Repasemos: ¿Que es un ModelMap?

Un **ModelMap** es una implementación de `Map` diseñada específicamente para su uso en controladores de **Spring MVC**. Funciona como un contenedor de datos clave-valor que permite pasar información desde el controlador hacia la vista.

Este objeto proporciona diversos métodos que pueden resultar útiles en la manipulación de datos dentro de los controladores, tanto en **métodos GET como POST**, permitiendo inyectar información en los archivos **HTML** de la vista.

Algunos de los métodos más utilizados son:

- **`addAllAttributes(Map<String, ?> attributes)`**: Agrega todos los atributos de otro **Map** al **ModelMap**. Es útil cuando necesitas incluir múltiples atributos en una sola operación.
- **`containsAttribute(String key)`**: Verifica si el **ModelMap** contiene un atributo con una clave específica.
- **`getAttribute(String key)`**: Devuelve el valor del atributo asociado con la clave especificada.
- **`isEmpty()`**: Comprueba si el **ModelMap** está vacío, es decir, si no contiene ningún atributo.
- **`put(String key, Object value)`**: Agrega un par clave-valor al **ModelMap**. A diferencia de **`addAttribute()`**, este método **no devuelve una referencia**, por lo que se recomienda cuando solo se necesita almacenar un dato en el modelo.

¿Cuándo es útil utilizar ModelMap?

En **Spring MVC**, el uso de **ModelMap** en los métodos del controlador resulta útil en distintos escenarios, entre ellos:

- **Pasar datos a la vista**: Se emplea para transferir objetos de dominio, listas u otros datos desde el controlador hacia la vista.
- **Manejo de redirecciones y reenvíos**: Cuando un controlador realiza una **redirección** o un **reenvío**, a menudo es necesario pasar información a la vista de destino. **ModelMap** facilita esta tarea de forma eficiente.

- **Integración con tecnologías de vista**: Es un componente clave en **Spring MVC**, ya que se integra con motores de plantillas como **Thymeleaf**, **JSP** u otras tecnologías de presentación.
- **Manejo de múltiples atributos**: Permite agregar varios atributos en un solo método del controlador, lo que contribuye a mejorar la claridad del código.

Ejemplos de uso

1. Pasar mensajes de éxito o error a la vista

Supongamos que queremos mostrar mensajes de **éxito** o **error** a los usuarios. Desde el controlador, almacenamos estos mensajes en un **ModelMap**, permitiendo su uso dinámico en la vista.

Ejemplo en el controlador:

```java
@PostMapping("/registro")
public String registrarAutor(@RequestParam String nombre, ModelMap model) {
    try {
        autorServicio.crearAutor();
        model.put("exito", "¡El autor fue registrado correctamente!");
    } catch (Exception e) {
        model.put("error", e.getMessage());
        return "autor_form.html"
    }
    return "resultado";
}
```

Podemos ver, que el primer término es la llave → “exito” y el segundo término es el valor de la llave → “El Autor fue registrado correctamente!”.

En nuestro HTML, tenemos previsto un DIV que contendrá, de existir, el mensaje de error o éxito según corresponda. ¿Cómo? Planteando un condicional, que si el objeto es distinto a nulo muestre el texto.

```html
<div th:if="${exito}!=null" class="card text-white bg-success mb-3 mt-3 mensajeExito" >
    <div class="card-body">
        <h5 class="card-title">Exito 👍</h5>
        <p class="card-text" th:text="${exito}"></p>
    </div>
</div>
<div th:if="${error}!=null" class="card text-white bg-danger mb-3 mensajeError" >
    <div class="card-body">
        <h5 class="card-title">Error 👎</h5>
        <p class="card-text" th:text="${error}"></p>
    </div>
</div>
```

📌 Explicación:

- Se verifica si la variable **"exito"** o **"error"** existe en el **ModelMap**.
- Si la clave está presente, se muestra el mensaje correspondiente.

💡 ¿Por qué usamos `put()` y no `addAllAttributes()`?

En este caso, solo necesitamos almacenar un dato. No es necesario agregar múltiples atributos a la vez, por lo que **`put()`** es la opción más adecuada.

2. Pasar una colección de objetos a la vista

Si queremos enviar una lista de objetos al **HTML**, podemos almacenarlos en el **ModelMap** y recorrerlos dinámicamente en la vista con **Thymeleaf**.

Ejemplo en el controlador:

```java
@GetMapping("/lista")
public String listarAutores(ModelMap model) {
    List<Autor> autores = autorServicio.listarAutores();
    model.addAttribute("autores", autores);
    return "autor_list.html";
}
```

📌 **Explicación:**

- Se obtiene una lista de autores desde el servicio.
- La lista se almacena en el **ModelMap** con la clave **"autores"**.
- Se retorna la vista **autor_list.html**, que contendrá la información.

Uso en el HTML con Thymeleaf:

```html
<table class="table table-dark">
    <thead>
        <tr>
            <th scope="col">Id del Autor</th>
            <th scope="col">Nombre del Autor</th>


        </tr>
    </thead>
    <tbody th:each="autor : ${autores}">
        <tr>
            <th scope="row" th:text="${autor.id}"></th>
            <td th:text="${autor.nombre}"></td>

        </tr>

    </tbody>
</table>
```

📌 **Explicación:**

- La etiqueta `th:each="autor : ${autores}"` recorre la colección de autores.
- Se accede a los atributos del objeto **Autor** mediante `th:text="${autor.nombre}"` y `th:text="${autor.nacionalidad}"`.

💡 **Ventaja de este enfoque:**
Permite manejar información de manera dinámica sin modificar el HTML cada vez que cambian los datos.

El uso de **ModelMap** en **Spring MVC** es una herramienta eficaz para pasar información entre el controlador y la vista. Se recomienda en los siguientes casos:

✔ Cuando necesitas compartir mensajes de éxito o error.
✔ Para transferir listas de objetos a la vista.
✔ Al trabajar con redirecciones y reenvíos.
✔ Para mantener una estructura de código clara y eficiente.

Si se requiere manejar **un solo dato**, `put()` es suficiente. En cambio, cuando se necesita agregar múltiples atributos en un solo paso, `addAllAttributes()` es la mejor opción.