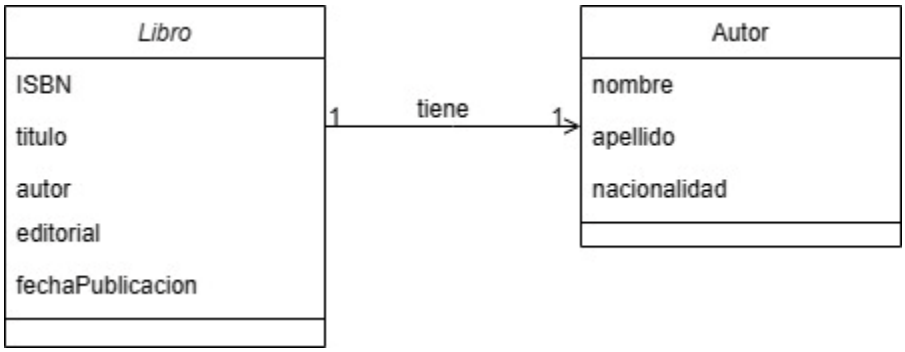
Este documento tiene el objetivo de explicar mediante un ejemplo cómo funciona el patrón MVC. Para esto vamos a realizar el alta y modificación de un libro, y que tiene como atributo un objeto de tipo Autor. También vamos a ver paso a paso como cargar y obtener los datos de un menú de opciones*(<select></select>)*.

Además, mostramos un ejemplo de cómo guardar y modificar un atributo imagen de un objeto.

Libro y Autor

Para esta explicación vamos a tomar como ejemplo la relación entre las clases Libro y Autor.



Como se puede ver esta es una relación uno a uno, donde un libro tiene un autor. Vamos a trabajar de forma harcodeada el alta y modificación de un libro, que tenga la opción de seleccionar de una lista de autores y guardar, al libro, en una lista de libros. Para esto, lo que tendríamos que tener definido son las clases Autor y Libro en el paquete model.

```java
package ar.edu.unju.fi.model;

import org.springframework.stereotype.Component;

@Component
public class Autor {

    private int id;
    private String nombre;
    private String apellido;
    private String nacionalidad;

```

```java
@Component
public class Libro {

    private int id;
    private int isbn;
    private String titulo;
    private String editorial;

    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate anioPublicacion;

    @Autowired
    private Autor autor;
```

Como se puede ver en la segunda imagen la clase Libro tiene como atributo autor del tipo Autor, al cual le hemos agregado la anotación *@Autowired*, esta anotación inyecta una dependencia con la clase Autor por lo que no es necesario instanciarla.

También deberíamos tener armada una vista con un formulario para la carga de datos de un libro. Como se ha visto en clases, al trabajar con la plantilla Thymeleaf lo que vamos a hacer es relacionar los campos de los formularios mediante sus atributos para que se logre una comunicación y conexión con el controlador.

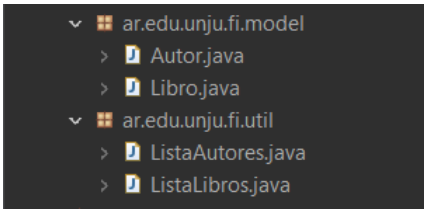
```html
<form th:action="@{/libros/guardar}"
        th:object="${unLibro}"
        method="post"
        enctype="multipart/form-data">
        <div class="mb-3">
                <input type="hidden" class="form-control" id="id" th:field="*{id}" >
        </div>
        <div class="mb-3">
                <label for="titulo" class="form-label">Titulo</label>
                <input type="text" class="form-control" id="titulo" th:field="*{titulo}"
                aria-describedby="Ingrese titulo del libro">
        </div>
        <div class="mb-3">
                <label for="isbn" class="form-label">ISBN</label>
                <input type="text" class="form-control" id="isbn" th:field="*{isbn}"
                aria-describedby="Ingrese ISBN del libro">
        </div>
        <div class="mb-3">
                <label for="autor" class="form-label">Autor</label>
                <select class="form-select" id="autor" th:field="*{autor.id}">
                        <option th:each="a: ${autores}"
                        th:value="${a.id}"
                        th:text="${a.nombre}+'-'+${a.apellido}"></option>
                </select>
        </div>
        <div class="mb-3">
                <label for="editorial" class="form-label">Editorial</label>
                <input type="text" class="form-control" id="editorial"
                th:field="*{editorial}">
        </div>
        <div class="mb-3">
                <label for="anioPublicacion" class="form-label">Año de Publicación</label>
                <input type="date" class="form-control" id="anioPublicacion"
                th:field="*{anioPublicacion}">
        </div>
        <div class="mb-3 d-flex justify-content-center">
                <button type="submit" class="btn btn-success">GUARDAR</button>
        </div>
</form>
```

Y además una página denominada “listaLibros.html” para mostrar una tabla con todos los libros guardados.

```
<table class="table table-responsive table-striped bg-white m-0">
        <thead >
           <tr class="text-center">
           <th scope="col">ID</th>
           <th scope="col">ISBN</th>
           <th scope="col">TITULO</th>
           <th scope="col">AUTOR</th>
           <th scope="col">EDITORIAL</th>
           <th scope="col">AÑO DE PUBLICACIÓN</th>
           <th scope="col"> ACCIONES</th>
         </tr>
        </thead>
        <tbody class="table-group-divider">
           <tr class="text-center" th:each="l : ${libros}">
           <td th:text="${l.id}"></td>
           <td th:text="${l.isbn}"></td>
           <td th:text="${l.titulo}"></td>
           <td th:text="${l.autor.nombre}+' '+${l.autor.apellido} "></td>
           <td th:text="${l.editorial}"></td>
           <td th:text="${l.anioPublicacion}"></td>
           <td ><a class="btn btn-warning" th:href="@{/libros/modificar/}+${l.id}"
                >MODIFICAR</a>
                <a class="btn btn-danger" href="#">ELIMINAR</a>
           </td>
         </tr>
        </tbody>
</table>
```

LISTA DE AUTORES Y LISTA DE LIBROS

Lo primero que necesitamos es una lista que nos permita guardar objetos del tipo Autor y otra que nos permita guardar Libros. Estas listas provisorias funcionarán como nuestra persistencia de datos por el momento así que crearemos un paquete denominado útil, y dentro de ella creamos una clase denominada ListaAutores y otra ListaLibros.



En la clase ListaAutores vamos a tener precargada una lista de autores para poder trabajar con el alta de Libros. Creamos una lista del tipo Autor, tanto la definición de la lista como los métodos están definidos con la palabra reservada *static* para no instanciar esta clase al momento de utilizarla.

```java
package ar.edu.unju.fi.util;

import java.util.ArrayList;

public class ListaAutores {
    public static List<Autor> lista= new ArrayList<Autor>();

    public static List<Autor> getListaAutores() {
        if(lista.isEmpty()) {
            lista.add(new Autor(1,"Ana","Frank","Alemana"));
            lista.add(new Autor(2,"Jose Mauro","de Vasconcelos","Brasileña"));
            lista.add(new Autor(3,"Edgar Allan","Poe","Estadounidense"));
        }

        return lista;
    }

    public static Autor findAutorById(int id) {
        Autor autor= new Autor();
        for( Autor a: lista) {
            if (id==a.getId()) {
                autor=a;
                break;
            }
        }
        return autor;
    }

    public static void setAutor( Autor autor) {
        autor.setId(lista.get(lista.size()-1).getId()+1);
        lista.add(autor);
    }
}
```

El método *getListaAutores()* nos devuelve la lista de los autores precargados, y únicamente se cargará con los datos precargados si ésta se encuentra vacía (esta lógica puede parecer errónea pero solo lo manejaremos así para este ejemplo).

Luego tenemos un método denominado *findAutorById()* que recibe como parámetro un id. Este método funciona como una búsqueda ya que de acuerdo al id enviado por parámetro buscará en la lista de autores y me devolverá el autor que tenga este id.

Por último se puede ver el método setAutor() que recibe por parámetro un objeto del tipo Autor. Que, como se puede observar, antes de guardar al autor, obtiene de la lista de autores, el ultimo valor de id guardado para luego sumarle un valor y finalmente agregarlo a la lista. Este método simula lo que sería la persistencia de datos y nos introduce en lo que realmente veremos al momento de trabajar con la capa de repositorio y servicio.

CONTROLADOR Y VISTA

Ahora vamos a realizar el controlador que recibirá y resolverá las peticiones realizadas por el cliente.

```java
@Controller
@RequestMapping("/libros")
public class LibroController {

	@Autowired
	private Libro unLibro;

	@Autowired
	private Autor unAutor;

	@GetMapping("/lista")
	public ModelAndView getPageLibros() {
		ModelAndView mav= new ModelAndView("listaLibros");
		mav.addObject("libros", ListaLibros.getListLibros());
		return mav;
	}

	@GetMapping("/formulario")
	public ModelAndView getPageFormLibro() {
		unLibro= new Libro();
		ModelAndView mav= new ModelAndView("nuevoLibro");
		mav.addObject("unLibro", unLibro);
		mav.addObject("autores",ListaAutores.getListaAutores());
		return mav;
	}

	@PostMapping("/guardar")
	public ModelAndView postPageSaveLibro(@ModelAttribute("unLibro") Libro libro)  {
		unAutor=ListaAutores.findAutorById(libro.getAutor().getId());
		libro.setAutor(unAutor);
		if(libro.getId()>0)
			ListaLibros.libros.set(libro.getId()-1, libro);
		else
			ListaLibros.addLibro(libro);

	    ModelAndView mav= new ModelAndView("listaLibros");
	    mav.addObject("libros", ListaLibros.getListLibros());
		return mav;
	}

	@GetMapping("/modificar/{id}")
	public ModelAndView getPageEditLibro(@PathVariable("id") int id) {
		unLibro= ListaLibros.findLibroById(id);
		ModelAndView mav= new ModelAndView("nuevoLibro");
		mav.addObject("unLibro", unLibro);
		mav.addObject("autores",ListaAutores.getListaAutores());
		return mav;
	}

}
```

Agregamos las anotaciones correspondientes para que esta clase funcione como un controlador *(@Controller)* y la anotación *@RequestMapping("/libros")* que recibirá todas las solicitudes de la vista que comiencen con la cadena *"/libros"* y que redireccionará al método especifico que se solicita. Por ejemplo, si mi solicitud es */libros/formulario,* esta cadena nos dice que se ejecute las sentencias del método que tenga esta dirección, que para nuestro ejemplo sería el primer método declarado.

Inyectamos dependencias de objetos de la clase Libro y Autor con la anotación *@Autowired.*

@GetMapping(“/nombredeladirección”)

Método *getPageFormLibro()*

```
27    @GetMapping("/formulario")
28    public ModelAndView getPageFormLibro() {
29        ModelAndView mav= new ModelAndView("nuevoLibro");
30        mav.addObject("unLibro", unLibro);
31        mav.addObject("autores",ListaAutores.getListaAutores());
32        return mav;
33    }
```

Este método se encarga de mostrar el formulario para la carga de un libro. Como es una petición donde se solicita la carga de la vista de un formulario y no enviamos ningún dato (para este ejemplo), lo correcto es realizar peticiones de tipo **GET**. Ya que esta petición por lo general se utiliza para solicitar recursos, por ejemplo, solicitamos una simple página html, una tabla con datos, una búsqueda de un ente en específico etc. Es por esto que utilizamos la anotación *@GetMapping* para que las peticiones con el argumento */libros/formulario* sean ejecutados en este método.

Como se puede ver este método devuelve un objeto del tipo *ModelAndView*, por lo que si observamos en la línea 29 se define un objeto denominado mav del tipo *ModelAndView* y que se envía mediante su constructor el nombre de la página que se va a renderizar, en este ejemplo es el formulario para un nuevo libro.

Además de la página “nuevoLibro.html” vamos cargar la lista de autores para que se pueda seleccionar mediante la etiqueta *<select></select>* como se vio al comienzo.

```
17          <form th:action="@{/libros/guardar}"
18            th:object="${unLibro}"
19            method="post">
20            <div class="mb-3">
21              <input type="hidden" class="form-control" id="id"
22                     th:field="*{id}" >
23            </div>
24            <div class="mb-3">...
29            <div class="mb-3">...
34            <div class="mb-3">
35              <label for="autor" class="form-label">Autor</label>
36              <select class="form-select" id="autor" th:field="*{autor.id}">
37                  <option th:each="a: ${autores}"
38                          th:value="${a.id}"
39                          th:text="${a.nombre}+'-'+${a.apellido}"></option>
40              </select>
41            </div>
```

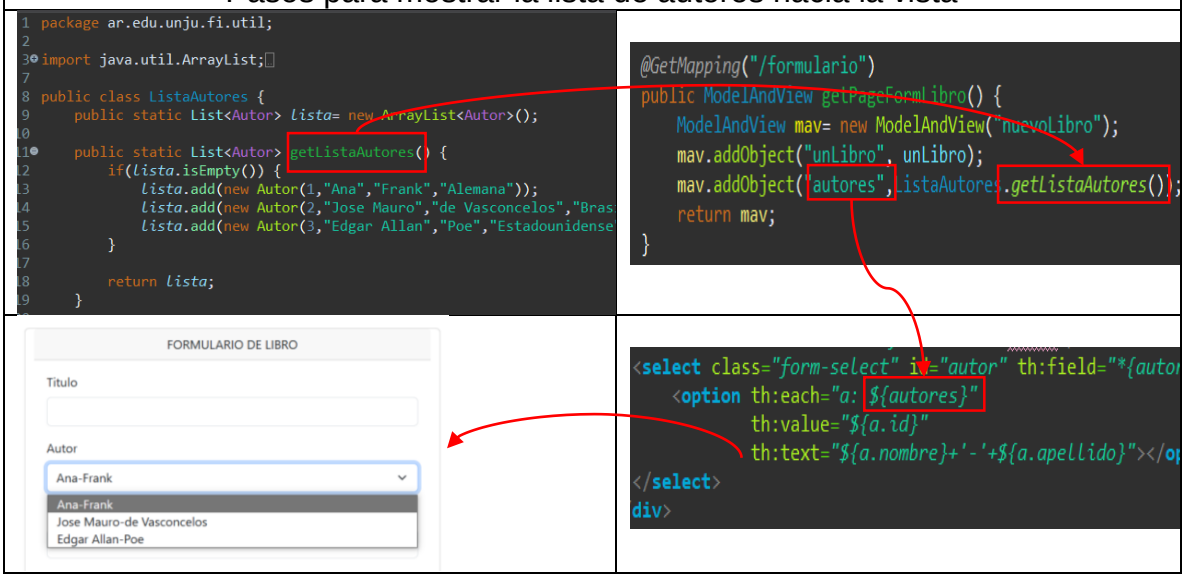
CARGA DE LA LISTA AUTORES AL MENU DE OPCIONES (<select></select>)

Con el atributo *th:each*, que nos brinda la plantilla thymeleaf, podemos cargar todos objetos que tiene la lista. En la etiqueta *<option></option>* traduciríamos la sentencia *th:each=”a: ${autores}”* como sabemos hacer el recorrido de una lista en java(foreach), para cada objeto denominado “a” de la lista “autores” se creará una porción de código con la etiqueta *<option></option>*. En donde *th:value* tendrá el valor que se corresponde con el atributo *th:field* que se encuentra en la etiqueta

<select></select>. Lo que quiere decir es que de acuerdo a la opción que seleccionemos, el valor *th:value* que se encuentra seleccionado será el valor que tomará *th:field.*

th:text solo nos permite colocar el valor que se mostrará en el menú desplegable *<select>.*

Pasos para mostrar la lista de autores hacia la vista



BOTON GUARDAR

```
<form th:action="@{/libros/guardar}"
  th:object="${unLibro}"
  method="post">
  <button type="submit" class="btn btn-primary">AGREGAR</button>
</form>
```

Cuando ya cargamos los datos y hacemos clic en guardar, se va a ejecutar la acción del formulario mediante el atributo *th:action*, que para nuestro ejemplo hace una petición a la url "*/libros/guardar*". Este tipo de petición es de tipo **POST** como se puede ver en la atributo *method* del formulario y además lleva los datos cargados en el objeto definido como ${*unLibro}* en el atributo *th:object*. Ahora nos dirigimos al controlador para saber qué es lo que pasa por ahí.

@PostMapping("/nombredeladirección")

Método *postPageSaveLibro()*

```java
@PostMapping("/guardar")
public ModelAndView postPageSaveLibro(@ModelAttribute("unLibro") Libro libro)  {
    unAutor=ListaAutores.findAutorById(libro.getAutor().getId());
    libro.setAutor(unAutor);
    if(libro.getId()>0)
        ListaLibros.libros.set(libro.getId()-1, libro);
    else
        ListaLibros.addLibro(libro);

    ModelAndView mav= new ModelAndView("listaLibros");
    mav.addObject("libros", ListaLibros.getListLibros());
    return mav;
}
```

Como vimos anteriormente al hacer clic en guardar se hace esta petición de tipo POST. Este tipo de peticiones se hacen cuando tenemos que enviar datos, principalmente datos grandes, o datos de formularios. Ya que por lo general los datos que se envían mediante este tipo de petición se realizan de forma oculta y no en la URL como en las peticiones de tipo GET.

Volviendo ahora con nuestro ejemplo este método se encargará de recibir los datos y guardarlos en la lista de libros que se definió en un comienzo. Este método recibe las peticiones tanto para agregar como modificar un libro.

Lo primero que vemos es que este método también devuelve un objeto del tipo *modelAndView*, y además recibe como parámetro un objeto del tipo libro, similar a los métodos que definimos comúnmente, con la particularidad que se utiliza una anotación llamada *@ModelAttribute*. Esta anotación lo que hace es recibir o atrapar el atributo libro del modelo o vista que fue enviado por el formulario. A su vez este valor lo guardamos en un objeto de tipo Libro con nombre libro para poder trabajar con él.

Con los datos obtenidos de la vista lo ideal sería agregar el objeto libro a la lista libros, pero vamos a retroceder un poco para explicar que sucede con el atributo autor del objeto Libro.

```
<form th:action="@{/libros/guardar}"
  th:object="${unLibro}"
  method="post">
  <div class="mb-3">
    <label for="titulo" class="form-label">Titulo</label>
    <input type="text" class="form-control" id="titulo"
           th:field="*{titulo}" aria-describedby="Ingrese titulo del libro">
  </div>
  <div class="mb-3">
    <label for="autor" class="form-label">Autor</label>
    <select class="form-select" id="autor" th:field="*{autor.id}">
        <option th:each="a: ${autores}"
                th:value="${a.id}"
                th:text="${a.nombre}+'-'+${a.apellido}"></option>
    </select>
  </div>
```

Cuando realizamos el enlace de cada atributo del objeto libro para cada campo del formulario con *th:field*, hay un detalle que observar para el atributo autor. Debido a que si *th:value* toma el valor del objeto autor y *th:field* toma el valor de autor, al momento de castear el objeto no puede castear su atributo autor. Por lo que

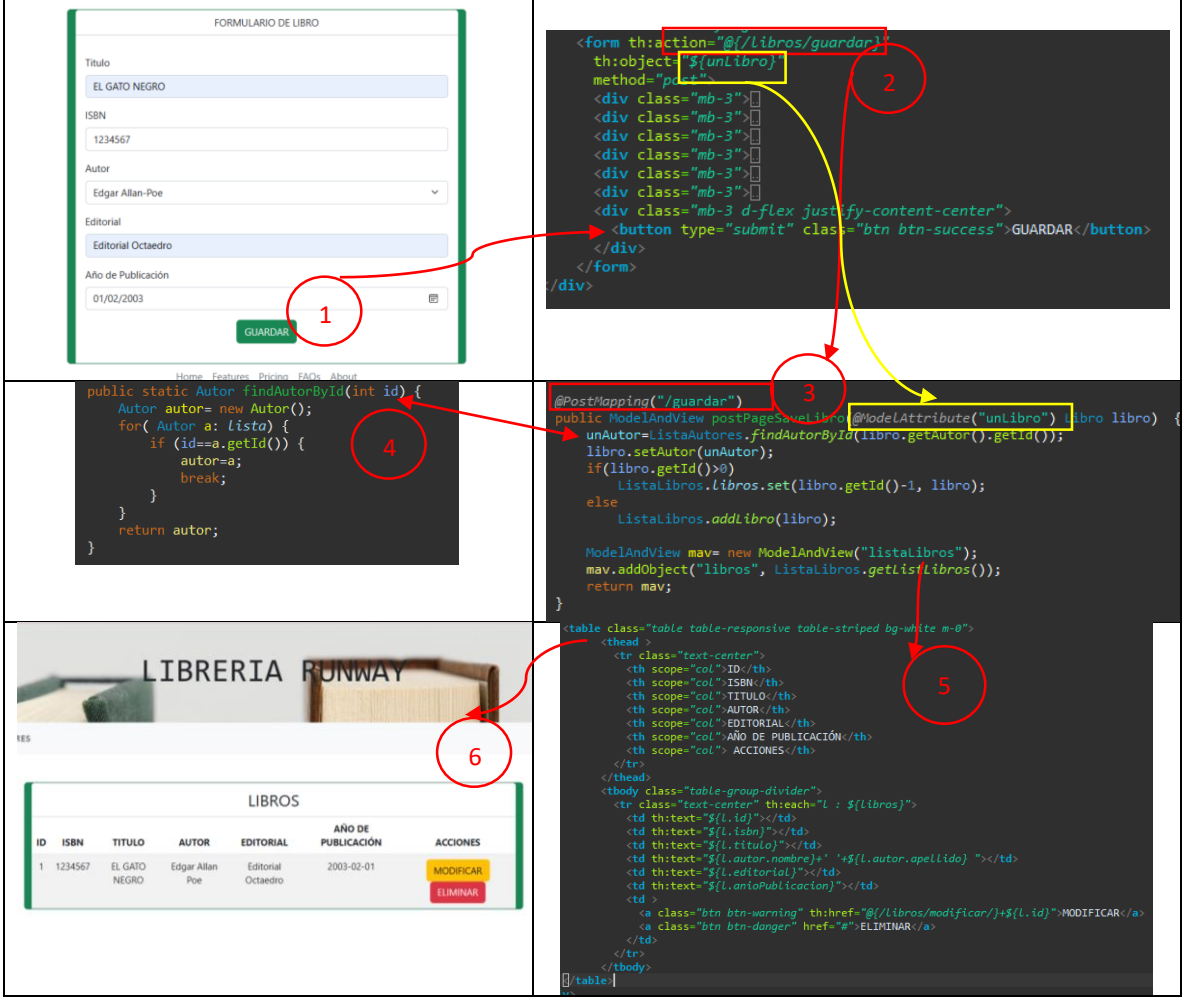
realizamos un pequeño cambio para poder guardar este atributo. En lugar de guardar el objeto autor, solo necesitaremos su id, ya que este es un dato único e irrepetible.

Entonces en lugar de guardar el atributo autor vamos a guardar en *th:field* y en *th:value* el id, como se puede ver en la imagen.

Volviendo al controlador, antes de guardar el objeto en la lista, vamos a buscar el autor de la lista de autores enviando su id invocando al método *findAutorById(idDelAutor)* definido y explicado anteriormente.

Luego vamos a realizar una condición para determinar si la petición se encarga de un alta o modificación de un libro. Es por esto que si el atributo id de nuestro objeto libro tiene un valor mayor a 0, estamos tratando una modificación así que agrega el nuevo objeto libro en la posición actual determinada por el id. Sino agrega el objeto autor al libro. Y adicionalmente se direccionará a la página html "*listaLibros*", por lo que se define un objeto del tipo *ModelAndView* cargando en su constructor la página a renderizar, y además enviando la lista de Libros ya definida.

Pasos desde la petición de la vista(Alta de un libro) hasta la muestra de los datos ya cargados



Metodo *getPageEditLibro()*

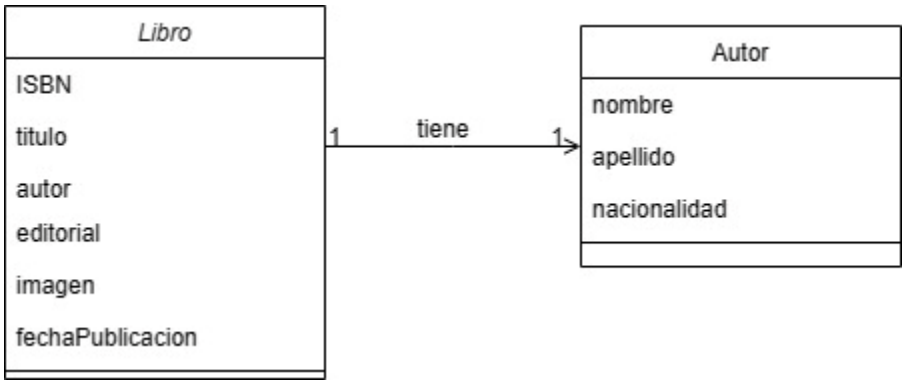
```
@GetMapping("/modificar/{id}")
public ModelAndView getPageEditLibro(@PathVariable("id") int id) {
    unLibro= ListaLibros.findLibroById(id);
    ModelAndView mav= new ModelAndView("nuevoLibro");
    mav.addObject("unLibro", unLibro);
    mav.addObject("autores",ListaAutores.getListaAutores());
    return mav;
}
```

Luego de realizar el alta y rendenrizar la vista de la lista de libros,(como vimos en el cuadro anterior), tenemos la opción de modificarlo. Al hacer clic en el botón "*modificar*", este hace una petición que toma el método *getPageEditLibro().* Este tipo de petición envía mediante la url el dato id del libro que queremos modificar, y atrapamos dicho dato con la anotación @PathVariable, ya que esta anotación nos permite saber cuál es la variable que se encuentra en dicha url.

El método se encarga de buscar y guardar en una variable *unLibro* el libro que se busca en la lista mediante su id. Se crea un objeto de tipo ModelAndView y se envía por el constructor la página html "nuevoLibro" que se va a renderizar. Además, se agrega a la vista el libro encontrado y la lista de autores que se necesita para cargar el menú de opciones que se encuentra en dicha página.

ALTA Y MODIFICACIÓN CON ATRIBUTO IMAGEN

CLASES LIBRO-AUTOR

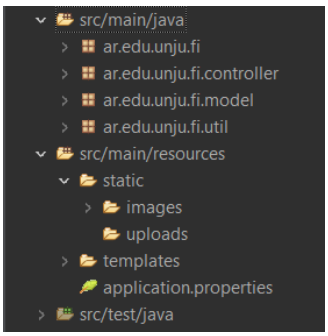


Con este ejemplo pretendemos mostrar una forma de persistir una imagen, aunque por el momento seguimos trabajando con listas.

Continuando con el ejemplo anterior Libro-Autor lo primero que haremos será agregar un atributo de tipo String para guardar el nombre de la imagen (portada) del libro.

```java
package ar.edu.unju.fi.model;

import java.time.LocalDate;

@Component
public class Libro {

    private int id;
    private int isbn;
    private String titulo;
    private String editorial;

    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private LocalDate anioPublicacion;

    private String imagen;

    @Autowired
    private Autor autor;
```

Ahora creamos una carpeta denominada *uploads* en nuestro proyecto. En esta carpeta vamos a guardar todas las imágenes de los objetos libro.



Luego en el paquete útil creamos una clase a la que vamos a llamar “UploadFile”. En esta clase lo que haremos será realizar los métodos necesarios para poder generar la ruta de la imagen, copiar y eliminar la imagen de una carpeta.

```
public class UploadFile {

        private final static String UPLOADS_FOLDER = "src/main/resources/static/uploads";

        public Resource load(String filename) throws MalformedURLException {
                Path path = getPath(filename);
                Resource resource = new UrlResource(path.toUri());
                if (!resource.exists() || !resource.isReadable()) {
                        throw new RuntimeException("Error in path: " + path.toString());
                }
                return resource;
        }

        public Path getPath(String filename) {
                return Paths.get(UPLOADS_FOLDER).resolve(filename).toAbsolutePath();
        }

        public String copy(MultipartFile file) throws IOException {
                String uniqueFilename = UUID.randomUUID().toString() + "_" +
                file.getOriginalFilename();
                Path rootPath = getPath(uniqueFilename);
                Files.copy(file.getInputStream(), rootPath);

                return uniqueFilename;
        }

        public boolean delete(String filename) {
                Path rootPath = getPath(filename);
                File file = rootPath.toFile();
                if(file.exists() && file.canRead()) {
                        if(file.delete()) {
                                return true;
                        }
                }
                return false;
        }

}
```

`private final static String UPLOADS_FOLDER = "src/main/resources/static/uploads";`	Creamos una constante denominada *UPLOADS_FOLDER*. Esta contendrá la ruta relativa a la carpeta uploads que creamos en nuestro proyecto.

`public Path getPath(String filename) {`
`  return Paths.get(UPLOADS_FOLDER).resolve(filename).toAbsolutePath();`
`}`
	Método *getPath()*: recibe como parámetro el nombre de la imagen y mediante la interfaz *Paths* devuelve la ruta absoluta de acuerdo a la combinación de la ruta relativa que se encuentra en la constante *UPLOADS_FOLDER* y el nombre de la imagen.

`public Resource load(String filename) throws MalformedURLException {`
`  Path path = getPath(filename);`
`  Resource resource = new UrlResource(path.toUri());`
`  if (!resource.exists() || !resource.isReadable()) {`
`  throw new RuntimeException("Error in path: " + path.toString());`
`  }`
`  return resource;`
`}`
	Método *load():* recibe como parámetro el nombre de la imagen y crea la ruta absoluta de la imagen llamando al método *getPath()*.
	En caso de que la ruta no exista o no se pueda leer se atrapa el error mediante una excepción.

`public String copy(MultipartFile file) throws IOException {`
`  String uniqueFilename = UUID.randomUUID().toString() + "_" + file.getOriginalFilename();`
`  Path rootPath = getPath(uniqueFilename);`
`  Files.copy(file.getInputStream(), rootPath);`
`  return uniqueFilename;`
`}`
	Metodo *copy():* Recibe como parámetro un objeto de tipo MultipartFile(clase que carga archivos en forma de formulario) y se encarga de copiar la imagen a la carpeta que contendrá todas las imágenes. En la variable *uniqueFilename* guarda el nombre de la imagen anidado a una cadena generada de forma random(UUID). Vuelve a crear una ruta absoluta con el nuevo nombre de la imagen y copia la imagen obtenida en el *file.*

`public boolean delete(String filename) {`
`  Path rootPath = getPath(filename);`
`  File file = rootPath.toFile();`
`  if(file.exists() && file.canRead()) {`
`  if(file.delete()) {`
`  return true;`
`  }`
`  }`
`  return false;`
`}`
	Metodo *delete():* Recibe como parámetro el nombre de la imagen, obtiene la ruta absoluta con el método *getPath()* y se capta en la variable *rootPAth*. Luego mediante la interfaz File crea un archivo con la ruta. Si el archivo existe se elimina.

Realizamos algunas modificaciones en la vista y el controlador de la siguiente forma.

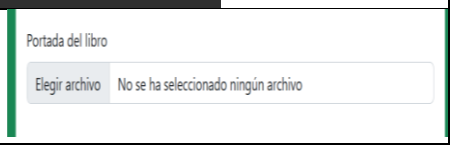
Página “nuevoLibro.html”

Para la modificación del alta de un libro vamos al formulario de la página “*nuevoLibro.html*” y agregamos:

- Un atributo en la etiqueta *form* denominada ***enctype= “multipart/form-data”.*** Este atributo aplica los protocolos necesarios para que el formulario pueda enviar archivos y no únicamente texto.
- Un campo para cargar la imagen. La etiqueta *input* será de tipo file y el atributo *name* es importante ya que mediante el vamos a captar el archivo enviado. El atributo th:text nos ayudará a mostrar el nombre de la imagen.

```
<form th:action="@{/libros/guardar}"
  th:object="${unLibro}"
  method="post"
  enctype="multipart/form-data">
  <div class="mb-3">
```

```
<div class="mb-3">
<label for="file" class=form-label>Portada del libro</label>
<input type="file" name="file" id="file" class="form-control"
      th:text="${unLibro.imagen}"/> <br>
</div>
```



Con estas modificaciones pasamos al controlador.

Controlador “LibroController”

- Inyectamos un objeto de tipo UploadFile.
- El método *postPageSaveLibro()* sufre algunas modificaciones para poder determinar si se guarda se actualiza o se elimina una imagen.

- Agregamos un método denominado *getPagePortadaLibro().* Este método se encarga de ejecutar la petición de mostrar la imagen de un libro seleccionado de la tabla.
- Agregamos un método denominado *goImage().* Este método se encarga de cargar la ruta de la imagen en un archivo de tipo Resource, y por medio de la clase ResponseEntity podremos enviar el archivo para poder mostrar la imagen en la vista.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.ModelAndView;
```

```
@Autowired
private UploadFile uploadFile;

@PostMapping("/guardar")
public ModelAndView postPageSaveLibro(@ModelAttribute("unLibro") Libro libro,
@RequestParam("file") MultipartFile image) throws Exception {
        unAutor=ListaAutores.findAutorById(libro.getAutor().getId());
        libro.setAutor(unAutor);
        if(libro.getId()>0) {
                unLibro= ListaLibros.findLibroById(libro.getId());
                if (!image.isEmpty()) {
                        if(unLibro.getImagen() != null && unLibro.getImagen().length() >0)
                                uploadFile.delete(unLibro.getImagen());
                        String uniqueFileName = uploadFile.copy(image);
                        libro.setImagen(uniqueFileName);
                }else {
                        if(unLibro.getImagen() != null)
                                libro.setImagen(unLibro.getImagen());
                }
                ListaLibros.libros.set(libro.getId()-1, libro);
                }else {
                        if (!image.isEmpty()) {
                        String uniqueFileName = uploadFile.copy(image);
                        libro.setImagen(uniqueFileName);
                }
                ListaLibros.addLibro(libro);
        }
        ModelAndView mav= new ModelAndView("listaLibros");
        mav.addObject("libros", ListaLibros.getListLibros());

        return mav;
}
```

```
@GetMapping("/detalle/{id}")
        public ModelAndView getPagePortadaLibro(@PathVariable("id") int id) {
                ModelAndView mav = new ModelAndView("detalleLibro");
                unLibro= ListaLibros.findLibroById(id);
                mav.addObject("titulo", "PORTADA DEL LIBRO "+unLibro.getTitulo());
                mav.addObject("filename",unLibro.getImagen());

                return mav;
        }
```

```
@GetMapping("/uploads/{filename}")
	public ResponseEntity<Resource> goImage(@PathVariable String filename) {
		Resource resource = null;
		try {
			resource = uploadFile.load(filename);
		} catch (MalformedURLException e) {
			e.printStackTrace();
		}
		return ResponseEntity.ok()
				.header(HttpHeaders.CONTENT_DISPOSITION, "attachment;
filename=\"" + resource.getFilename() + "\"")
				.body(resource);
	}
}
```

Método *postPageSaveLibro()*

```
public ModelAndView postPageSaveLibro(@ModelAttribute("unLibro") Libro libro,
      @RequestParam("file") MultipartFile image) throws Exception {
    unAutor=ListaAutores.findAutorById(libro.getAutor().getId());

    libro.setAutor(unAutor);
    if(libro.getId()>0) {
        unLibro= ListaLibros.findLibroById(libro.getId());
        if (!image.isEmpty()) {
            if(unLibro.getImagen() != null && unLibro.getImagen().length() > 0)
                uploadFile.delete(unLibro.getImagen());
            String uniqueFileName = uploadFile.copy(image);
            libro.setImagen(uniqueFileName);
        }else {
            if(unLibro.getImagen() != null)
                libro.setImagen(unLibro.getImagen());
        }
        ListaLibros.libros.set(libro.getId()-1, libro);
    }else {
        if (!image.isEmpty()) {
            String uniqueFileName = uploadFile.copy(image);
            libro.setImagen(uniqueFileName);
        }
        ListaLibros.addLibro(libro);
    }
    ModelAndView mav= new ModelAndView("listaLibros");
    mav.addObject("libros", ListaLibros.getListLibros());

    return mav;
}
```

Con la anotación @RequestParam atrapamos la imagen cargada desde el formulario y lo guardamos en un objeto de tipo MutipartFile.
El cambio se encuentra determinado a el estado de la imagen.
Modificación
Si el objeto *libro* recibido tiene cargado el id estamos ante una modificación, por lo que en la variable *unLibro* guardará el objeto libro actual buscado en la lista. Lo siguiente que hace es preguntar si el objeto *image* se encuentra cargada.
Si se encuentra cargada significa que hemos modificado la imagen en la vista, por lo tanto, pregunta si el objeto actual(*unLibro*) tiene cargada o no una imagen para eliminarla en el caso de que si tuviera.
Luego copia la nueva imagen en la carpeta que definimos al comienzo. Esto lo hace mediante el metodo *copy* de la clase *uploadFile*.
Sino se encuentra cargada una nueva imagen, entonces determina si el objeto actual(*unLibro*) tiene una imagen para poder recuperar el nombre de la imagen y guardar en el objeto que tiene las modificaciones(*libro*). Finalmente se actualiza el libro de la lista.

Alta
Si el objeto libro no tiene cargado un id, estamos agregando un nuevo libro, pero también hace el control para determinar si se va a guardar o no una imagen.
Si el objeto *image* se encuentra cargado copia la imagen en la carpeta que definimos al comienzo. Esto lo hace mediante el método *copy* de la clase *uploadFile*. Por último se da el alta del libro a la lista.

Siguiendo este camino del formulario pasamos a modificar la tabla de la página "*listaLibros.html*" y agregamos una columna para mostrar la imagen.

```
<th scope="col">PORTADA</th>
<td>
  <a th:if="${l.imagen != null}" class="btn btn-warning"
     th:href="@{/libros/detalle/}+${l.id}">VER</a>
</td>
<td >
```

LIBROS

ID	ISBN	TITULO	AUTOR	EDITORIAL	AÑO DE PUBLICACIÓN	PORTADA	ACCIONES
1	12345	EL GATO NEGRO	Jose Mauro de Vasconcelos	LOUIE	2020-12-21	VER	MODIFICAR ELIMINAR

Para este ejemplo se colocó un enlace que redirecciona *(@{/libros/detalle/}+${l.Id})* a otra página para poder mostrar la imagen de un libro en específico, así que creamos una página denominada "detalleLibro.html" que contendrá la siguiente porción de código.

```
<div class="container-fluid ">
        <div class="row col-md-6 offset-md-3 mt-5">
                <div class="card bg-success ">
                    <div class="card-header text-center fs-3 bg-white" th:text="${titulo}"></div>
                    <div class="card-body bg-white text-center">
                        <img class="img-thumbnail rounded float-left"
                             th:src="@{'/libros/uploads/'}+ ${filename}"
                             th:alt="${titulo}"
                             style="max-width:70%">
                    </div>
                    <div class="card-footer bg-white d-flex justify-content-center">
                        <a class="btn btn-success" th:href="@{/libros/lista}">VOLVER</a>
                    </div>
                </div>
        </div>

</div>
```

Ahora vamos al controlador para comprender los métodos que trabajaran con la petición de mostrar una imagen.

Método getPagePortadaLibro()

```
@GetMapping("/detalle/{id}")
public ModelAndView getPagePortadaLibro(@PathVariable("id") int id) {
    ModelAndView mav = new ModelAndView("detalleLibro");
    unLibro= ListaLibros.findLibroById(id);
    mav.addObject("titulo", "PORTADA DEL LIBRO "+unLibro.getTitulo());
    mav.addObject("filename",unLibro.getImagen());

    return mav;
}
```

Este método recibe el id como dato mediante la url, y atrapa el dato en la variable id mediante la anotación @PathVariable. Ya lo demás como hemos estado viendo es conocido. Se indica la vista que se va a mostrar, luego se busca el libro, en la lista, mediante su id y finalmente se agregan a la vista el título y el nombre de la portada del libro.

Como pudimos ver al realizar la petición el método nos redirige a la página "detalleLibro.html"

```
10        <div class="container-fluid ">
11        <div class="row col-md-6 offset-md-3 mt-5">
12            <div class="card bg-success ">
13                <div class="card-header text-center fs-3 bg-white" th:text="${titulo}"></div>
14                <div class="card-body bg-white text-center">
15                    <img class="img-thumbnail rounded float-left"
16                         th:src="@{'/libros/uploads/'}+ ${filename}"
17                         th:alt="${titulo}"
18                         style="max-width:70%">
19                </div>
20                <div class="card-footer bg-white d-flex justify-content-center">
21                    <a class="btn btn-success" th:href="@{/libros/lista}">VOLVER</a>
22                </div>
23            </div>
24
25        </div>
```

Lo importante de esta porción de la pagina es que utilizamos el titulo enviado por el método para mostrarlo en la línea 13 y como atributo *alt* de la imagen. Otro observación es como se muestra la imagen en la línea 16, ya que el atributo th:src que tiene que contener la url de la imagen nos redirecciona a otra petición que analizaremos a continuación.

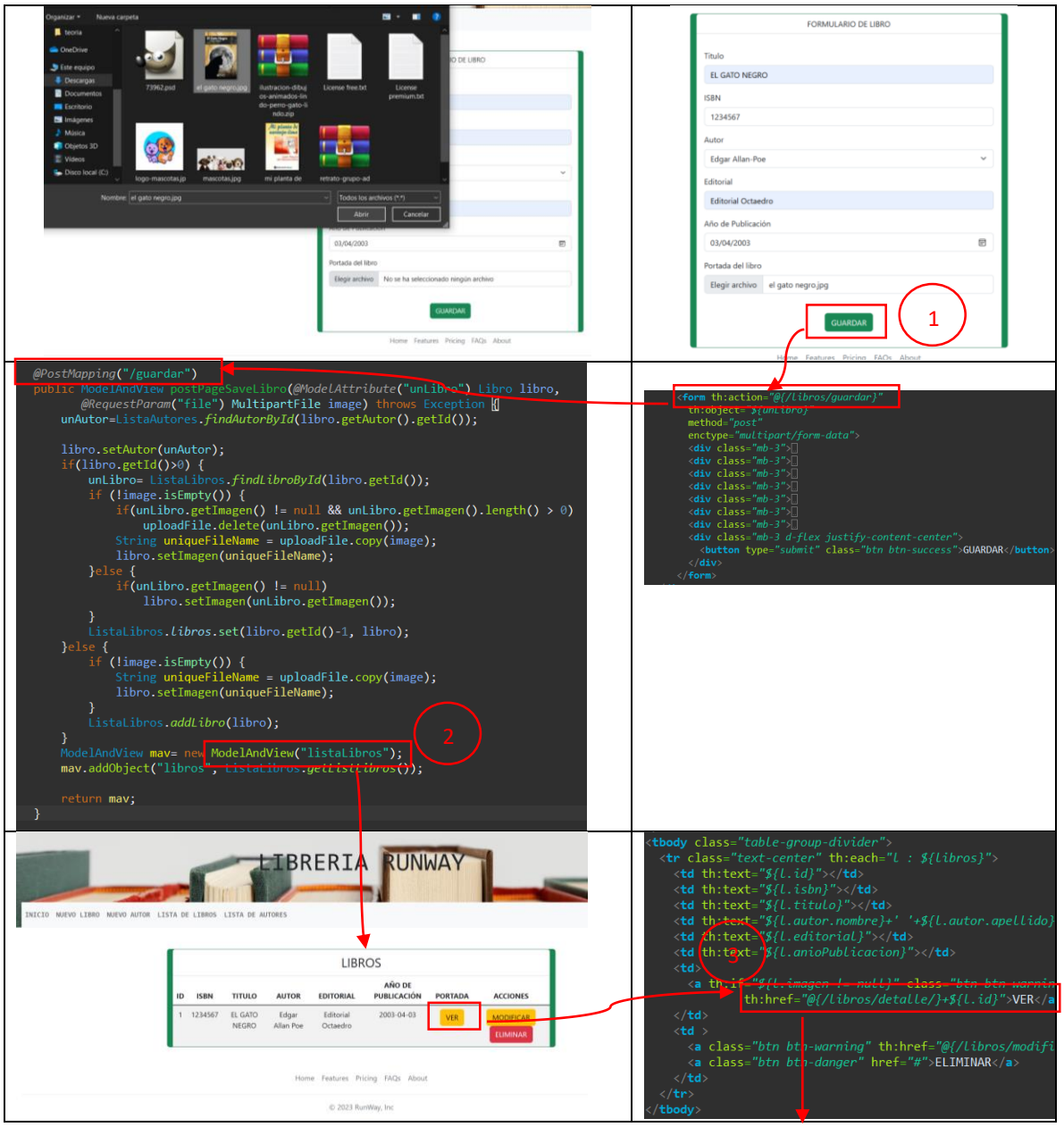
Método goImage()

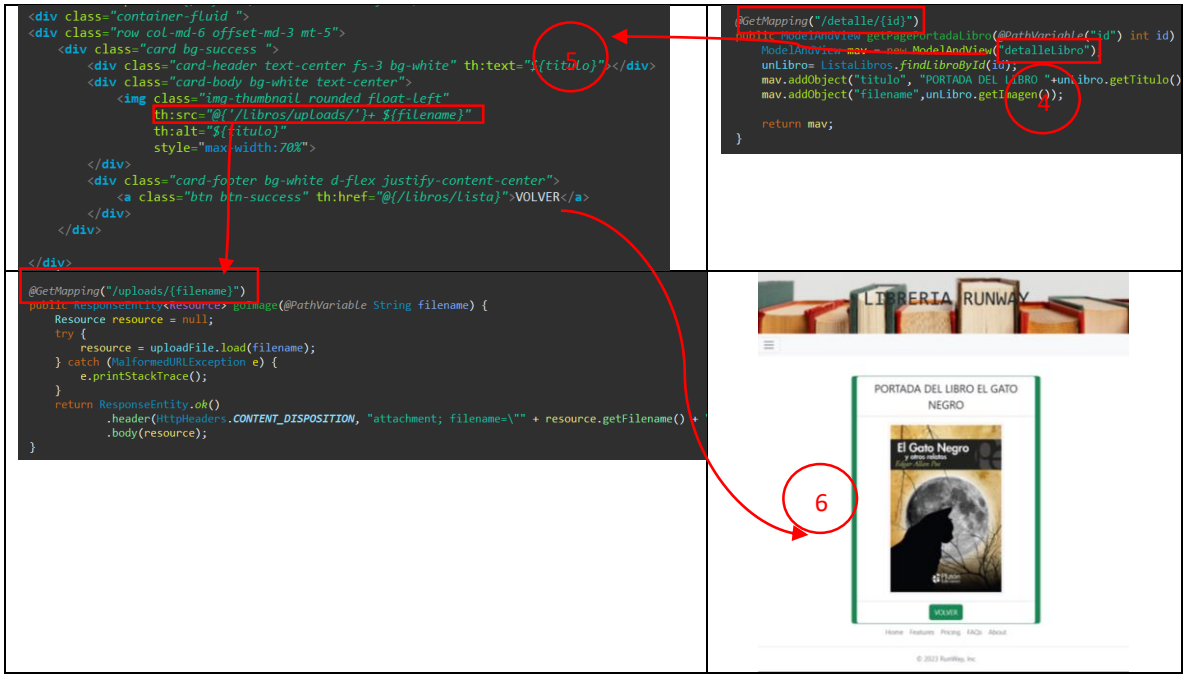
```
@GetMapping("/uploads/{filename}")
public ResponseEntity<Resource> goImage(@PathVariable String filename) {
    Resource resource = null;
    try {
        resource = uploadFile.load(filename);
    } catch (MalformedURLException e) {
        e.printStackTrace();
    }
    return ResponseEntity.ok()
            .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" + resource.getFilename() + "\"")
            .body(resource);
}
```

Este método recibe el nombre de la imagen(enviado junto a la url) y mediante la anotación @PathVariable atrapamos el valor. El método devuelve un objeto del tipo ResponseEntity, esto lo hacemos para poder trabajar con más libertad a la hora de cargar la imagen ya que sin estas sentencias no se podrá mostrar las imágenes cargadas a la carpeta uploads, ya que necesitaríamos actualizar la carpeta cuando el proyecto se encuentra en ejecución para ver los archivos.

Finalmente, el recorrido quedaría de la siguiente forma:

ALTA DE UN LIBRO





AJUSTE DE LOS LIMITES DE CARGA DE ARCHIVOS

Por lo general la configuración de carga de archivos en Spring Boot esta establecido como máximo 128Kb. Es por esto que vamos a configurar la carga de archivos, vamos al archivo application.properties (src/main/resources/application.properties) y agregamos lo siguiente