Material descargable

egg

SPRING FRAMEWORK

La **mejora continua de la experiencia del usuario** es un objetivo central en el desarrollo de plataformas digitales. Este proceso se sustenta en tres pilares fundamentales: **seguridad, usabilidad y personalización**. En este contexto, te ofrecemos material teórico que te permitirá profundizar en aspectos clave para optimizar dicha experiencia.

Nos centraremos en dos elementos esenciales que pueden marcar una diferencia significativa: la recuperación de información desde el objeto de sesión, lo que posibilita la personalización de los elementos visualizados, y la adaptación de los errores HTTP, para lograr una interacción más fluida y comprensible.

Además, exploraremos cómo características específicas de Thymeleaf, como `th:fragments` y `sec:authorize`, pueden ser aprovechadas para personalizar tus archivos HTML de forma eficiente, trabajando de la mano con el objeto `HttpSession`.

Introducción al objeto HttpSession

El objeto `HttpSession` se utiliza para gestionar el estado de la sesión del usuario en una aplicación web Java, permitiendo almacenar y recuperar información específica de cada sesión. A continuación, te presentamos algunos casos comunes de uso:

- **Autenticación de usuarios**: Después de que un usuario se autentica, se puede almacenar información relevante en la sesión, como su ID o sus roles, para verificar su acceso a diferentes partes de la aplicación.
- **Personalización de la experiencia del usuario**: Es posible almacenar preferencias del usuario, como idioma o configuraciones de tema, para personalizar la experiencia según sus gustos.
- **Almacenamiento de datos temporales**: El objeto `HttpSession` es útil para almacenar datos temporales durante la sesión del usuario, como los artículos de un carrito de compras en línea, hasta que el usuario complete la compra o cierre la sesión.

- **Mantenimiento de la sesión activa**: La sesión se mantiene activa durante la navegación del usuario, y su tiempo de expiración puede configurarse para cerrarse automáticamente después de un período de inactividad.

Implementación del objeto HttpSession

1. **Obtener el objeto HttpSession**: En un servlet de Java o un controlador de Spring MVC, puedes obtener el objeto `HttpSession` utilizando el método `request.getSession()`. Esto devolverá la sesión actual asociada con la solicitud o creará una nueva si no existe.
2. **Almacenar datos en la sesión**: Para almacenar datos en la sesión, se utilizan métodos como `setAttribute(String name, Object value)` del objeto `HttpSession`. Un ejemplo es almacenar información sobre el usuario durante un inicio de sesión exitoso.

Ejemplo de implementación:

```
   @Override
    public UserDetails loadUserByUsername(String email) throws
UsernameNotFoundException {

        Usuario usuario = usuarioRepositorio.buscarPorEmail(email);

        if (usuario != null) {
            List<GrantedAuthority> permisos = new ArrayList<>();
            GrantedAuthority p = new SimpleGrantedAuthority("ROLE_" +
usuario.getRol().toString());
            permisos.add(p);
            ServletRequestAttributes attr = (ServletRequestAttributes)
RequestContextHolder.currentRequestAttributes();
            HttpSession session = attr.getRequest().getSession(true);
            session.setAttribute("usuariosession", usuario);
            return new User(usuario.getEmail(), usuario.getPassword(), permisos);
        } else {
            return null;
        }

    }
```

3. **Recuperar datos de la sesión**: Puedes obtener datos de la sesión utilizando el método `getAttribute(String name)` de `HttpSession`.

Ejemplo de implementación

```
    @PreAuthorize("hasAnyRole('ROLE_USER', 'ROLE_ADMIN')")
    @GetMapping("/inicio")
    public String inicio(HttpSession session) {

        Usuario logueado = (Usuario) session.getAttribute("usuariosession");

        if (logueado.getRol().toString().equals("ADMIN")) {
```

```
            return "redirect:/admin/dashboard";
        }
           return "inicio.html";
    }
```

Luego en nuestro HTML usamos la información para “personalizar” la vista:

```
   <header class="header">
            <section class="contenedorMain">
                <div class="tituloBiblioteca">
                    <a class="navbar-brand" href="#">BIBLIOTECA EGG</a>
                </div>
                <div class="container contenedorLibros">
                    <p class="tituloLibros" th:if="${session.usuariosession !=
null}" th:text=" 'Hola de vuelta '+ ${session.usuariosession.nombre} "></p>
                    <a class="btn btn-xl mt-2 boton">Ver Listado</a>
                </div>
                <div class="bg-circle-1 bg-circle"></div>
                <div class="bg-circle-2 bg-circle"></div>
                <div class="bg-circle-3 bg-circle"></div>
                <div class="bg-circle-4 bg-circle"></div>
            </section>
        </header>
```

Introducción a uso de Fragments

`th:fragments` es una característica de Thymeleaf, un motor de plantillas utilizado en aplicaciones web Java. **Permite crear fragmentos reutilizables dentro de tus plantillas HTML.**

Puedes utilizar `th:fragment` en diversas etiquetas HTML como `<div>`, `<span>`, `<section>`, entre otras. La elección de la etiqueta depende del contexto y la estructura de tu HTML.

1. Definición de un fragmento: debes crear un archivo HTML que especifique la información que será utilizada como fragmento. Luego, dentro de este archivo, puedes definir el fragmento utilizando la etiqueta `<div>` u otra etiqueta HTML apropiada. No olvides incluir el atributo `th:fragment` con un nombre único para identificar el fragmento

```
<head th:fragment="head">
    <meta charset="utf-8" >
    <title>Biblioteca Egg</title>
    <link href="css/one-page-wonder.min.css" rel="stylesheet">
    <!-- CSS only -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/bootstrap.min.
```

```
css" rel="stylesheet"
integrity="sha384-0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6DOitfPri4tjfHxaWutUpFmBp4vmV
or" crossorigin="anonymous">
    <link rel="stylesheet" href="/css/index.css">
</head>
```

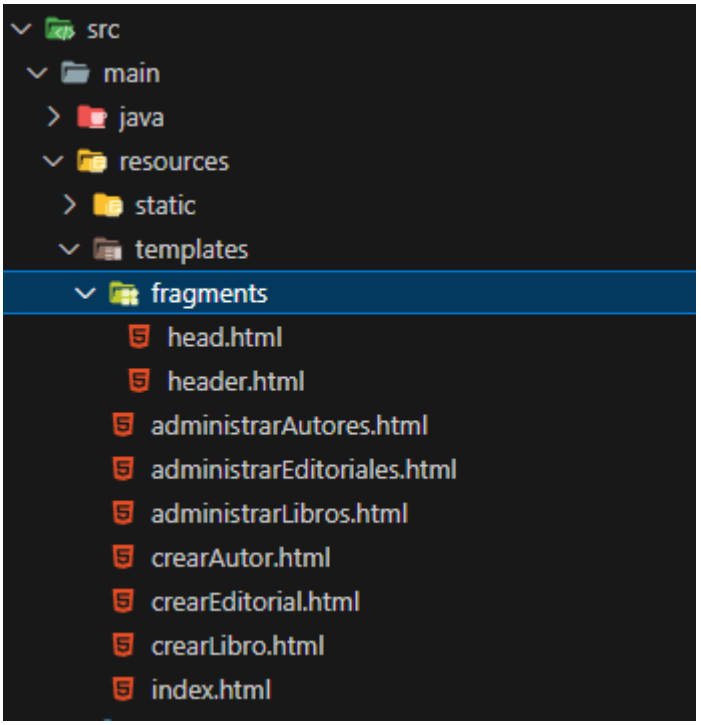
2. Uso de fragmentos en otras plantillas:

Puedes incluir fragmentos en otras plantillas con la etiqueta `th:include` o `th:replace`, referenciando el nombre del fragmento.

```
    <head th:replace="~{fragments/head :: head}"></head>
```

3. Ubicación de los archivos de fragmentos:

Para mantener una estructura organizada, es recomendable crear una carpeta "fragments" dentro de la carpeta `templates` de tu proyecto. La ruta sería: `TuProyecto/src/main/resources/templates/fragments`.



*En este ejemplo, hemos creado dos archivos HTML: uno para el `head` y otro para el `header`.

El uso de `th:fragment` en tus plantillas HTML mejora la legibilidad del código y facilita su mantenimiento. Al definir fragmentos reutilizables, aseguras que los elementos necesarios se encuentren presentes en cada plantilla. Esto favorece una

estructura coherente en tu aplicación web, lo que facilita su comprensión y mantenimiento a lo largo del tiempo.

Introducción al uso de sec:authorize

En el desarrollo web, la seguridad es esencial. `sec:authorize` de Thymeleaf es una herramienta poderosa para controlar el acceso a ciertas partes de una aplicación según los roles de los usuarios.

1. Incorporación de la dependencia:

Para usar `sec:authorize`, primero debes agregar la dependencia correspondiente en tu archivo `pom.xml`.

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity6</artifactId>
</dependency>
```

2. Declaración en la etiqueta HTML:

Es necesario declarar el uso de Thymeleaf y las extensiones de Spring Security en tu HTML.

```
<html xmlns:th="http://www.thymeleaf.org"
xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
```

3. Usos de Thymeleaf con Spring Security:

- **Autenticación de usuarios:** Puedes utilizar las etiquetas de Thymeleaf Spring Security para mostrar o ocultar contenido dependiendo del estado de autenticación del usuario. Por ejemplo, puedes mostrar un enlace para iniciar sesión si el usuario no ha iniciado sesión, o mostrar su nombre de usuario si ya está autenticado.

Ejemplo de implementación:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<head>
    <title>Inicio</title>
</head>
<body>
    <div th:if="${#authorization.expression('isAuthenticated()')}">
        <p>Bienvenido, <span th:text="${#authentication.name}"></span>!</p>
        <a href="/logout">Cerrar sesión</a>
```

```
    </div>
    <div th:unless="${#authorization.expression('isAuthenticated()')}">
        <a href="/login">Iniciar sesión</a>
    </div>
</body>
</html>
```

- **Autorización de acceso:** Thymeleaf Spring Security permite controlar el acceso a partes de tus páginas HTML en función de los roles o autoridades de los usuarios. Puedes utilizar las etiquetas para mostrar u ocultar secciones específicas del contenido según los permisos del usuario actual.

Ejemplo de implementación:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<head>
    <title>Recursos</title>
</head>
<body>
    <div sec:authorize="hasRole('ROLE_ADMIN')">
        <p>Solo visible para administradores.</p>
    </div>
    <div sec:authorize="hasAnyRole('ROLE_USER', 'ROLE_ADMIN')">
        <p>Visible para usuarios y administradores.</p>
    </div>
</body>
</html>
```

- **Seguridad en formularios:** Puedes integrar Thymeleaf Spring Security con formularios HTML para garantizar que solo los usuarios autorizados puedan enviar datos a través de ellos. Esto es útil para proteger funciones sensibles o datos confidenciales.

Ejemplo de implementación:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<head>
    <title>Formulario</title>
</head>
<body>
    <form th:action="@{/guardar}" method="post" sec:authorize="hasRole('ROLE_ADMIN')">
        <!-- Campos del formulario -->
        <button type="submit">Guardar</button>
    </form>
</body>
</html>
```

- **Mensajes de seguridad:** Thymeleaf Spring Security también te permite mostrar mensajes dinámicos relacionados con la seguridad, como mensajes

de éxito o error después de que un usuario haya realizado una acción, como iniciar sesión o cambiar su contraseña.

Ejemplo de implementación

```
</html>
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<head>
    <title>Resultado</title>
</head>
<body>
    <div th:if="${param.error}">
        <p>Error al iniciar sesión. Inténtelo de nuevo.</p>
    </div>
    <div th:if="${param.logout}">
        <p>Cierre de sesión exitoso.</p>
    </div>
</body>
</html>
```

Con esta integración, puedes gestionar de forma eficiente la seguridad y el acceso en tus aplicaciones web utilizando Thymeleaf y Spring Security.

Introducción a personalización vista errores HTTP

La personalización de la vista de errores HTTP es crucial para mejorar la experiencia del usuario en tu aplicación. En lugar de mostrar errores genéricos, puedes guiar a los usuarios con mensajes claros y útiles.

Para ello, se puede implementar una forma sencilla:

- Crear una vista html, que respete los estilos de tu proyecto.
- Crear un controlador dedicado al manejo de estos errores.

Ejemplo Completo: A continuación, te brindamos una posible implementación

Controlador de errores:

```
package com.egg.biblioteca.controladores;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.boot.web.servlet.error.ErrorController;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class ErroresControlador implements ErrorController {
// Método para manejar las solicitudes a la ruta "/error"
```

```java
    @RequestMapping(value = "/error", method = { RequestMethod.GET,
RequestMethod.POST })
    public ModelAndView renderErrorPage(HttpServletRequest httpRequest) {
        ModelAndView errorPage = new ModelAndView("error");
        String errorMsg = "";
        Integer httpErrorCode = getErrorCode(httpRequest);
        switch (httpErrorCode) {
            case 400: {
                errorMsg = "El recurso solicitado no existe.";
                break;
            }
            case 403: {
                errorMsg = "No tiene permisos para acceder al recurso.";
                break;
            }
            case 401: {
                errorMsg = "No se encuentra autorizado.";
                break;
            }
            case 404: {
                errorMsg = "El recurso solicitado no fue encontrado.";
                break;
            }
            case 500: {
                errorMsg = "Ocurrió un error interno.";
                break;
            }
            default: {
                errorMsg = "Se produjo un error inesperado.";
                break;
            }
        }
 // Agrega el código de error y el mensaje de error al modelo para mostrar en la
vista
        errorPage.addObject("codigo", httpErrorCode);
        errorPage.addObject("mensaje", errorMsg);
        return errorPage;
    }
// Método para obtener el código de estado HTTP del objeto HttpServletRequest
    private int getErrorCode(HttpServletRequest httpRequest) {
        Object statusCode =
httpRequest.getAttribute("jakarta.servlet.error.status_code");
        if (statusCode instanceof Integer) {
            return (Integer) statusCode;
        }
        // Maneja el caso en que el atributo es null o no es un Integer
        return -1; // O puedes usar otro valor por defecto adecuado
    }
 // Método que devuelve la ruta a la página de error
    public String getErrorPath() {
        return "/error.html";
    }
}
```

Vista error.html: Te proporcionamos el código de implementación que utiliza la información del **Model** dentro de la etiqueta `<header>`. Este enfoque facilita la

inserción dinámica de datos en tu plantilla HTML, asegurando que la información se muestre de manera consistente y controlada en la interfaz de usuario.

```
<header class="header">
            <section class="contenedorMain">
                <div class="tituloBiblioteca">
                    <a class="navbar-brand" th:text="'Error ' +
${codigo}">BIBLIOTECA EGG</a>
                    <p class="tituloLibros" th:text="${mensaje}"></p>

                </div>
                <section class="text-center container">
            <div class="row mt-3">
                <div class="divVolver">
                    <a th:href="@{/}" class="btn btn-secondary my-2 botonVolver
botones">Volver</a>
                </div>
            </div>
        </section>
                <div class="container contenedorLibros">
                    <p class="tituloLibros" th:if="${session.usuariosession !=
null}" th:text=" 'Hola de vuelta '+ ${session.usuariosession.nombre} "></p>

                </div>
                <div class="bg-circle-1 bg-circle"></div>
                <div class="bg-circle-2 bg-circle"></div>
                <div class="bg-circle-3 bg-circle"></div>
                <div class="bg-circle-4 bg-circle"></div>
            </section>
        </header>
```

Con estas herramientas, podrás gestionar de manera eficiente las sesiones, la seguridad, los fragmentos reutilizables y las vistas de error, optimizando la experiencia y el desarrollo de tu aplicación web.