Utilización de ModelAndView con Spring MVC y Thymeleaf como motor de plantillas

¿Qué es Thymeleaf?

Thymeleaf es un motor de plantillas que nos permite crear páginas HTML con contenido dinámico.

Dicho contenido, en nuestro caso, es enviado por los controladores a través de un objeto llamado **ModelAndView**. Este objeto contiene tanto la información (Model) con la que la página será llenada, así como de el nombre de la página a la que será enviada la información (View).

Para esto, tenemos que crear nuestra página HTML con ciertos tags especiales propios de Thymeleaf, que le indicarán que dicho tag debe ser reemplazado por los datos contenidos en el objeto ModelAndView.

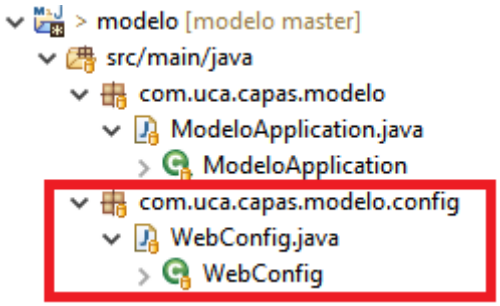
Sin embargo, para poder utilizar Thymeleaf, debemos de configurarlo en nuestra aplicación de Spring MVC.

Configuración de Thymeleaf en Spring MVC

1. Agregar la dependencia de Thymeleaf en nuestro pom.xml:

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

2. Debemos crear una clase que utilizará Spring para configurar y crear los objetos necesarios para poder utilizar Thymeleaf. Para esto crearemos un nuevo paquete de nombre **com.uca.capas.model.config,** el cual contendrá todas nuestras clases de configuración (en este caso solo será una), y dentro crearemos la clase **WebConfig.java:**



3. El contenido de dicha clase **WebConfig.java** será el siguiente (se irá por partes debido al tamaño de la clase):

```java
@Configuration
public class WebConfig {

    @Bean
    public ViewResolver viewResolver() {
        ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();

        viewResolver.setTemplateEngine(templateEngine());
        viewResolver.setCharacterEncoding("UTF-8");

        return viewResolver;
    }
```

- **@Configuration**: Esta anotación es utilizada por Spring para identificar una clase como de configuración, al iniciar el aplicativo y escanear todas las clases, si encuentra esta anotación entonces utiliza los métodos dentro de el para configurar dichos objetos.
- **@Bean**: Esta anotación hace que se ejecute el método y se obtenga el objeto de retorno para que sea utilizado por Spring en el aplicativo.

Ahora bien, el primer objeto que debemos configurar es el ViewResolver, el cual, como hemos visto en clases anteriores, es el encargado de manejar los archivos que servirán para la vista, en nuestro caso HTML. Para esto, crearemos un método (sin parámetros) que devuelva un objeto de tipo **ViewResolver,** luego, instanciaremos un objeto de tipo **ThymeleafViewResolver** (porque estaremos utilizando Thymeleaf para el manejo de las vistas).

Utilizaremos, de este nuevo objeto, el método **setTemplateEngine**, el cual recibirá lo que devolverá el método **templateEngine()** (y que veremos más adelante).

Como último paso, utilizamos el método **setCharacterEncoding** y le mandamos como parámetro el string "UTF-8".

Por último, devolveremos el objeto.

Método templateResolver()

```java
@Bean
public ClassLoaderTemplateResolver templateResolver() {

    ClassLoaderTemplateResolver templateResolver = new ClassLoaderTemplateResolver();

    templateResolver.setPrefix("templates/");
    templateResolver.setSuffix(".html");
    templateResolver.setTemplateMode("HTML");
    templateResolver.setCharacterEncoding("UTF-8");

    return templateResolver;
}
```

Este método se utiliza para crear el objeto que tendrá la ubicación de donde se encontrarán nuestras vistas (archivos HTML) en nuestro proyecto, así como del tipo que estará manejando (HTML).

Devolverá un objeto de tipo **ClassLoaderTemplateResolver** y no tendrá parámetros.

Instanciamos un objeto de tipo **ClassLoaderTemplateResolver** y utilizaremos los siguientes métodos:

- **setPrefix:** Indica la carpeta de nuestro proyecto donde se encontrarán los archivos HTML, en este caso será sobre la carpeta **templates** que se encuentra en **src/main/resources**.
- **setSuffix:** Indica la extensión que tendrán los archivos de recurso, y que será **.html**
- **setTemplateMode:** Indica el tipo de archivos que serán manejados, como son HTML le enviaremos el string "HTML".
- **setCharacterEncoding:** La codificación que estaremos utilizando en nuestros archivos, y que será UTF-8.

Por último, devolvemos el objeto configurado.

Método templateEngine()

```
@Bean
public SpringTemplateEngine templateEngine() {
    SpringTemplateEngine templateEngine = new SpringTemplateEngine();
    templateEngine.setTemplateResolver(templateResolver());
    return templateEngine;
}
```

Este método se utiliza para indicarle a Spring el lugar de donde serán tomados los archivos de vista. Devolverá un objeto de tipo **SpringTemplateEngine**.

Instanciaremos un objeto de ese tipo, y utilizaremos el método **setTemplateResolver** el cual recibe de parámetro lo que devuelve el método **templateResolver()** que vimos anteriormente, recordemos que dicho método devuelve un objeto que contiene la información de la ubicación de los archivos de vista.

Método addResourceHandlers(ResourceHandlerRegistry)

```
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry
      .addResourceHandler("/resources/**")
      .addResourceLocations("/resources/");
}
```

Este método es utilizado para poder acceder a los recursos estáticos de nuestra aplicación, esto es archivos javascript, css, imágenes, etc. Sin esto, Spring no nos podrá proveer de dichos recursos al acceder a la aplicación desde el navegador web.

Este método no devolverá nada y recibirá de parámetro un objeto de tipo **ResourceHandlerRegistry**, el cual es provisto por Spring y sirve para indicarle el lugar donde se encuentran nuestros recursos estáticos.

Para esto, de este objeto provisto por parámetro, utilizaremos el método **addResourceHandler**, el cual recibirá el string "/resources/**", con esto le estamos diciendo que se encargue de manejar todos los recursos estáticos dentro de la carpeta **resources**.

Por último, tenemos que indicarle el lugar de estos recursos con el método **addResourceLocations**, y que recibirá de parámetro el string "/resources/".

Con esto, ya hemos configurado Thymeleaf con Spring MVC.

Ahora, abramos el archivo **index.html** ubicado en **templates/commons** en **src/main/resources** y busquemos el siguiente fragmento de código:

```
60      <div class="container-fluid">
61          <h1 class="mt-4">Env&iacute;o de objetos a p&aacute;gina</h1>
62          <div class="card mb-4">
63              <div class="card-body">
64                  <p class="mb-0">Este es un ejemplo de uso del objeto ModelAndView. Objeto enviado por el controlador:</p>
65                      <span th:text="${hora}" />
66              </div>
67          </div>
68      </div>
```

Si vemos en la línea 65 utilizamos el tag de HTML , sin embargo le hemos agregado el atributo **th:text="${hora}"**, el cual es un atributo especial de Thymeleaf con el que crea el tag **span** con el contenido que venga del modelo con nombre **hora**, el cual vendrá desde el controlador en el objeto **ModelAndView** y que hemos seteado utilizando el método **addObject.**

Ahora crearemos el controlador que será el encargado de devolver el objeto **ModelAndView.**

Crearemos el paquete **com.uca.capas.modelo.controller** y una clase en el llamada **MainController.java,** el cual tendrá el siguiente método:

```java
@RequestMapping("/index")
public ModelAndView index() {
    ModelAndView mav = new ModelAndView();
    mav.addObject("hora", Calendar.getInstance().getTime().toString());
    mav.setViewName("commons/index");
    return mav;
}
```

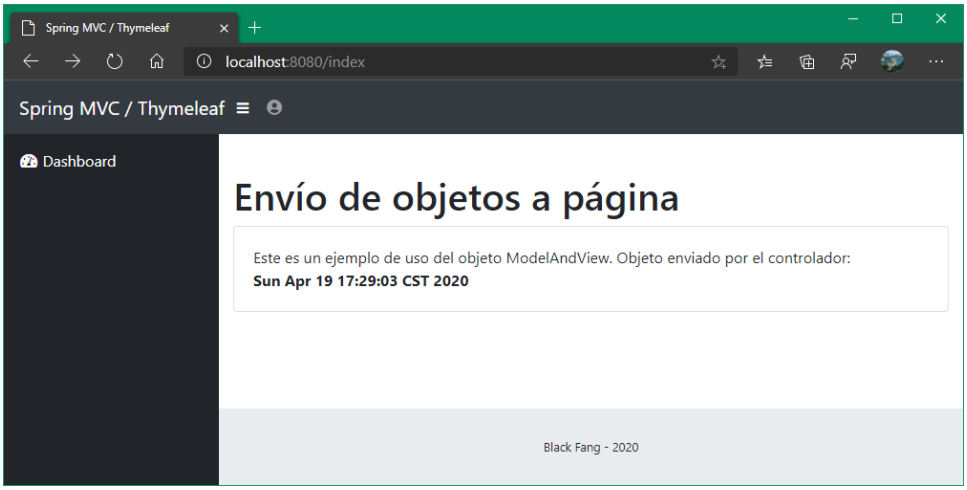
Como vemos, ahora el método, además de estar mapeado a la URL /index, devolverá un objeto **ModelAndView**, por lo que instanciamos un nuevo objeto de ese tipo.

Luego, tenemos que añadir el dato (modelo) que será reemplazado en nuestro **index.html**, específicamente en el tag **span**, para esto utilizamos el método **addObject** que recibe dos parámetros, primero el nombre con el que llamaremos al modelo (en este caso **hora**), y segundo el dato en sí (que puede ser cualquier objeto), para este ejemplo le mandaremos la fecha y hora actual.

Por último, devolvemos el objeto.

Levantemos el aplicativo ejecutando el método **main** de la clase **ModeloApplication** y al terminar ingresemos a la URL:

http://localhost:8080/index



Como vemos, el tag **span** ha sido reemplazado con el dato que viene en el ModelAndView.

También es posible enviar colecciones como listas en el ModelAndView, y recorrerlas utilizando atributos de Thymeleaf.

Al controlador anterior, agregaremos una colección al objeto ModelAndView y le llamaremos “colores”:

```java
@RequestMapping("/index")
public ModelAndView index() {
    ModelAndView mav = new ModelAndView();

    List<String> lista = new ArrayList<>();
    lista.add("Rojo");
    lista.add("Verde");
    lista.add("Azul");

    mav.addObject("hora", Calendar.getInstance().getTime().toString());
    mav.addObject("colores", lista);
    mav.setViewName("commons/index");
    return mav;
}
```

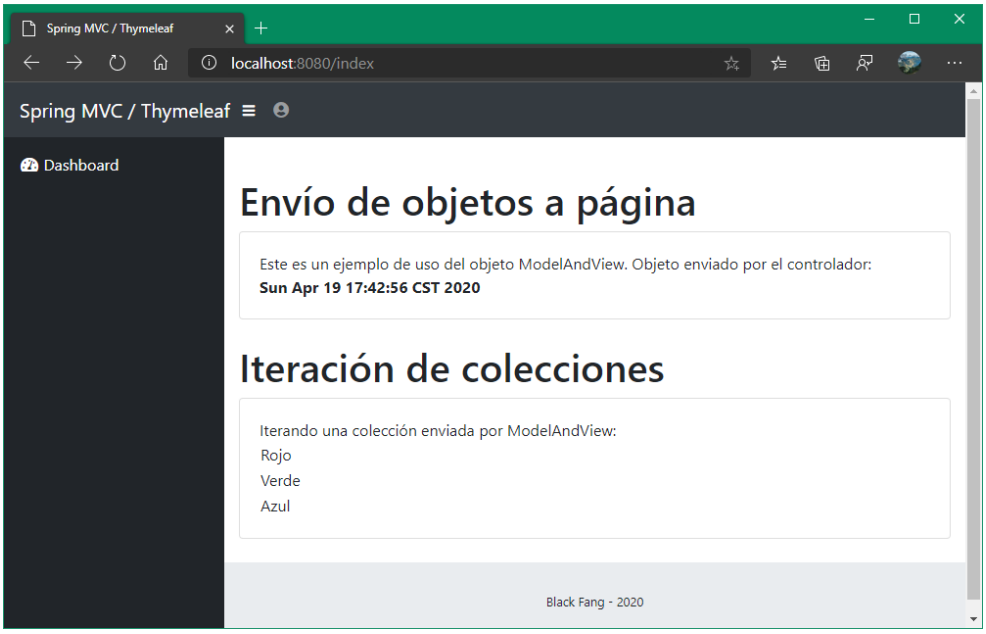
Y en nuestro **index.html** agregaremos lo siguiente:

```
69          <h1 class="mt-4">Iteraci&oacute;n de colecciones</h1>
70          <div class="card mb-4">
71              <div class="card-body">
72                  <p class="mb-0">Iterando una colecci&oacute;n enviada por ModelAndView:</p>
73                  <table>
74                      <tr th:each="c : ${colores}">
75                          <td th:text="${c}" />
76                      </tr>
77                  </table>
78              </div>
79          </div>
```

Como vemos en la línea 73 hemos creado una tabla, ahora bien, para cada elemento de la colección crearemos una fila, es por esto que al tag **tr** le agregaremos el atributo de Thymeleaf **th:each="c : ${colores}"** donde nos referimos a cada elemento de la colección llamado **c** que esté en la colección de nombre **colores** (y que viene en el ModelAndView).

Ahora pintaremos el nombre de cada color en una columna, utilizando el tag **td** y que tendrá el atributo **th:text="${c}"**, con esto, se creará la columna **td** con el valor del elemento de la colección.

Al ingresar a la URL obtendremos lo siguiente:



El proyecto Maven lo pueden encontrar en el siguiente repositorio:

https://github.com/jlozano86/modelo.git

De ahora en adelante, estaremos trabajando con este proyecto, extendiéndolo según los temas que se vayan viendo en clase.