lógica del controlador:
   Ejemplo para la Url: http://localhost:8080/saludo?nombre=Juan

```java
@GetMapping("/saludo")
public String saludar(@RequestParam("nombre") String nombre) {
    return "Hola, " + nombre + "!";
}
```

También se usan para capturar valores enviados por diversos elementos HTML de un formulario: text, checkbox, radio, etc.. todo a través de sus campos name.

```html
<form th:action="@{/guardar}" th:method="post">
    <label>
        <input type="checkbox" name="intereses" value="deportes"> Deportes
    </label>
    <label>
        <input type="checkbox" name="intereses" value="musica"> Música
    </label>
    <button type="submit">Enviar</button>
</form>
```

El controlador:

```
@PostMapping("/guardar")
public String guardar(@RequestParam List<String> intereses) {
    System.out.println("Intereses seleccionados: " + intereses);
    return "resultado"; // Retorna una vista llamada "resultado"
}
```

4. Para hacer una redirección en el controlador se usa redirect:/endpoint
    - Ejemplo: **return** "redirect:/clientes";

Practica:

- En la vista form.html, crear un formulario con dos input de tipo text: uno para el nombre y otro para la edad. El formulario debe tener un atributo **th:object llamado user** para vincular ese objeto con los datos del formulario.

- ¿Qué tenemos que hacer para poder un objeto User? Crear en una carpeta model una clase User.java con dos atributos privados: nombre y edad.
  Para vincular el **objeto user** al formulario, antes de cargar el formulario.html, el controlador debe pasarle a la vista el objeto user. ¿Cómo lo harías?

- Añadir un botón a la vista form.html llamado Enviar de tipo submit. (Necesitará que el formulario tenga method = "post"). Para el action usar la ruta /form.

- Añadir la funcionalidad del botón submit. Es necesario un nuevo método en el controlador que sea llamado cuando la url sea /form y … ¿el method, cual seria?.
  Después de pulsar el botón, mostrar una página result.html que indique: el nombre, la edad y un mensaje: "Formulario enviado con éxito". Esos valores se tendrán que pasar desde el controlador.

Ejercicios:

Ejercicio 1: Partimos del código del ejercicio 6 del tema 4.

Crear la vista index.html que mostrará una tabla con todos los clientes. Esta vista se cargará al cargar localhost://8080/clientes.
La cabecera de la tabla tiene el id, el nombre y la ciudad. También muestra la columna Acciones
Cada fila muestra el id, el nombre, la ciudad del cliente y en la última columna, un enlace 'ver'.

Ejercicio 2:

El enlace 'ver' del ejercicio anterior cargará en una página el detalle del cliente (cliente-detalle.html) el id, el nombre, la calle y ciudad del cliente. Utilizad el método del controlador que devuelve los datos de un usuario.
Añadir un enlace 'volver' que vuelva al listado de todos los clientes.

Ejercicio 3:

En el listado de clientes, añadir un enlace, 'Agregar cliente', que redirija a una página cliente-formulario.html. Contiene un formulario para poder guardar un nuevo cliente dando nombre, calle y ciudad y un botón para guardar (submit). Al pulsar guardar, se deben de almacenar los datos del usuario. Utilizad el método que guardaba los datos.

Ejercicio 4:

En la página principal: index.html añadir al final un enlace nuevo: buscar por provincia.
Debe cargar una pagina ciudad.html que tenga un cuadro de texto y un botón. Al pulsar el botón, se debe de cargar la misma página con el valor en el cuadro de texto y un listado con los nombres de los clientes de dicha ciudad. Utilizad el método existente en el controlador que busca por ciudad.

Ejercicio 5:

Partimos del código del ejercicio 8 del tema 4.
Crear una página index.html con una lista con 3 opciones:

```
<ul>
<li><a>Ver todos los usuarios</a></li>
<li><a>Añadir un nuevo usuario</a></li>
<li><a>Primer Usuario disponible</a></li>
</ul>
```

Crear la funcionalidad ‘Ver todos los usuarios’ que cargue una página html: usuarios.html con el listado de todos los usuarios. Debe mostrar, id, nombre, email y el estado del perfil.

Ejercicio 6:

Crear la funcionalidad ‘Añadir un nuevo usuario’ de la página index.html. Debe mostrar una página: usuario-form.html con 4 campos: nombre, email, biografía (tipo área) y estado. Al pulsar el botón submit, se guardará la información y se cargará automáticamente el listado de todos los usuarios.

Ejercicio 7:

Crear la funcionalidad ‘Primer Usuario disponible’ de la página index.html. Debe mostrar la página primerUsuario.html con un título ‘Primer Usuario disponible’, el nombre, el email y biografía. Si no hubiera ningún usuario con estado disponible, deberá mostrar un mensaje indicándolo en vez de mostrar lo anterior.

Ejercicio 8:

Añadir una nueva columna al final de la tabla de usuarios.html: Acciones. Tendrá un enlace con el texto ‘ver’ en cada una de las filas. Cuando se pulse el enlace, se cargará la página usuario-form con los datos del usuario pulsado. En vez de aparecer el botón submit, mostrará un enlace para volver al listado.

Ejercicio 9:

Partimos del código del ejercicio 10 del tema 4.
Cuando se cargue el endpoint de los empleados (/empleados), se debe ver la página empleados-lista.html con una tabla y todos sus empleados. Para cada uno, su nombre, puesto, y email.

Ejercicio 10:

Añadir a la página empleados-lista.html un enlace para poder cargar un formulario(empleado-form.html) y añadir un nuevo empleado a la base de datos: Nombre, puesto y email. Al terminar debe cargar de nuevo el listado de empleados.

Ejercicio 11:

Añadir el endpoint /oficinas, se debe visualizar la página oficinas-lista.html. Y mostrará una tabla con el listado de todas las oficinas. Para cada una, ubicación y teléfono.

Ejercicio 12:

Incluir un enlace en oficinas-lista.html para poder añadir una nueva oficina (oficina-form.html). Debe mostrar un formulario para guardar la ubicación y el teléfono. También debe de mostrarse una lista de checkboxs con el conjunto de empleados que hay para que pueda seleccionar varios. (Utilizar @RequestParam).
Al final de la página, poner un enlace para ir a la página de alta de un nuevo empleado.
Cuando guarde en base de datos, volverá automáticamente al listado de oficinas.

Ejercicio 13:

En la página oficinas-lista.html, añadir una nueva columna al final: Acciones. Tendrá un enlace: 'Ver empleados'. Al pulsarlo, debajo de la tabla, en la misma página, se cargará una lista con los empleados de la oficina de la forma: nombre (puesto).

Ejercicio 14:

Partimos del código del ejercicio 13 del tema 4.
Crear una página: autores-lista.html, que muestre una tabla con los autores y sus respectivos libros.
La tabla tiene dos columnas, una para el nombre y otra para los libros, en una lista.
Se cargará la página cuando se llame a /autores.

Ejercicio15:

Crear la página autor-formulario.html para dar de alta un nuevo autor. Guardará su nombre y una vez almacenado en BD, automáticamente, se cargará la página del listado de autores.

Ejercicio 16:

Crear una nueva página libro-formulario.html para añadir un libro a un autor existente. Esta página mostrará un formulario con un campo para el título, y un desplegable donde elegir el autor. (select y option para cada autor existente).
Al guardar, redireccionar a la página de autores.

Ejercicio 17:

Crear una página: buscar-autores.html. Tendrá un formulario con input de tipo text donde introducir el nombre a buscar. Al pulsar el botón de tipo submit, mostrará debajo una lista con los autores que haya encontrado. Si no hay ninguno, escribir, No se encontraron autores con ese nombre.

Ejercicio 18:

Partimos del código del ejercicio 15 del tema 4.

Crear una página index.html que se cargue al iniciar la aplicación. Tendrá los siguientes enlaces:
- Mostrar listado de cursos
- Alta de curso
- Alta de estudiante
- Buscar estudiantes por nombre

Ejercicio 19:

Crear la página cursos-lista.html que muestre una tabla con todos los cursos. Para cada curso mostrará el nombre, la descripción, y la tercera columna tendrá una lista con los nombres de los estudiantes. (cursos/listado). Poner también un enlace para cargar el menú inicio.

Ejercicio 20:

Crear la página curso-formulario.html para dar de alta un curso. Será un formulario con el nombre y la descripción del curso. El botón guardar almacenará el curso en base de datos y cargará a continuación la lista de cursos. (cursos/crear , cursos/guardar).

Ejercicio 21:

Crear la página estudiante-formulario.html para dar de alta un estudiante. Mostrará el nombre, el email y un desplegable múltiple con los cursos existentes. (estudiantes/crear).
Al guardar, se mostrará el listado de cursos.

Ejercicio 22:

Crear una página estudiante-buscar.html (/estudiantes/buscar), que tendrá un formulario con un input para escribir un nombre y un botón. El botón buscará el conjunto de estudiantes y devolverá una tabla en la misma página, (en un div). Mostrará el nombre, email y una tercera columna con una lista y sus cursos. Al final, un enlace para ver el listado de cursos.