Acceso a Datos

UT5 – ACCESO WEB CON SPRING BOOT

MVC

1. Spring MVC: Controladores

- Un controlador usualmente es el encargado de preparar el modelo (*los datos manejados por la aplicación*) y seleccionar el nombre de la vista que será utilizada para mostrar el modelo al cliente.

- El modelo es una implementación de la interface Map en la cual podemos almacenar datos en formato *clave/valor.* Estos datos serán enviados a la vista para su correcta representación, puede ser: HTML, PDF, etc., debemos decir que un controlador es capaz de generar una respuesta sin necesidad de una vista, esto es útil a la hora de crear servicios que generan respuestas en formatos como: XML, JSON, etc.

1. Spring MVC: Controladores

Típicamente definimos los controladores usando las anotaciones @Controller y @RequestMapping, además de la clase ModelAndView usada para devolver el modelo y el nombre de la vista.

CONTROLADOR BÁSICO

```
@Controller
public class MainController {

    @GetMapping({"/", "/welcome"})
    public String welcome(@RequestParam(name="nombre", required=false,
                defaultValue="Mundo") String nombre, Model model) {

        model.addAttribute("nombre", nombre);
        return "index";
    }

}
```

Ruta (o rutas) que podemos introducir en el navegador

Información que estamos enviando a la plantilla

Ruta de la plantilla
Prefijo por defecto: /src/main/resources/templates
Sufijo por defecto: .html
Ruta completa: /src/main/resources/templates/index.html

1.1. Mapeo de más de una URI

La anotación @RequestMapping y sus derivadas (@GetMapping, @PostMapping, …) pueden recibir más de una ruta como argumento. Lo hacen recibiendo varias entre { }.

```
@GetMapping({"/", "/index", "/list"})
```

De esta forma, tanto si invocamos a /, como a /index o /list, todas las llamadas se harán al mismo método.

1.2. Uso de @RequestMapping

Esta es la anotación original para mapear cualquier tipo de ruta HTTP con un método. De hecho, podríamos sustituir este código:

```
@GetMapping("/")
public String welcome(@RequestParam(name="name", required=false,
defaultValue="Mundo") String name, Model model)
```

por este otro:

```
@RequestMapping(value="/", method=RequestMethod.GET)
public String welcome(@RequestParam(name="name", required=false,
defaultValue="Mundo") String name, Model model)
```

1.2. Uso de @RequestMapping

Podemos utilizar también la anotación @RequestMapping para definir un segmento de ruta a nivel de controlador, de forma que:

```
@Controller
@RequestMapping("/app")
public class MainController {

    @GetMapping("/")
    public String welcome(@RequestParam(name="name", required=false,
    defaultValue="Mundo") String name, Model model)
    {
        model.addAttribute("nombre", name);
        return "index";
     }

}
```

La ruta para invocar el controlador sería http://localhost:9000/app/. Si añadimos más métodos de controlador a esta clase, la ruta app afectaría a todos los métodos.

1.3. Argumentos

https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/arguments.html

1.4. Tipos de retorno

https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/return-types.html

2. Spring MVC: Modelos

Model:

Comencemos con el concepto más básico: el Modelo. En pocas palabras, el Model puede proporcionar atributos utilizados para representar vistas.

Para proporcionar una vista con datos manipulables, simplemente agregamos estos datos a su objeto Modelo. Además, los Maps con atributos se pueden combinar con las instancias de Model:

2. Spring MVC: Modelos

Model:

```
@GetMapping("/showViewPage")
public String passParametersWithModel(Model model) {
      Map<String, String> map = new HashMap<>();
      map.put("spring", "mvc");
      model.addAttribute("message", "Hola mundo");
      model.mergeAttributes(map);
      return "viewPage";
        }
```

2. Spring MVC: Modelos

ModelMap:

Al igual que la interfaz de Model anterior, ModelMap también se utiliza para pasar valores para representar una vista.

La ventaja de ModelMap es que nos da la capacidad de pasar una colección de valores y tratar estos valores como si estuvieran dentro de un Map:

```
@GetMapping("/printViewPage")
 public String passParametersWithModelMap(ModelMap map) {
   map.addAttribute("welcomeMessage", "welcome");
   map.addAttribute("message", "Hola mundo");
   return "viewPage";
        }
```

2. Spring MVC: Modelos

ModelAndView:

La interfaz final para pasar valores a una vista es ModelAndView.

Esta interfaz nos permite pasar toda la información requerida por Spring MVC en una declaración:

```
@GetMapping("/goToViewPage")
 public ModelAndView passParametersWithModelAndView() {
   ModelAndView modelAndView = new ModelAndView("viewPage");
   modelAndView.addObject("message", "Hola mundo");
   return modelAndView;
       }
```

2. Spring MVC: Modelos

ModelAndView:

Todos los datos, que colocamos dentro de estos Modelos, son utilizados por una vista, en general, una vista-plantilla para representar la página web.

Si tenemos un motor de plantillas como Thymeleaf podemos pasar a la vista todos los datos almacenados en el modelo.

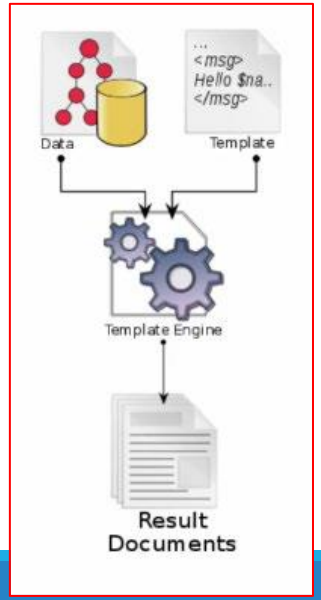
3. Spring MVC: Vista

Es la parte encargada de renderizar las plantillas que visualizará el usuario. Spring MVC está diseñado para ser independiente de la tecnología de la vista. Podemos verlo como piezas de un puzzle que encajan. Así podemos seleccionar la tecnología más adecuada.

Las vistas en Spring MVC se implementan como plantillas HTML. Se generan páginas HTML partiendo de una plantilla y de la información que viene del Controlador a través del Modelo.

3. Spring MVC: Vista

Existen diversos motores de plantillas que se pueden usar con Spring MVC: *JSP, Thymeleaf, FreeMarker, etc…*



MOTORES DE PLANTILLAS

plantilla

```
<div ...>
    <p>${nombre}</p>
    <p>${apellidos}</p>
</div>
```

modelo

nombre = Pepe

apellidos=Pérez Pérez

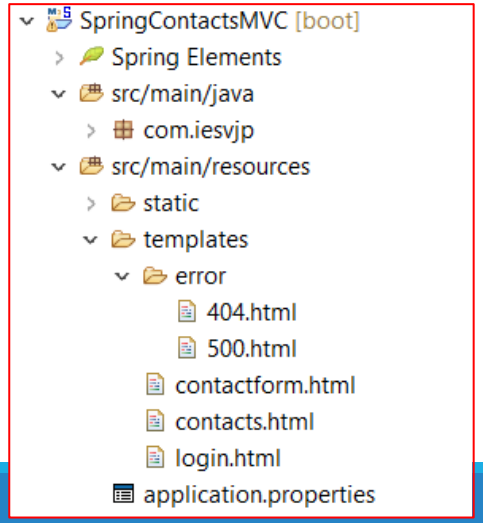
resultado

```
<div ...>
    <p>Pepe</p>
    <p>Pérez Pérez</p>
</div>
```

3. Spring MVC: Vista

Nosotros usaremos **Thymeleaf**. En el siguiente enlace podéis ver una comparativa entre JSP y Thymeleaf : https://openwebinars.net/blog/thymeleaf-vs-jsp/

Las vistas que vayamos a utilizar las guardaremos en el proyecto en la carpeta **src/main/resource/templates.**



3.1. Cargar una vista vs. redirigir

Como hemos visto, la forma más sencilla de que un controlador nos lleve a una vista es devolviendo el nombre de la plantilla a renderizar como un String:

```
@GetMapping("/")
        public String welcome(Model model) {
            model.addAttribute("mensaje", "¡Hola a todos!");
            return "index";

        }
```

3.1. Cargar una vista vs. redirigir

Sin embargo, habrá ocasiones en las que nos interese que un controlador nos lleve directamente a otro. Un escenario típico es, tras haber procesado un formulario (por ejemplo, de inserción de un nuevo registro); posiblemente, después de procesar esa petición, queramos visualizar el listado completo de registros, y así comprobar que el nuevo registro ha sido insertado.

Para poder hacer una redirección, incluimos la palabra redirect: en el valor de retorno del método, seguido de la ruta del controlador al cual queremos redirigirnos.

```
@PostMapping("/empleado/new/submit")
 public String nuevoEmpleadoSubmit(@ModelAttribute("empleadoForm") Empleado
                                                                  nuevoEmpleado) {
   servicio.add(nuevoEmpleado);
   return "redirect:/empleado/list";
       }
```

3.2. Thymeleaf

Thymeleaf se diferencia de las demás tecnologías en que las plantillas son ficheros HTML válidos que pueden verse en un navegador sin necesidad de servidor web (natural templating). Esta característica es ideal para la separación de roles: diseñadores y desarrolladores.

¿Qué es el natural templating?

3.2. Thymeleaf

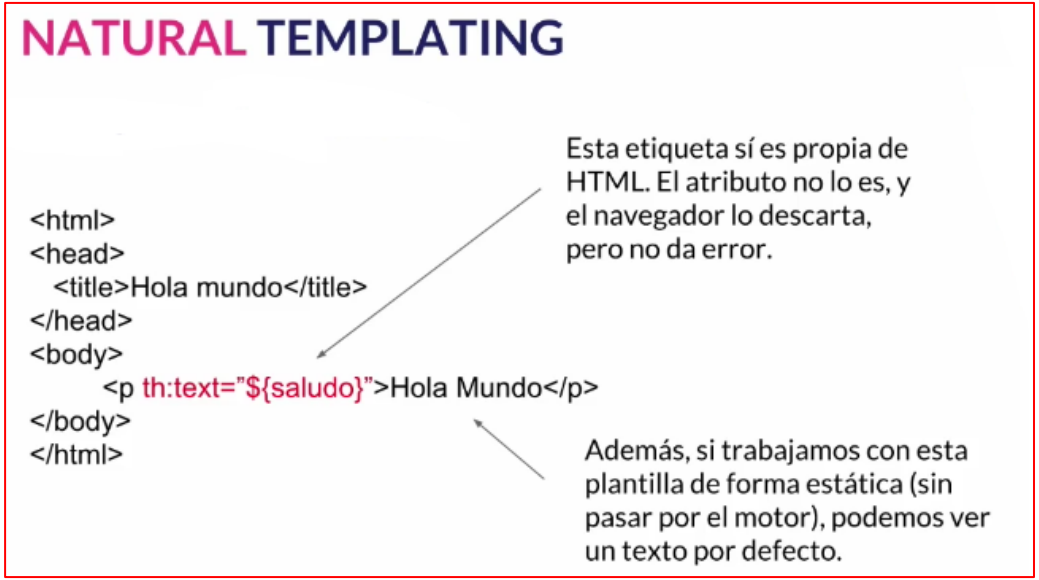
Muchos motores de plantillas que trabajan con HTML basan su funcionamiento en añadir nuevas etiquetas a las que ya define este lenguaje de marcado, para poder realizar su funcionamiento.

Por contra, Thymeleaf funciona añadiendo, a las etiquetas HTML estándar, una serie de atributos. Estos, en caso de visualizar estáticamente un documento, son descartados por el navegador. Esto aporta la ventaja de que podemos trabajar tanto el diseño web como la integración con el aplicativo sobre un mismo documento (cosa que no podríamos hacer, por ejemplo con JSP).

3.2. Thymeleaf

La sintaxis de las plantillas Thymeleaf se define en las páginas HTML mediante la etiqueta **th**

Los navegadores ignorarán el espacio de nombre que no entienden (**th**) con lo que la página seguirá siendo válida.



3.2. Thymeleaf

Para conocerlo con más detalle, vamos a dedicar un apartado entero a Thymeleaf

Dudas y preguntas