Material descargable

egg

SPRING FRAMEWORK

Manejo y uso de imágenes en proyectos Spring

En la búsqueda continua por mejorar la experiencia del usuario, ofrecer la posibilidad de cargar imágenes para personalizar la interacción se ha convertido en un aspecto esencial en el diseño de las plataformas. La personalización de elementos visuales permite crear una conexión más cercana entre los usuarios y la plataforma, aportando una identidad visual que facilita la interacción y genera un impacto positivo en la experiencia general.

Por ejemplo, permitir que los usuarios carguen su foto de perfil no solo ofrece una forma de identificación visual, sino que también humaniza la experiencia digital, promoviendo el reconocimiento y fortaleciendo las relaciones entre los usuarios y la plataforma.

En el contexto de un proyecto Spring, existen diversas formas de almacenar imágenes en una base de datos. Las más comunes incluyen:

- **Almacenamiento directo en la base de datos como BLOB (Binary Large Object):** En este enfoque, las imágenes se convierten en un arreglo de bytes y se almacenan directamente en una columna BLOB en la base de datos. Para implementar este método, puedes usar el tipo de dato `byte[]` en tu entidad JPA, lo que permite representar las imágenes y guardarlas en la base de datos.
- **Almacenamiento de archivos en el sistema de archivos y referencia en la base de datos:** Aquí, las imágenes se almacenan en el sistema de archivos del servidor, mientras que solo se guarda la ruta o la URL de la imagen en la base de datos. Este enfoque es útil cuando deseas gestionar las imágenes de forma externa y solo necesitas almacenar las referencias en tu base de datos.
- **Almacenamiento en bases de datos de objetos binarios dedicadas:** Algunas bases de datos, como MongoDB, ofrecen soporte nativo para almacenar objetos binarios, incluyendo imágenes. En estos casos, puedes almacenar las imágenes directamente en la base de datos utilizando tipos de datos como `BinData`.
- **Almacenamiento en servicios de almacenamiento en la nube:** Una alternativa a los métodos anteriores es almacenar las imágenes en servicios

de almacenamiento en la nube, como Amazon S3, Google Cloud Storage o Azure Blob Storage. En este caso, solo se almacena la URL o la referencia a la imagen en la base de datos, lo que permite optimizar el rendimiento y reducir la carga de la base de datos local.

Cada uno de estos enfoques tiene sus propias ventajas y consideraciones en términos de rendimiento, escalabilidad, mantenimiento y costos. A la hora de diseñar un proyecto, es fundamental seleccionar el método más adecuado según los requisitos y las limitaciones del sistema.

En esta sección, nos centraremos en el enfoque de almacenamiento directo en la base de datos como BLOB, que es uno de los más utilizados en aplicaciones basadas en Spring.

1. Clase Entidad para el Manejo de Imágenes

Para gestionar imágenes en un proyecto, es fundamental crear una entidad dedicada que represente las imágenes a almacenar. Esta clase debe incluir los atributos necesarios para poder almacenar y recuperar las imágenes de manera eficiente.

Ejemplo de la creación de la clase `Imagen` como entidad:

```
package com.egg.biblioteca.entidades;
package com.egg.biblioteca.entidades;

import jakarta.persistence.Basic;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.Lob;
import org.hibernate.annotations.GenericGenerator;

@Entity
public class Imagen {
    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

    private String mime;
    private String nombre;

    @Lob
    @Basic(fetch = FetchType.LAZY)
    @Column(columnDefinition = "LONGBLOB")
    private byte[] contenido;
```

```
    public Imagen() {
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getMime() {
        return mime;
    }

    public void setMime(String mime) {
        this.mime = mime;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public byte[] getContenido() {
        return contenido;
    }

    public void setContenido(byte[] contenido) {
        this.contenido = contenido;
    }

}
```

Aspectos clave:

- **`id`:** Es un identificador único para cada imagen. Se genera automáticamente utilizando un UUID para asegurar su unicidad.
- **`mime`:** Es el tipo MIME de la imagen, como "image/jpeg" o "image/png". Este tipo de dato es importante para identificar el formato de la imagen y permitir su correcta interpretación.
- **`nombre`:** Representa el nombre de la imagen.
- **`contenido`:** Es un arreglo de bytes que almacena el contenido binario de la imagen. Se utiliza para guardar la imagen en la base de datos.

Anotaciones clave:

- **`@Lob`:** Se usa para indicar que el campo `contenido` es un objeto grande, como un BLOB (Binary Large Object). Esto es necesario cuando manejamos datos binarios de gran tamaño, como las imágenes.
- **`@Basic(fetch = FetchType.LAZY)`:** Define que el atributo `contenido` se cargará de manera perezosa (Lazy Loading), lo que significa que solo se recuperará cuando se necesite explícitamente, optimizando el rendimiento.
- **`@Column(columnDefinition = "LONGBLOB")`:** Configura la columna de la base de datos para que sea del tipo LONGBLOB, lo que garantiza que haya suficiente espacio para almacenar grandes cantidades de datos binarios.

Declaración de la relación con otras entidades

Una vez definida la entidad `Imagen`, podrás incluir un atributo de tipo `Imagen` en otras clases, como `Usuario`, para representar la imagen asociada a un usuario. Este enfoque facilita la relación entre el usuario y su foto de perfil, permitiendo almacenar y recuperar la imagen asociada de manera sencilla.

Ejemplo de cómo declarar un atributo en la clase `Usuario`:

```
@OneToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "imagen_id")
private Imagen imagen;
```

En este ejemplo, el atributo `imagen` establece una relación uno a uno con la entidad `Imagen`. La anotación `@OneToOne` indica que cada usuario tiene una sola imagen asociada, y el atributo `@JoinColumn(name = "imagen_id")` define la columna en la base de datos que se usará para asociar ambas entidades.

2. Clase Repositorio para el manejo de imágenes

Para manejar la comunicación con la base de datos en un proyecto Spring, es necesario crear un repositorio que facilite las operaciones CRUD (Crear, Leer, Actualizar, Eliminar). El repositorio en Spring se define como una interfaz que extiende `JpaRepository` o una de sus variantes (por ejemplo, `CrudRepository`), lo que le otorga las capacidades necesarias para interactuar con la base de datos.

Ejemplo de creación de la clase `ImagenRepositorio`

```
package com.egg.biblioteca.repositorios;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
```

```java
import com.egg.biblioteca.entidades.Imagen;

@Repository
public interface ImagenRepositorio extends JpaRepository<Imagen, String> {
    // Aquí puedes definir métodos personalizados si es necesario, como:
    // List<Imagen> findByNombre(String nombre);
}
```

Con este repositorio en su lugar, podrás acceder a las imágenes almacenadas en la base de datos de manera eficiente y sencilla, sin necesidad de implementar manualmente la lógica de acceso a los datos.

3. Clase Servicio para carga y visualización de imágenes

Para gestionar la lógica de negocio relacionada con las imágenes en tu aplicación, debes crear un servicio que contenga los métodos necesarios para guardar, actualizar y listar imágenes. El servicio debe interactuar con el repositorio para almacenar las imágenes y realizar operaciones según sea necesario.

Ejemplo de la clase `ImagenServicio`

```java
package com.egg.biblioteca.servicios;

import com.egg.biblioteca.entidades.Imagen;
import com.egg.biblioteca.excepciones.MiException;
import com.egg.biblioteca.repositorios.ImagenRepositorio;
import java.util.List;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.multipart.MultipartFile;

@Service
public class ImagenServicio {

    @Autowired
    private ImagenRepositorio imagenRepositorio;

    // Método para guardar una nueva imagen
    public Imagen guardar(MultipartFile archivo) throws MiException {
        if (archivo != null) {
            try {
                Imagen imagen = new Imagen();
                imagen.setMime(archivo.getContentType());
                imagen.setNombre(archivo.getName());
                imagen.setContenido(archivo.getBytes());
                return imagenRepositorio.save(imagen);
            } catch (Exception e) {
                System.err.println(e.getMessage());
            }
```

```
        }
        return null;
    }

    // Método para actualizar una imagen existente
    public Imagen actualizar(MultipartFile archivo, String idImagen) throws
MiException {
        if (archivo != null) {
            try {
                Imagen imagen = new Imagen();
                if (idImagen != null) {
                    Optional<Imagen> respuesta =
imagenRepositorio.findById(idImagen);
                    if (respuesta.isPresent()) {
                        imagen = respuesta.get();
                    }
                }
                imagen.setMime(archivo.getContentType());
                imagen.setNombre(archivo.getName());
                imagen.setContenido(archivo.getBytes());
                return imagenRepositorio.save(imagen);
            } catch (Exception e) {
                System.err.println(e.getMessage());
            }
        }
        return null;
    }

    // Método para listar todas las imágenes
    @Transactional(readOnly = true)
    public List<Imagen> listarTodos() {
        return imagenRepositorio.findAll();
    }
}
```

Explicación de los métodos:

- **`guardar(MultipartFile archivo)`**:
  - Recibe un archivo de tipo `MultipartFile` proporcionado por el usuario al cargar una imagen.
  - Valida si el archivo es nulo.
  - Crea una nueva instancia de `Imagen`, asignando sus atributos (tipo MIME, nombre y contenido).
  - Guarda la imagen en la base de datos mediante el repositorio `imagenRepositorio`.
- **`actualizar(MultipartFile archivo, String idImagen)`**:
  - Similar al método `guardar`, pero en este caso busca una imagen existente usando el `idImagen`.
  - Si la imagen existe, se actualizan los atributos y se guarda de nuevo en la base de datos.

- **`listarTodos()`**:
  - Este método permite obtener todas las imágenes almacenadas en la base de datos. Está marcado con `@Transactional(readOnly = true)`, lo que indica que este método solo realiza operaciones de lectura.

Cómo usar este servicio en otras clases:

Cuando necesites utilizar el servicio de imágenes en otras partes de tu aplicación, como en el servicio de usuario, simplemente debes inyectar `ImagenServicio` a través de la anotación `@Autowired`. Esto te permitirá acceder a sus métodos sin tener que duplicar la lógica.

```java
@Transactional
public void registrar(MultipartFile archivo, String nombre, String email, String password, String pa

    validar(nombre, email, password, password2);
    Usuario usuario = new Usuario();

    usuario.setNombre(nombre);
    usuario.setEmail(email);

    usuario.setPassword(new BCryptPasswordEncoder().encode(password));
    usuario.setRol(Rol.USER);
    Imagen imagen = imagenServicio.guardar(archivo);
    usuario.setImagen(imagen);
    usuarioRepositorio.save(usuario);
}
```

Esto garantiza que las responsabilidades estén bien separadas y que puedas manejar las imágenes de manera eficiente sin duplicar código en diferentes partes de la aplicación.

4. Clase Controlador para carga y visualización de imágenes

En la clase `ImagenControlador`, se deben realizar las modificaciones para manejar las solicitudes de carga y visualización de imágenes en la aplicación. El controlador debe ser capaz de recibir archivos de tipo `MultipartFile`, que corresponde al archivo cargado desde el cliente, y debe retornar imágenes correctamente desde la base de datos cuando se soliciten.

Ejemplo de implementación de `ImagenControlador`:

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

import com.egg.biblioteca.entidades.Usuario;
import com.egg.biblioteca.servicios.ImagenServicio;
import com.egg.biblioteca.servicios.UsuarioServicio;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("/imagen")
public class ImagenControlador {

    @Autowired
    private UsuarioServicio usuarioServicio;

    @Autowired
    private ImagenServicio imagenServicio;

    // Obtener imagen de perfil de usuario
    @GetMapping("/perfil/{id}")
    public ResponseEntity<byte[]> imagenUsuario(@PathVariable String id) {
        Usuario usuario = usuarioServicio.getOne(id);
        byte[] imagen = usuario.getImagen().getContenido();
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.IMAGE_JPEG);
        return new ResponseEntity<>(imagen, headers, HttpStatus.OK);
    }

    // Subir imagen de perfil para un usuario
    @PostMapping("/perfil/{id}")
    public ResponseEntity<String> actualizarImagenPerfil(@PathVariable String id,
@RequestParam("archivo") MultipartFile archivo) {
        try {
            // Llamada al servicio para guardar la imagen
            imagenServicio.actualizar(archivo, id);
            return new ResponseEntity<>("Imagen actualizada exitosamente",
HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>("Error al actualizar la imagen",
HttpStatus.BAD_REQUEST);
        }
    }}
```

Explicación:

- **Obtener la imagen de perfil**:

    - `@GetMapping("/perfil/{id}")`: Este método maneja las solicitudes `GET` para recuperar la imagen de perfil de un usuario. Utiliza el ID del usuario, que se pasa como parámetro de la URL (`@PathVariable`), para obtener la imagen de la base de datos.
    - `ResponseEntity<byte[]>`: La respuesta contiene los datos de la imagen en un arreglo de bytes, junto con los encabezados apropiados para definir el tipo de contenido como `image/jpeg`.
- **Actualizar la imagen de perfil**:
    - `@PostMapping("/perfil/{id}")`: Este método maneja las solicitudes `POST` para actualizar la imagen de perfil de un usuario. Recibe el archivo `MultipartFile` como parámetro mediante `@RequestParam("archivo")`, que debe coincidir con el nombre del campo en el formulario HTML.
    - Se llama al método `actualizar()` del servicio `ImagenServicio` para actualizar la imagen en la base de datos. En este caso, también se captura y maneja cualquier excepción que pueda ocurrir.

Consideraciones adicionales:

- Asegúrate de que el HTML tenga un formulario adecuado para cargar archivos con `enctype="multipart/form-data"`. Esto es necesario para que los archivos se envíen correctamente al servidor.
- El método `actualizar()` en el servicio `ImagenServicio` debe ser capaz de manejar el ID del usuario correctamente para actualizar la imagen asociada a él.

Este enfoque permite que el controlador gestione tanto la carga como la visualización de imágenes de manera eficiente y coherente con las demás capas de la aplicación.

5. HTML para carga y visualización de imágenes

Carga de imágenes: Para permitir la carga de imágenes desde un formulario HTML, se debe incluir un campo de entrada de tipo `file` dentro del formulario. Esto permitirá al usuario seleccionar el archivo desde su dispositivo. Además, debes asegurarte de incluir el atributo `enctype="multipart/form-data"` en el formulario para poder enviar archivos binarios al servidor, como una imagen.

Ejemplo de formulario HTML para cargar imágenes:

```
<form class="formulario" th:action="@{/registro}" method="POST"
enctype="multipart/form-data">
    <div class="form-group my-3">
        <input type="file" class="form-control" name="archivo">
    </div>
    <!-- Otros campos del formulario -->
</form>
```

Este formulario permite enviar un archivo, como una imagen, al servidor para su procesamiento y almacenamiento.

Visualización de imágenes: Cuando necesites mostrar la imagen de un perfil o cualquier otro tipo de imagen almacenada, puedes usar el método del controlador que recupera la imagen de la base de datos, transformándola en un array de bytes que luego es mostrado en la página. El atributo `th:src` se utiliza para indicar la URL de la imagen, construyendo dinámicamente la ruta basada en el ID del usuario.

Ejemplo de código HTML para mostrar una imagen de perfil:

```
<div class="container">
    <a th:if="${session.usuariosession.imagen != null}" class="d-flex
align-items-center">
        <img class="logo"
th:src="@{/imagen/perfil/__${session.usuariosession.id}__}" >
    </a>
 </div>
```

- **`th:if="${session.usuariosession.imagen != null}"`**: Verifica si el usuario tiene una imagen de perfil almacenada. Si es así, se renderiza el bloque HTML.
- **`th:src="@{/imagen/perfil/__${session.usuariosession.id}__}"`**: Crea dinámicamente la URL para acceder a la imagen de perfil usando el ID del usuario.

Esta estructura facilita la carga y la visualización de imágenes en tu aplicación, permitiendo tanto la actualización como la visualización de imágenes de perfil de los usuarios de forma sencilla.

Te animo a seguir explorando y perfeccionando la implementación de imágenes en tus proyectos. Considera nuevas formas de optimizar la carga, almacenamiento y visualización de imágenes en tus aplicaciones. Para profundizar más, puedes consultar recursos confiables como la **documentación oficial de Spring Framework**, que es muy útil para conocer las mejores prácticas y configuraciones para manejar archivos binarios, o explorar otros sitios especializados en desarrollo web. La exploración continua y el aprendizaje constante son esenciales para mejorar tus habilidades como desarrollador y seguir creciendo en la programación.