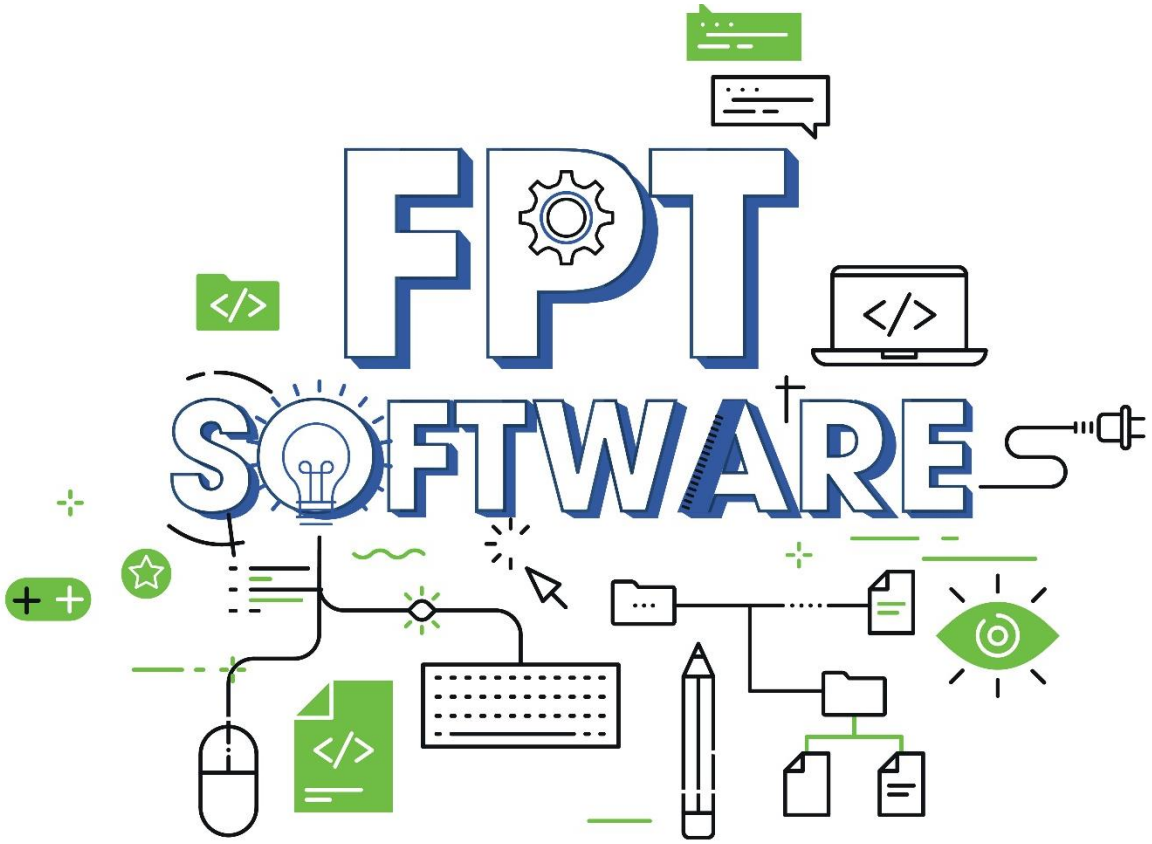
SPRING BOOT

Upload and Download File

Design by: DieuNT1



Agenda

1. Introduction
2. Spring Boot File Upload and Download Rest API
3. Thymeleaf File Upload with Spring Boot
4. Question and Answer

Lesson Objectives

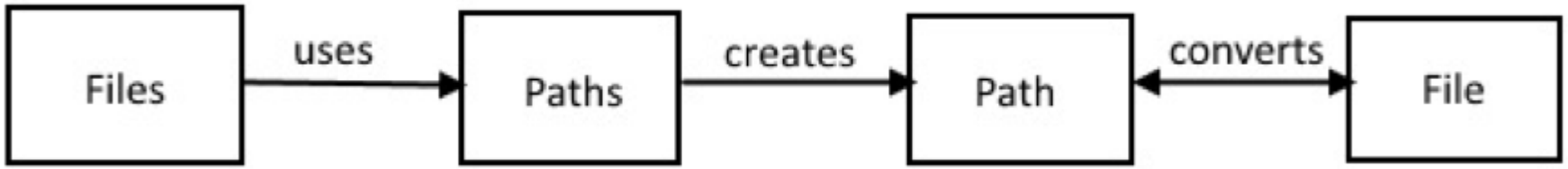
1. Understand the fundamentals of file uploading and downloading using Spring Boot.
2. Able to use Multipart file uploads in Spring Boot applications.
3. Process uploaded files and store them on the server's filesystem or other storage solutions.
3. Integrate error handling and validation for file uploads and downloads.

Section 1

Introduction

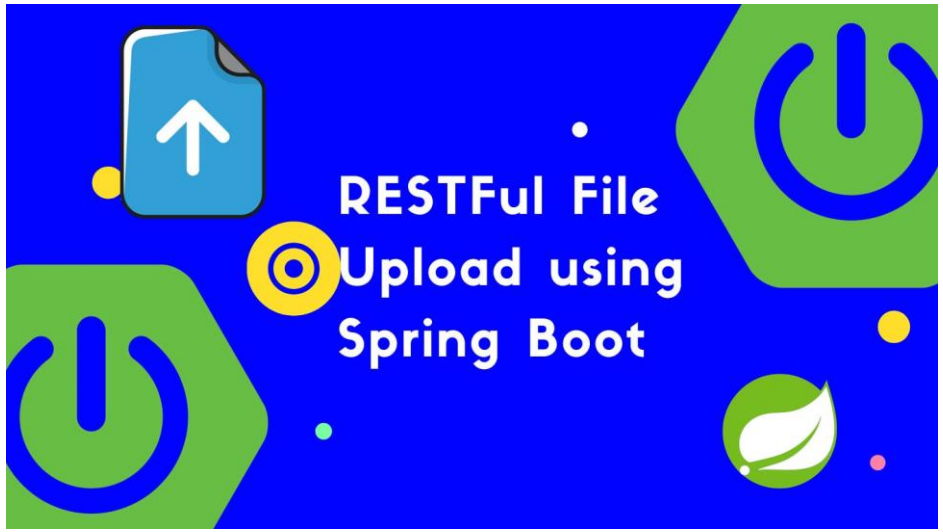
Introduction

- In modern **Web Applications**, file uploading is a common functionality that can be easily implemented using **Spring Boot** and **REST API**.
- In this lesson, we'll discuss how to create a Spring Boot application to upload files, including details about project structure, controllers, and services, and testing the functionality through **Postman**.
- Before participating in this lesson, trainees are required to have knowledge of *Java IO*, understanding of *Path*, *Paths* and its basic methods.



Introduction

- Spring Boot provides a powerful and straightforward way to implement **file uploading and downloading** functionality in your applications.
- This functionality allows users to send files to your server (upload) and retrieve files from your server (download).



Section 2

Spring Boot File Upload and Download Rest API

Tools and Technologies Used

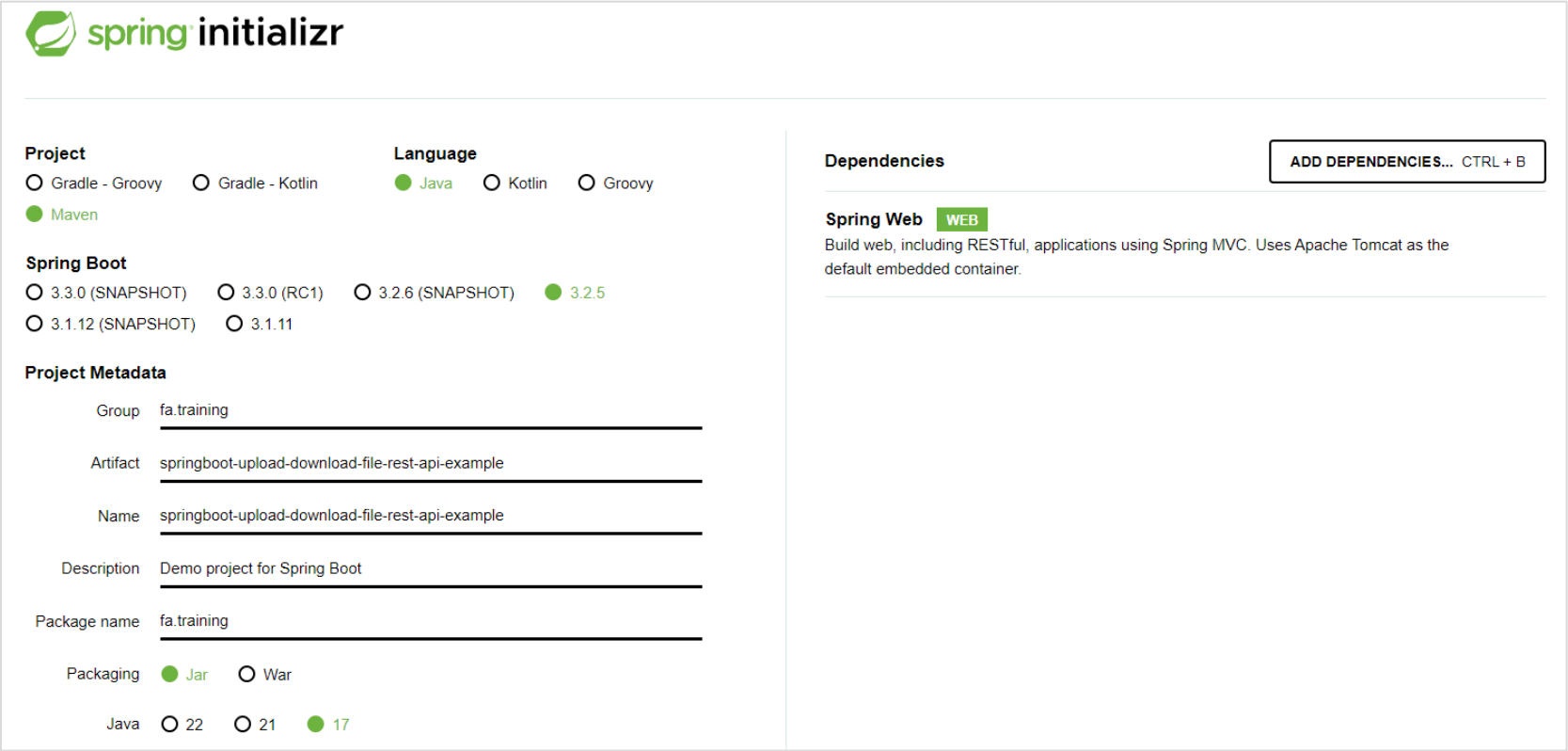
We'll first build the REST APIs for uploading and downloading files, and then test those APIs using Postman.

Tools and Technologies Used:

- Spring Boot - 3+
- JDK - 17 or later
- Spring Framework - 6+
- Maven - 3.6+
- IDE - Eclipse or Spring Tool Suite (STS)

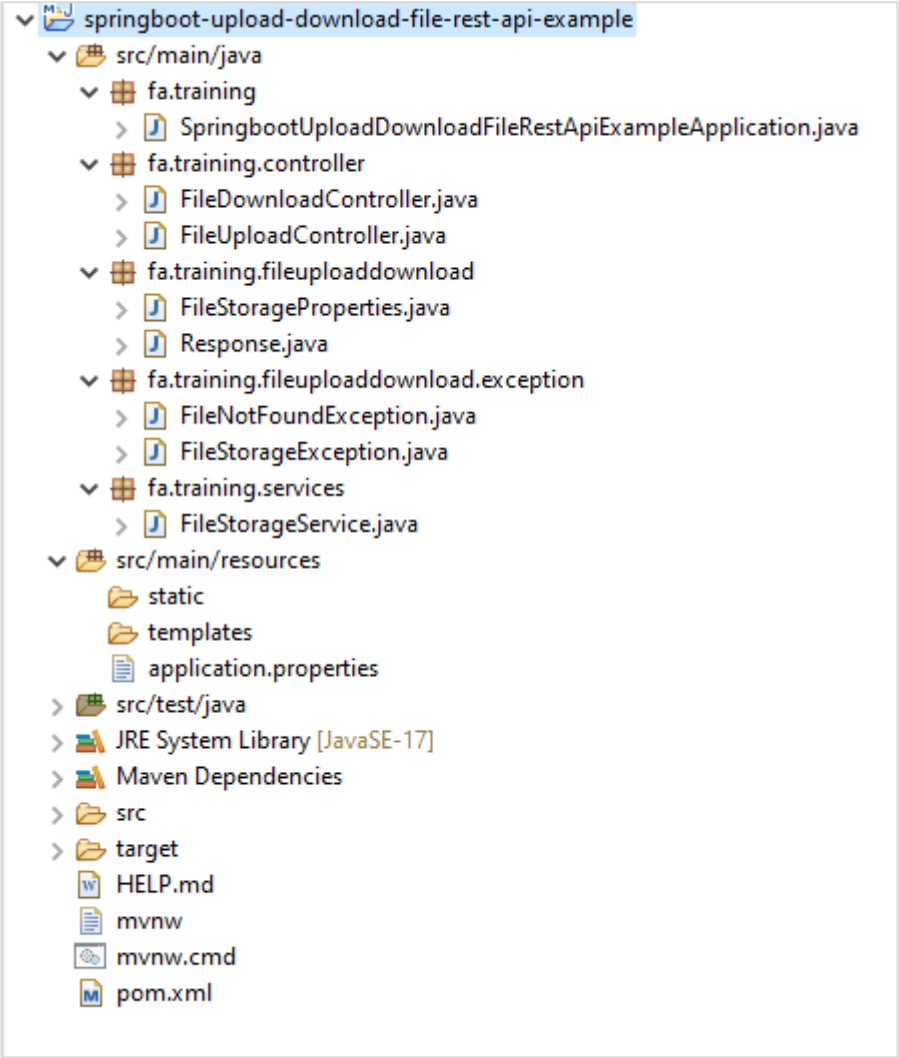
1. Create and Import Spring Boot Project

- The simplest way is to use Spring Initializr at **http://start.spring.io/**, which is an online Spring Boot application generator.



1. Create and Import Spring Boot Project

- Project Structure:



2. The pom.xml File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.2.5</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<groupId>fa.training</groupId>
<artifactId>springboot-upload-download-file-rest-api-
example</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springboot-upload-download-file-rest-api-example</name>
<description>Demo project for Spring Boot</description>

<properties>
    <java.version>17</java.version>
</properties>

<dependencies>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    </dependency>
</dependencies>

<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>


<build>
    <plugins>
    <plugin>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    </plugins>
</build>


</project>
```

3. Configuring Server and File Storage Properties

- Let's configure our Spring Boot application to enable Multipart file uploads, and **define the maximum file size** that can be uploaded.
- We'll also configure the **directory into which all the uploaded files will be stored**.
- Open the *src/main/resources/application.properties* file, and add the following properties to it -

```
spring.application.name=springboot-upload-download-file-rest-api-example

## MULTIPART (MultipartProperties)
# Enable multipart uploads
spring.servlet.multipart.enabled=true
# Threshold after which files are written to disk.
spring.servlet.multipart.file-size-threshold=2KB
# Max file size.
spring.servlet.multipart.max-file-size=200MB
# Max Request Size
spring.servlet.multipart.max-request-size=215MB

## File Storage Properties
file.upload-dir=./uploads

server.port=8081
```

4. Automatically binding properties to a POJO class

- **Spring Boot** has an awesome feature called @ConfigurationProperties using which you can automatically bind the properties defined in the application.properties file to a POJO class.
- Let's define a POJO class called **FileStorageProperties** inside `fa.training.fileuploaddownload` package to bind all the file storage properties -

```
@Component
@ConfigurationProperties(prefix = "file")
@Getter
@Setter
public class FileStorageProperties {
    // file.upload-dir=./uploads --> uploadDir = ./uploads
    private String uploadDir;

}
```

Response class

- This **Response** class is used to return responses from the */uploadFile* and */uploadMultipleFiles* APIs.
- Create a **Response** class inside fa.training.fileuploaddownload.payload package with the following contents -

```
@Setter
@Getter
public class Response {
    private String fileName;
    private String fileDownloadUri;
    private String fileType;
    private long size;


    public Response(String fileName, String fileDownloadUri,
          String fileType, long size) {
        this.fileName = fileName;
        this.fileDownloadUri = fileDownloadUri;
        this.fileType = fileType;
        this.size = size;
    }
}
```

5. Custom Exception Classes

- Following are the definitions of those exception classes will be used in the project.
- All the exception classes go inside the package *fa.training.fileuploaddownload.exception*).
- **FileNotFoundException class**:

```
package fa.training.fileuploaddownload.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class FileNotFoundException extends RuntimeException {

   private static final long serialVersionUID = 1L;


   public FileNotFoundException(String message) {
           super(message);
   }

   public FileNotFoundException(String message, Throwable cause) {
           super(message, cause);
   }
}
```

5. Custom Exception Classes

- **FileStorageException class:**

```
package fa.training.fileuploaddownload.exception;

public class FileStorageException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public FileStorageException(String message) {
        super(message);
    }

    public FileStorageException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

5. Writing APIs for File Upload and Download

- **File Upload Rest API**
  - ✓ Let's now write the REST APIs for uploading single as well as multiple files. Create a new controller class called **FileUploadController** inside *fa.training.controller* package and add the following code to it -

```
package fa.training.controller;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import fa.training.fileuploaddownload.Response;
import fa.training.services.FileStorageService;

@RestController
public class FileUploadController {
   @Autowired
   private FileStorageService fileStorageService;
```

5. Writing APIs for File Upload and Download

```
@PostMapping("/uploadFile")
public Response uploadFile(@RequestParam("file") MultipartFile file) {
    String fileName = fileStorageService.storeFile(file);


    String fileDownloadUri = ServletUriComponentsBuilder.fromCurrentContextPath()
    .path("/downloadFile/")
    .path(fileName).toUriString();


    return new Response(fileName, fileDownloadUri, file.getContentType(), file.getSize());
}


@PostMapping("/uploadMultipleFiles")
public List<Response> uploadMultipleFiles(@RequestParam("files") MultipartFile[] files) {
   return Arrays.asList(files).stream().map(file -> uploadFile(file)).collect(Collectors.toList());
}

} // end FileUploadController class
```

5. Writing APIs for File Upload and Download

- **Explain:**
  - ✓ `org.springframework.web.multipart.MultipartFile interface:` **The *MultipartFile* provides access to details about the uploaded file**, including file name, file type, and so on.
  - ✓ **Methods:**

Method	Description
byte[] **getBytes**()	Return the contents of the file as an array of bytes.
String **getContentType**()	Return the content type of the file.
InputStream **getInputStream**()	Return an InputStream to read the contents of the file from.
String **getName**()	Return the name of the parameter in the multipart form.
String **getOriginalFilename**()	Return the original filename in the client's filesystem.
default Resource **getResource**()	Return a Resource representation of this MultipartFile.
long **getSize**()	Return the size of the file in bytes.
void **transferTo**(File dest)	Transfer the received file to the given destination file.
default void **transferTo**(Path dest)	Transfer the received file to the given destination file.

5. Writing APIs for File Upload and Download

- **Explain:**
  - ✓ `fileStorageService.storeFile(file):` call service methods to store the file in target location.
  - ✓ `ServletUriComponentsBuilder class:` A builder for UriComponents that offers static factory methods to extract information from an HttpServletRequest.
  - ✓ `ServletUriComponentsBuilder.path():` Append to the path of this builder. The given value is appended as-is to previous path values without inserting any additional slashes.
  - ✓ **For example:**
    - builder.path("/first-").path("value/").path("/{id}").build("123") // Results is "/first-value/123

5. Writing APIs for File Upload and Download

- **Explain:**
  - ✓ `ServletUriComponentsBuilder.`*`fromCurrentContextPath`*`():` Return a builder initialized with the host, port, scheme, context path, and the servlet mapping of the given request.
  - ✓ **For example**:
    - If the servlet is mapped by name, i.e. "/main/*", then the resulting path will be /appContext/main.
    - If the servlet path is not mapped by name, i.e. "/" or "*.html", then the resulting path will contain the context path only.

5. Writing APIs for File Upload and Download

- **File Download Rest API:** Create a new controller class called **FileDownloadController** inside *fa.training.controller* package and add the following code to it -

```
package fa.training.controller;

import java.io.IOException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import fa.training.services.FileStorageService;
import jakarta.servlet.http.HttpServletRequest;

@RestController
public class FileDownloadController {
    private static final Logger logger = LoggerFactory.getLogger(FileDownloadController.class);
```

5. Writing APIs for File Upload and Download

```java
@Autowired
private FileStorageService fileStorageService;

@GetMapping("/downloadFile/{fileName:.+}")
public ResponseEntity<Resource> downloadFile(@PathVariable String fileName, HttpServletRequest request) {
    // Load file as Resource
    Resource resource = fileStorageService.loadFileAsResource(fileName);

    // Try to determine file's content type
    String contentType = null;
    try {
        contentType = request.getServletContext().getMimeType(resource.getFile().getAbsolutePath());
    } catch (IOException ex) {
        logger.info("Could not determine file type.");
    }

    // Fallback to the default content type if type could not be determined
    if (contentType == null) {
        contentType = "application/octet-stream";
    }

    return ResponseEntity.ok().contentType(MediaType.parseMediaType(contentType))
    .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" + resource.getFilename() + "\"")
    .body(resource);
} // end downloadFile() method
} // end FileDownloadController class
```

5. Writing APIs for File Upload and Download

- **Explain:**
  - ✓ org.springframework.core.io.**Resource**:
    - Interface for a resource descriptor that abstracts from the actualtype of underlying resource, such as a file or class path resource.
    - An InputStream can be opened for every resource if it exists inphysical form, but a URL or File handle can just be returned forcertain resources. The actual behavior is implementation-specific.
  - ✓ ServletContent**.getMimeType():**
    - Returns the MIME type of the specified file, or null if the MIME type is not known. The MIME type is determined by the configuration of the servlet container, and may be specified in a web application deployment descriptor. Common MIME types are "text/html" and "image/gif".

6. Service for Storing and Retrieving Files

- Create a new class called **FileStorageService.java** inside *fa.training.services* with the following contents -

```
package fa.training.services;

import java.io.IOException;
import java.net.MalformedURLException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;
import org.springframework.web.multipart.MultipartFile;

import fa.training.fileuploaddownload.FileStorageProperties;
import fa.training.fileuploaddownload.exception.FileNotFoundException;
import fa.training.fileuploaddownload.exception.FileStorageException;

@Service
public class FileStorageService {
  private final Path fileStorageLocation;
```

6. Service for Storing and Retrieving Files

```
@Autowired
public FileStorageService(FileStorageProperties fileStorageProperties) {
    this.fileStorageLocation = Paths.get(fileStorageProperties.getUploadDir()).
                                toAbsolutePath().normalize();


    try {
        Files.createDirectories(this.fileStorageLocation);
    } catch (Exception ex) {
        throw new FileStorageException("Could not create the directory where the
                                        uploaded files will be stored.", ex);
    }
}
```

6. Service for Storing and Retrieving Files

```java
public String storeFile(MultipartFile file) {
   // Normalize file name
   String fileName = StringUtils.cleanPath(file.getOriginalFilename());
   try {
     // Check if the file's name contains invalid characters
     if (fileName.contains("..")) {
        throw new FileStorageException("Sorry! Filename contains invalid path sequence "
                                    + fileName);
     }

     // Copy file to the target location (Replacing existing file with the same name)
     Path targetLocation = this.fileStorageLocation.resolve(fileName);
     Files.copy(file.getInputStream(), targetLocation, StandardCopyOption.REPLACE_EXISTING);

     return fileName;

   } catch (IOException ex) {
     throw new FileStorageException("Could not store file "
                 + fileName + ". Please try again!", ex);
   } // end storeFile() method
} // end FileStorageService
```

6. Service for Storing and Retrieving Files

- **Explain:**

(1)`org.springframework.util.StringUtils` class:

- ✓ The Apache Commons Lang 3 library provides support for manipulation of core classes of the Java APIs. This support includes methods for ***handling strings***, ***numbers***, ***dates***, ***concurrency***, ***object reflection*** and ***more***.
- ✓ In addition to providing a general introduction to the library, this tutorial demonstrates methods of the StringUtils class which is used for manipulation of String instances.
- ✓ `StringUtils.cleanPath()` method: Normalize the path by suppressing sequences like "path/.." and inner simple dots.

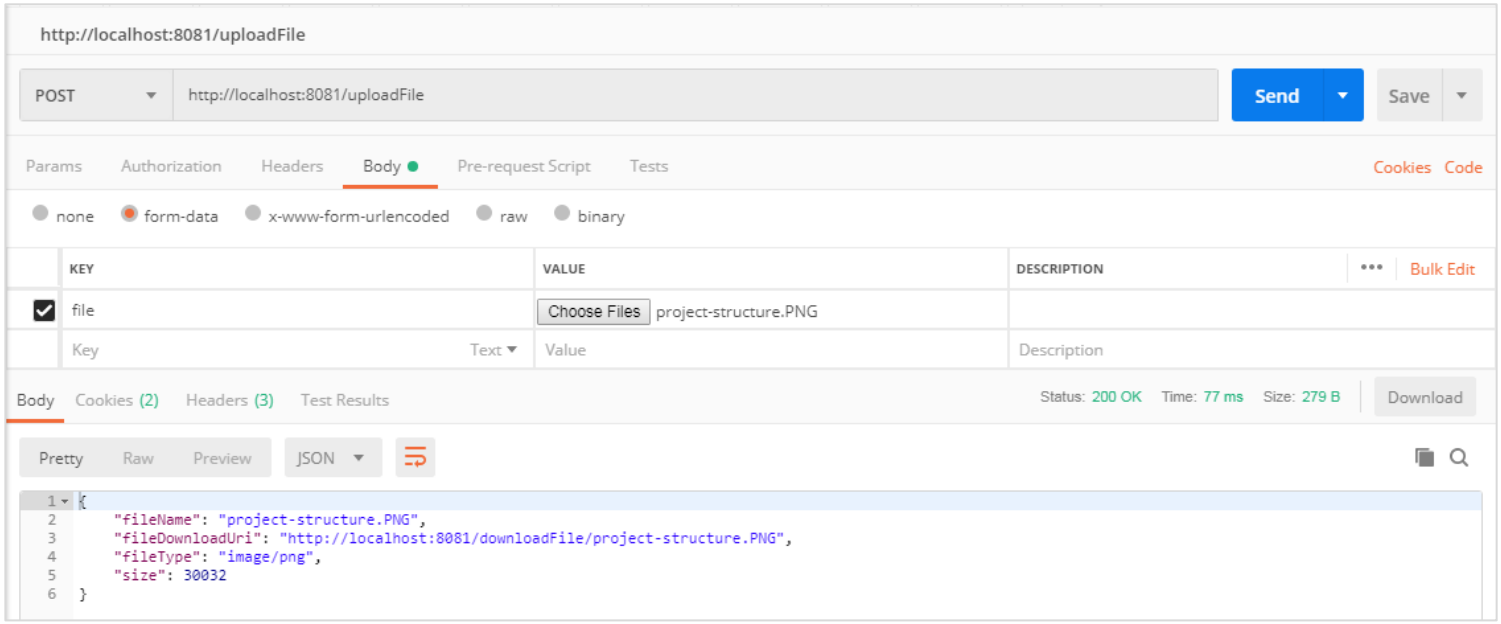
7. Running the Application and Testing the APIs via Postman

- We're done developing our backend APIs. Let's run the application and test the APIs via **Postman**. Type the following command from the root directory of the project to run the application –

```
mvn spring-boot:run
```

- Once started, the application can be accessed at **http://localhost:8081**

1. Upload File



7. Running the Application and Testing the APIs via Postman

2. Upload Multiple Files

POST http://localhost:8081/uploadMultipleFiles Send Save

KEY	VALUE	DESCRIPTION
files	Choose Files 3 files	
Key	Value	Description

Bulk Edit

Body Cookies (2) Headers (3) Test Results Status: 200 OK Time: 125 ms Size: 588 B Download

Pretty Raw Preview JSON


```
[
    {
        "fileName": "new-junior-born.png",
        "fileDownloadUri": "http://localhost:8081/downloadFile/new-junior-born.png",
        "fileType": "image/png",
        "size": 692614
    },
    {
        "fileName": "project-structure.PNG",
        "fileDownloadUri": "http://localhost:8081/downloadFile/project-structure.PNG",
        "fileType": "image/png",
        "size": 30032
    },
    {
        "fileName": "spring-web-annotations.PNG",
        "fileDownloadUri": "http://localhost:8081/downloadFile/spring-web-annotations.PNG",
        "fileType": "image/png",
        "size": 82759
    }
]
```

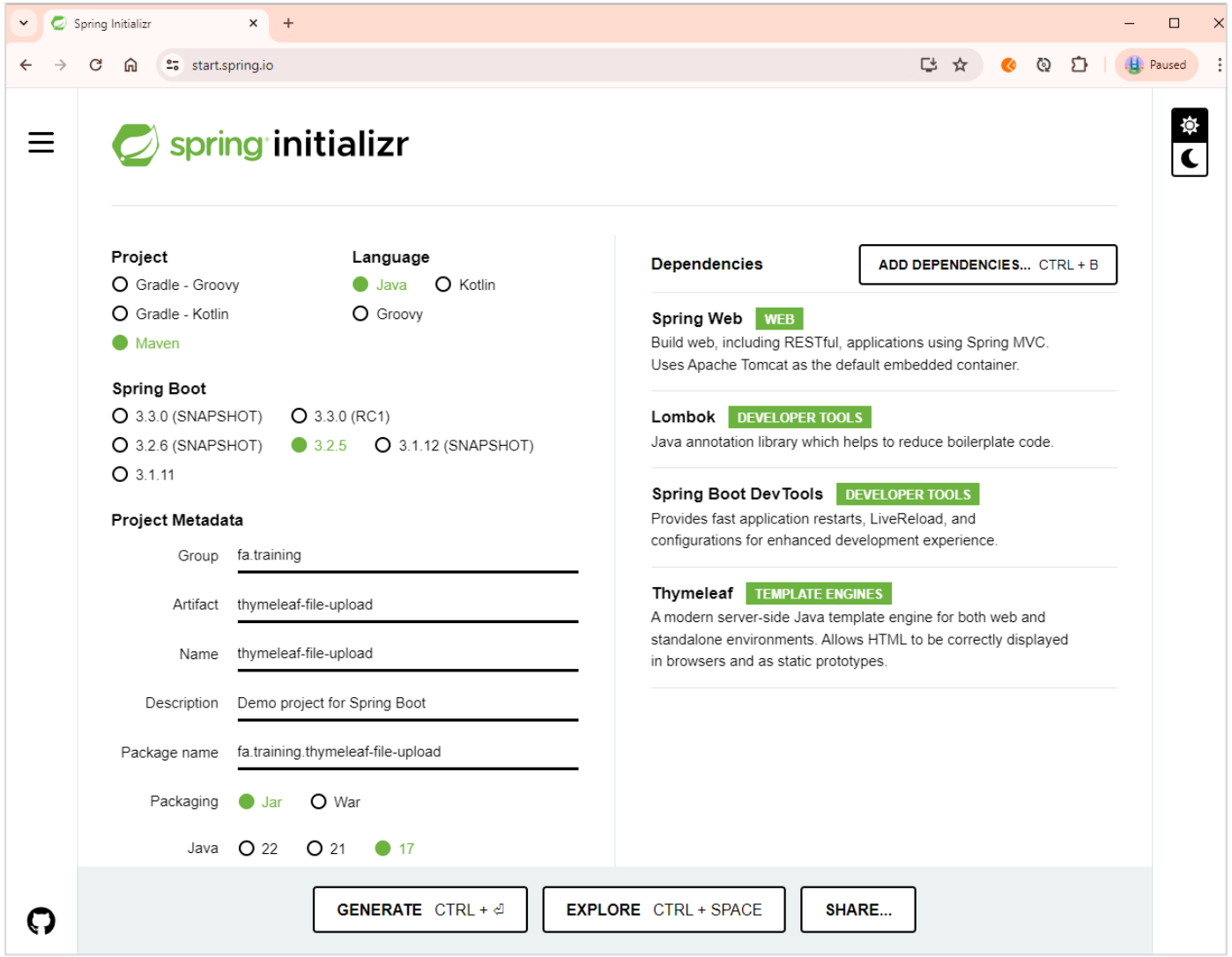
Demo

- The trainer will **code a demo** and **guide the trainee** on how to read an Excel/CSV file after uploading it.
- See more:
  - ✓ https://www.bezkoder.com/spring-boot-upload-excel-file-database/
  - ✓ https://javatechonline.com/read-excel-file-in-java-spring-boot-upload-db/
  - ✓ https://www.bezkoder.com/spring-boot-upload-csv-file/

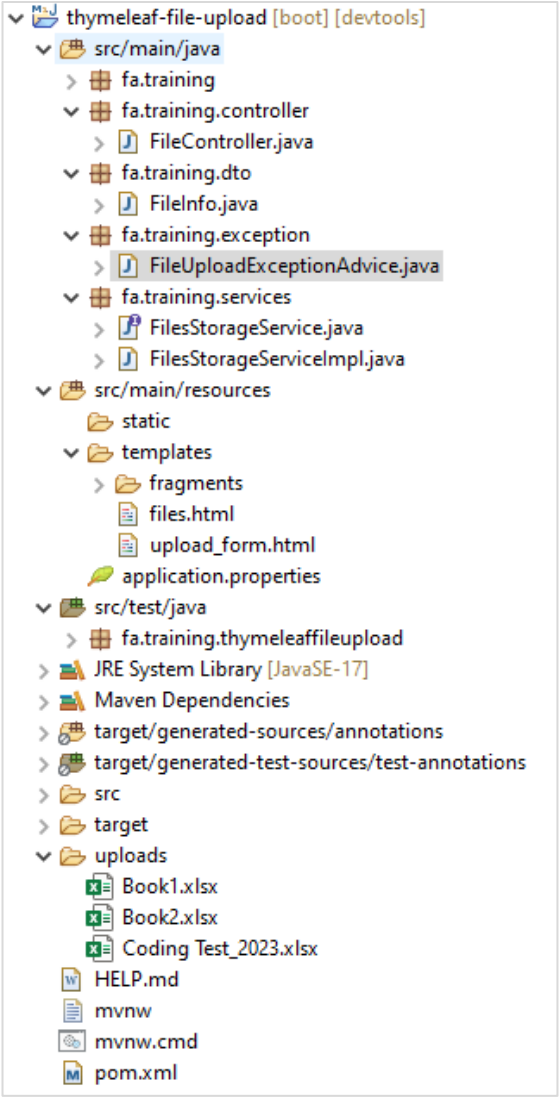
Section 3

Thymeleaf File Upload with Spring Boot

Initialize a new Project



Project Structure



Project Structure

- **Explain:**
  - ✓ **FileInfo** contains information of the uploaded file.
  - ✓ **FilesStorageService** helps us to initialize storage, save new file, load file, get list of Files' info, delete files.
  - ✓ **FileController** uses **FilesStorageService** to handle file upload/download and template requests.
  - ✓ **FileUploadExceptionAdvice** handles exception when the controller processes file upload.
  - ✓ `template` stores HTML template files for the project.
  - ✓ `application.properties` contains configuration for Servlet Multipart.
  - ✓ `uploads` is the static folder for storing files.
  - ✓ **pom.xml** for Spring Boot dependency.

application.properties

- Let's define the maximum file size that can be uploaded in `application.properties` as following:

```
spring.application.name=thymeleaf-file-upload

#
spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=10MB
```

ThymeleafFileUploadApplication class

```java
package fa.training;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import fa.training.services.FilesStorageService;

@SpringBootApplication
public class ThymeleafFileUploadApplication implements CommandLineRunner {
    @Autowired
    private FilesStorageService storageService;

    public static void main(String[] args) {
        SpringApplication.run(ThymeleafFileUploadApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        storageService.init();
    }
}
```

- ✓ ***CommandLineRunner*** is a simple Spring Boot interface with a *run* method.
- ✓ Spring Boot will automatically call the *run* method of all beans implementing this interface after the application context has been loaded.

The pom.xml file

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

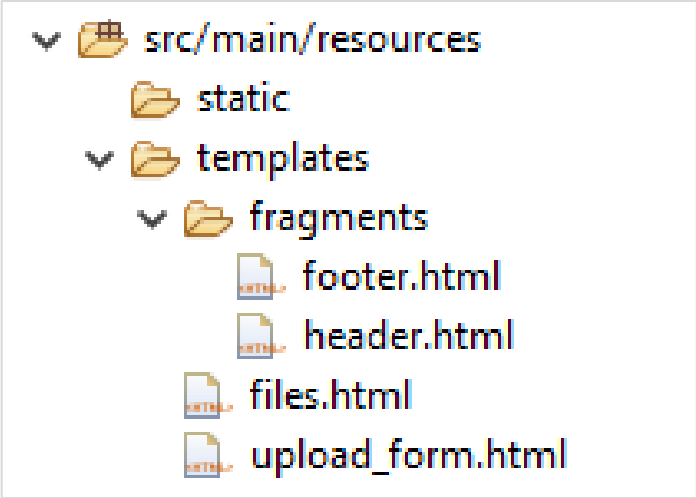
```
<dependency>
   <groupId>org.webjars</groupId>
   <artifactId>bootstrap</artifactId>
   <version>4.6.2</version>
</dependency>

<dependency>
   <groupId>org.webjars</groupId>
   <artifactId>jquery</artifactId>
   <version>3.6.1</version>
</dependency>

<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator-core</artifactId>
</dependency>
```

- ✓ As a Java developer, you are probably familiar with the JAR (Java Archive) file format, which is used to package many class files and their associated metadata into a single file.
- ✓ WebJars is simply taking the concept of a JAR and applying it to client-side libraries or resources. For example, the jQuery library may be packaged as a JAR and made available to your Spring MVC application.

See more WebJars: https://www.baeldung.com/maven-webjars, https://spring.io/blog/2014/01/03/utilizing-webjars-in-spring-boot

1. Setup the Template

- In **src/main/resources** folder, create folder and file as following structure:



1. Setup the Template

- **Header and Footer:** We will use Thymeleaf Fragments (th:fragment) to reuse some common parts such as header and footer.
  - ✓ **fragments**/*header.html*

```
<header th:fragment="header">
    <nav class="navbar navbar-expand-md bg-dark navbar-dark mb-3">
       <a class="navbar-brand" th:href="@{/files}">
       FSA Training
    </a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#topNavbar">
        <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="topNavbar">
       <ul class="navbar-nav">
          <li class="nav-item"><a class="nav-link" th:href="@{/files/new}">Upload</a></li>
          <li class="nav-item"><a class="nav-link" th:href="@{/files}">Files</a></li>
       </ul>
    </div>
    </nav>
</header>
```

  - ✓ **fragments**/*footer.html*

```
<footer class="text-center">
    Copyright © FSA.Training
</footer>
```

1. Setup the Template

- upload-form.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <meta name="viewport" content="width=device-width,initial-scale=1.0,minimum-scale=1.0" />
  <title>FSA Training - Thymeleaf File Upload example</title>
  <link rel="stylesheet" type="text/css" th:href="@{/webjars/bootstrap/css/bootstrap.min.css}" />
  <script type="text/javascript" th:src="@{/webjars/jquery/jquery.min.js}"></script>
  <script type="text/javascript" th:src="@{/webjars/bootstrap/js/bootstrap.min.js}"></script>
</head>

<body>
<div th:replace="fragments/header :: header"></div>
```

1. Setup the Template

- upload-form.html

```
<div class="container" style="max-width: 800px; height: 300px">
<h3 class="mb-3">Thymeleaf File Upload example</h3>

<form id="uploadForm" method="post" th:action="@{/files/upload}" enctype="multipart/form-data">
  <input id="input-file" type="file" name="file" />
  <button class="btn btn-sm btn-outline-success float-right" type="submit"> Upload </button>
</form>

<div th:if="${message != null}" class="alert alert-secondary alert-dismissible fade show text-center
                                        message mt-3" role="alert"> [[${message}]]
   <button type="button" class="close btn-sm" data-dismiss="alert" aria-label="Close">
      <span aria-hidden="true">×</span>
   </button>
</div>
</div>

<hr>
<div th:replace="fragments/footer :: footer"></div>
</body>

</html> <!-- end upload-form.html -->
```

2. Create Service for File Storage

- First we need an interface that will be autowired in the Controller.
- In `fa.training.services` package, create **FilesStorageService** interface like following code:

```
package fa.training.services;

import java.nio.file.Path;
import java.util.stream.Stream;

import org.springframework.core.io.Resource;
import org.springframework.web.multipart.MultipartFile;

public interface FilesStorageService {
  public void init();

  public void save(MultipartFile file);

  public Resource load(String filename);

  public void delete(String filename);

  public Stream<Path> loadAll();
}
```

2. Create Service for File Storage

- Now we create implementation of the interface.

```
package fa.training.services;

import java.io.IOException;
import java.net.MalformedURLException;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;

import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;
import org.springframework.stereotype.Service;
import org.springframework.util.FileSystemUtils;
import org.springframework.web.multipart.MultipartFile;

@Service
public class FilesStorageServiceImpl implements FilesStorageService {
  private final Path root = Paths.get("./uploads");
```

2. Create Service for File Storage

- Now we create implementation of the interface.
- `FilesStorageServiceImpl.init()`: the method is called when the application starts in the run() method. See the above **ThymeleafFileUploadApplication** class.

```
@Override
public void init() {
    try {
       Files.createDirectories(root);
    } catch (IOException e) {
       throw new RuntimeException("Could not initialize folder for upload!");
    }
}
```

2. Create Service for File Storage

- The **save**() method to store a file to the filesystem.

```
@Override
public void save(MultipartFile file) {
  try {
    Files.copy(file.getInputStream(), this.root.resolve(file.getOriginalFilename()));
  } catch (Exception e) {
    if (e instanceof FileAlreadyExistsException) {
      throw new RuntimeException("A file of that name already exists.");
    }
    throw new RuntimeException(e.getMessage());
  }
}
```

2. Create Service for File Storage

- The **load**() method to read file and return a resourse object:

```
@Override
public Resource load(String filename) {
   try {
      Path file = root.resolve(filename);
      Resource resource = new UrlResource(file.toUri());

      if (resource.exists() || resource.isReadable()) {
         return resource;
      } else {
         throw new RuntimeException("Could not read the file!");
      }

   } catch (MalformedURLException e) {
      throw new RuntimeException("Error: " + e.getMessage());
   }
}
```

2. Create Service for File Storage

- The **loadAll**(): read all files to display.

```
    @Override
    public void delete(String filename) {
        try {
            Path path = root.resolve(filename); // delete a selected file
            Files.delete(path);
        } catch (IOException e) {
            throw new RuntimeException("Error: " + e.getMessage());
        }
    }

    @Override
    public Stream<Path> loadAll() {
        try {
            return Files.walk(this.root, 1)
            .filter(path ->!path.equals(this.root))
            .map(this.root::relativize);

        } catch (IOException e) {
           throw new RuntimeException("Could not load the files!");
        }
    }
}
```

3. Create Controller for File Upload

- In `fa.training.controller` package, we create **FileController**.
  - ✓ We use @GetMapping and @PostMapping annotation is for mapping HTTP GET & POST requests onto specific handler methods:
    - GET /files/new: newFile() – return upload_form.html template
    - POST /files/upload: uploadFile() – upload a File
  - ✓ `@Autowired` to inject implementation of **FilesStorageService** bean to local variable.
  - ✓ **Code sample:**

```java
package fa.training.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;

import fa.training.services.FilesStorageService;

@Controller
public class FileController {
    @Autowired
    FilesStorageService storageService;
```

3. Create Controller for File Upload

- The **newFile():** a get method to show upload-form screen.

```
@GetMapping("/files/new")
public String newFile(Model model) {
   return "upload_form";
}
```

- **uploadFile()** method:

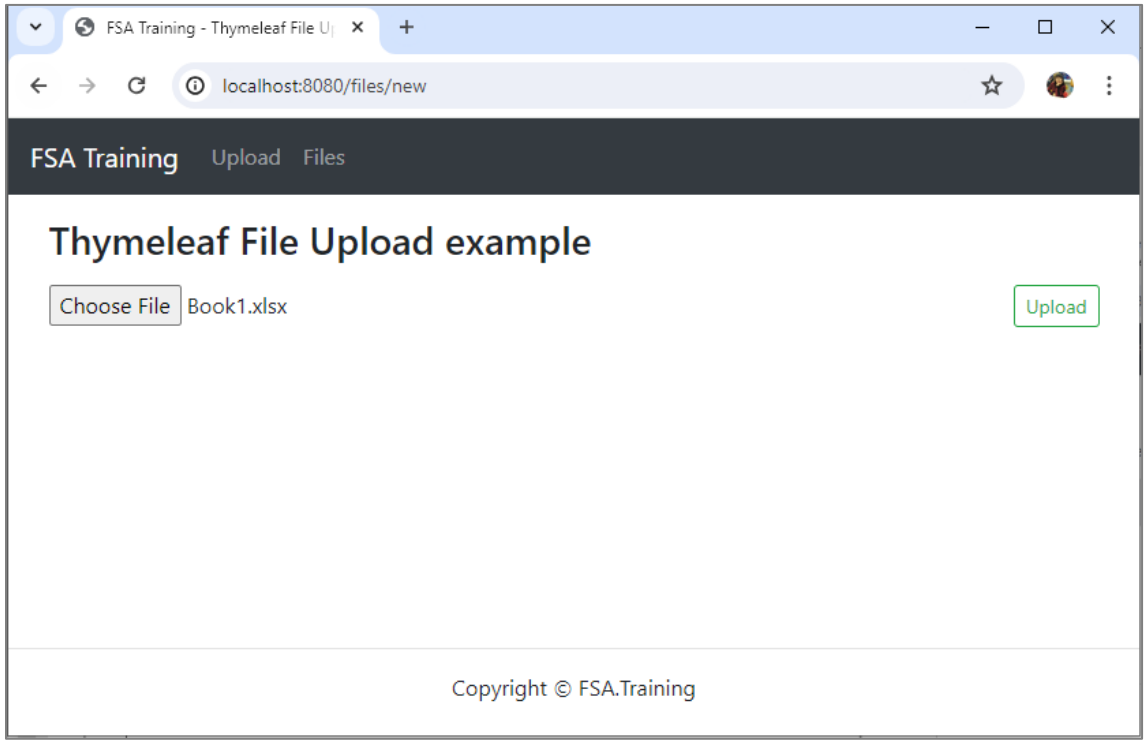
```
   @PostMapping("/files/upload")
   public String uploadFile(Model model, @RequestParam("file") MultipartFile file) {
      String message = "";

      try {
         storageService.save(file);

         message = "Uploaded the file successfully: " + file.getOriginalFilename();
         model.addAttribute("message", message);
      } catch (Exception e) {
         message = "Could not upload the file: " + file.getOriginalFilename() + ". Error: " +
         e.getMessage();
         model.addAttribute("message", message);
      }

      return "upload_form";
   }
} // end FileController class
```

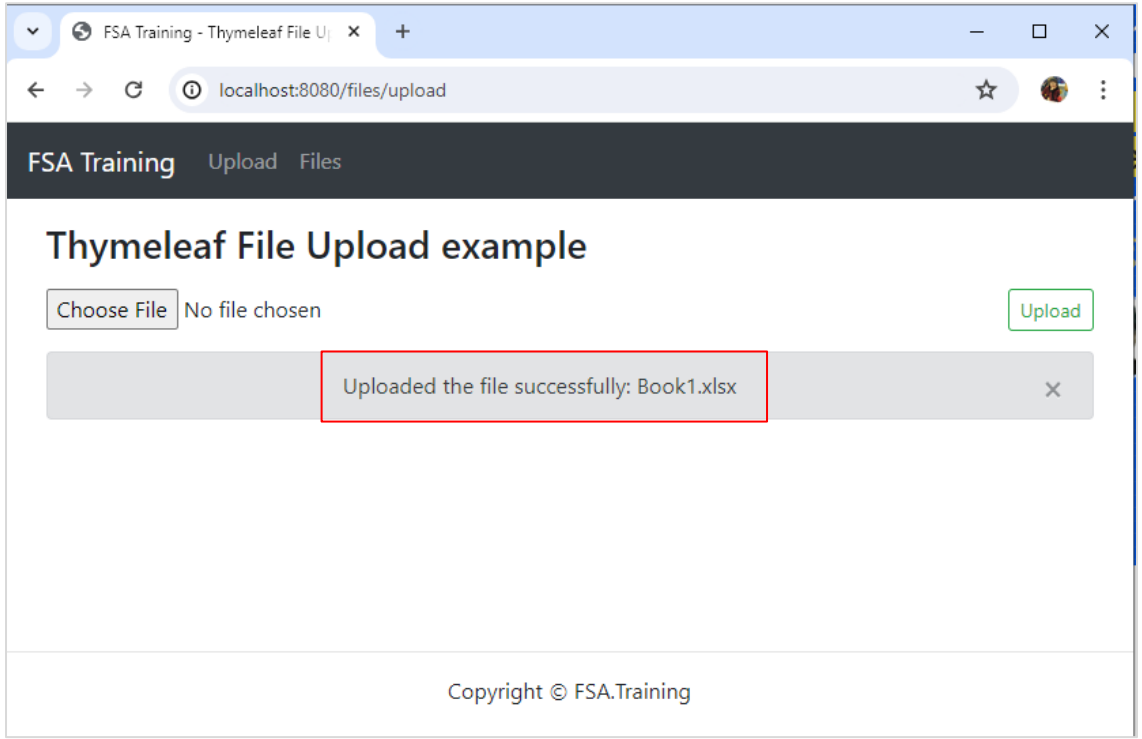
4. Running the Application

- **Run the Spring Boot File Upload example**

```
mvn spring-boot:run
```

- **Access:** http://localhost:8080/files/new





5. Create Controller for Display List of Files

- Firstly, we need to create **FileInfo** class which has fields: `name` & `url`.

```
@Getter
@Setter
public class FileInfo {
   private String name;
   private String url;

   public FileInfo(String name, String url) {
      this.name = name;
      this.url = url;
   }
}
```

5. Create Controller for Display List of Files

- In the Controller, we will return **List of FileInfo** objects as model attribute.
  - ✓ GET /files: **getListFiles**() – return files.html template
  - ✓ GET /files/[filename]: **getFile**() – download a File by filename
- **Code sample**: update *FileController.java*

```
@Controller
public class FileController {

  @Autowired
  FilesStorageService storageService;

  @GetMapping("/")
  public String homepage() {
     return "redirect:/files";
  }

  // …
```

5. Create Controller for Display List of Files

- **getListFiles**() method: list all of files in the filesystem

```
@GetMapping("/files")
public String getListFiles(Model model) {
   List<FileInfo> fileInfos = storageService.loadAll()
        .map(path -> {
             String filename = path.getFileName().toString();
             String url = MvcUriComponentsBuilder.fromMethodName(FileController.class, "getFile",
                                                                 path.getFileName().toString())
                                                 .build()
                                                 .toString();
             return new FileInfo(filename, url);
        })
        .collect(Collectors.toList());

   model.addAttribute("files", fileInfos);

   return "files";
}
```

5. Create Controller for Display List of Files

- **getFile**() method: download a File by filename

```
@GetMapping("/files")
public String getListFiles(Model model) {
   List<FileInfo> fileInfos = storageService.loadAll()
        .map(path -> {
             String filename = path.getFileName().toString();
             String url = MvcUriComponentsBuilder
                   .fromMethodName(FileController.class, "getFile", path.getFileName().toString())
                   .build()
                   .toString();
             return new FileInfo(filename, url);
        })
        .collect(Collectors.toList());

   model.addAttribute("files", fileInfos);

   return "files";
}
```

5. Create Controller for Display List of Files

- **deleteFile**() method: delete a selected file

```
@GetMapping("/files/delete/{filename:.+}")
public String deleteFile(@PathVariable String filename) {

    storageService.delete(filename);

    return "redirect:/files";
}
```

6. Create view for Display List of Files

- *files.html*

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="viewport" content="width=device-width,initial-scale=1.0,minimum-scale=1.0" />
<title>Thymeleaf File Upload example</title>

<link rel="stylesheet" type="text/css" th:href="@{/webjars/bootstrap/css/bootstrap.min.css}" />
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.2.0/css/all.min.css"
integrity="sha512-xh6O/CkQoPOWDdYTDqeRdPCVd1SpvCA9XXcUnZS2FmJNp1coAFzvtCN9BmamE+4aHK8yyUHUSCcJHgXloTyT2A=="
crossorigin="anonymous" referrerpolicy="no-referrer" />

<script type="text/javascript" th:src="@{/webjars/jquery/jquery.min.js}"></script>
<script type="text/javascript" th:src="@{/webjars/bootstrap/js/bootstrap.min.js}"></script>
</head>

<body>
<div th:replace="fragments/header :: header"></div>

<div class="container-fluid" style="max-width: 600px; margin: 0 auto; height: 300px">
<h2 class="text-center">List of Files</h2>
```

6. Create view for Display List of Files

- *files.html*

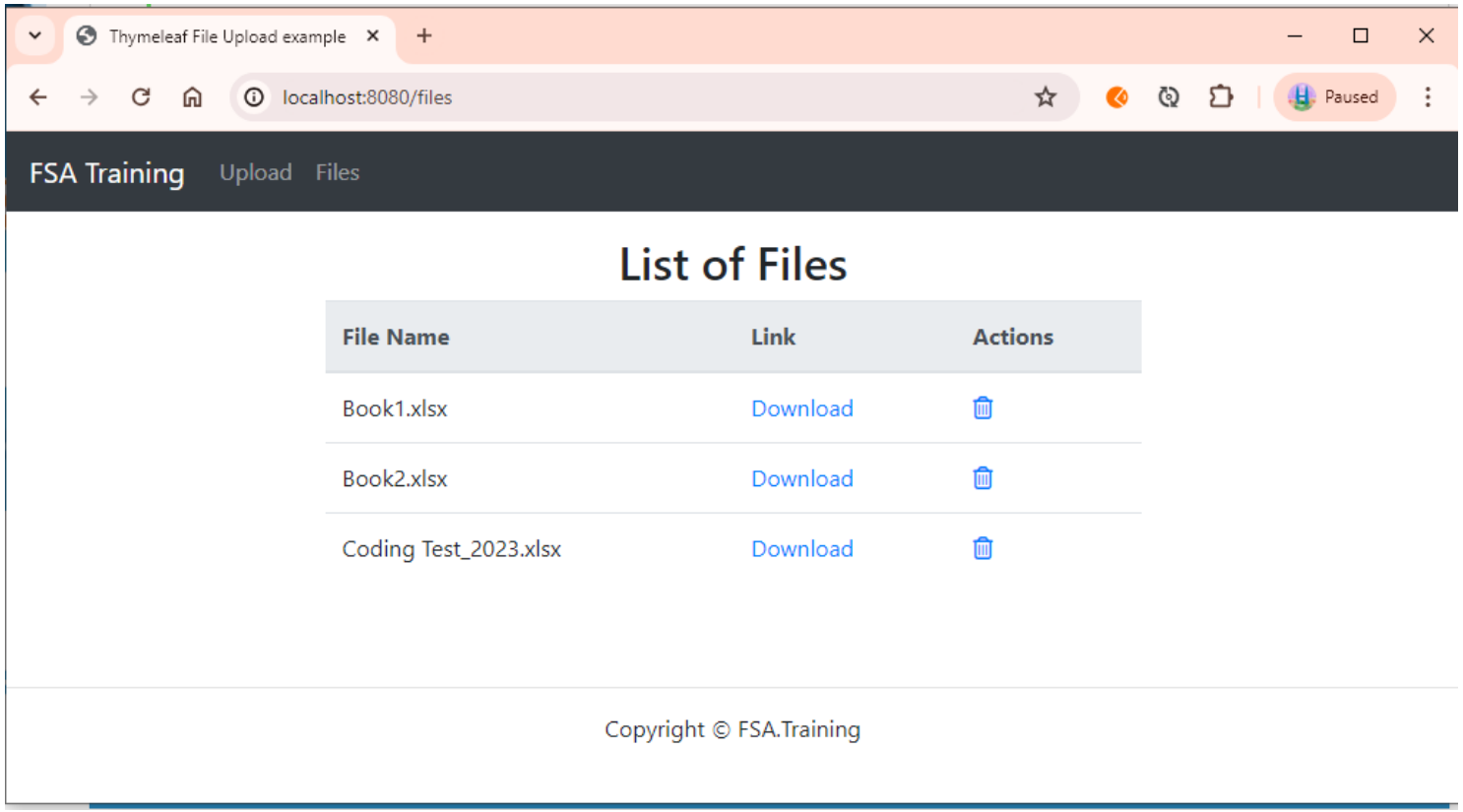
```
<div th:if="${files.size() > 0}">
<table class="table table-hover">
    <thead class="thead-light">
      <tr>
         <th scope="col">File Name</th><th scope="col">Link</th><th scope="col">Actions</th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="file : ${files}">
         <td>[[${file.name}]]</td><td><a th:href="@{${file.url}}">Download</a></td>
         <td><a th:href="@{'/files/delete/' + ${file.name}}" th:fileName="${file.name}" id="btnDelete"
              title="Delete this file" class="fa-regular fa-trash-can icon-dark btn-delete"></a></td>
      </tr>
    </tbody>
</table>
</div>

<div th:unless="${files.size() > 0}"><span>No files found!</span></div>
</div>
<hr>
<div th:replace="fragments/footer :: footer"></div>
</body>

</html> <!-- end files.html -->
```

7. Running the Application

- **Run the Spring Boot File Upload example**

```
mvn spring-boot:run
```

- **Access:** http://localhost:8080/files

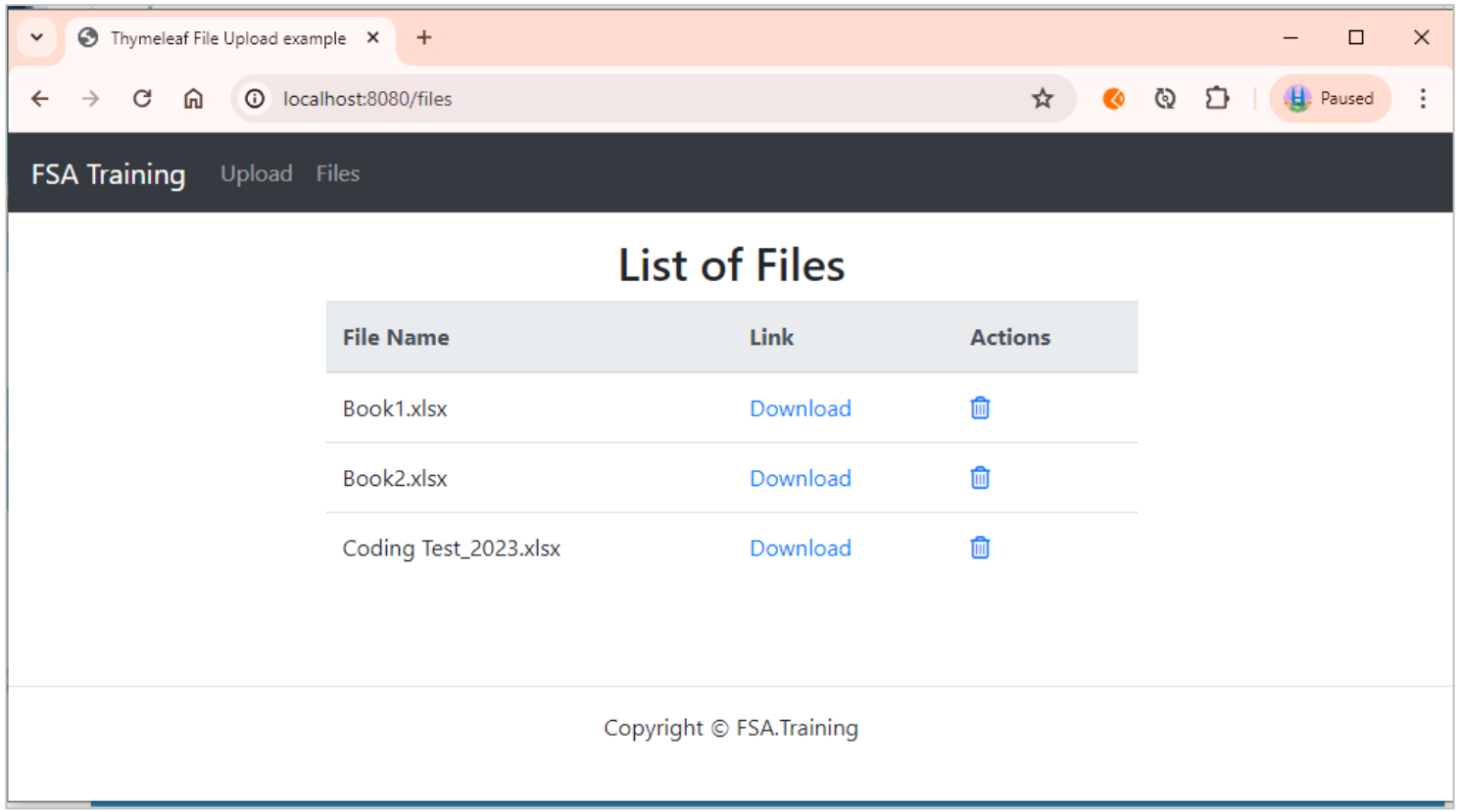


7. Running the Application

- **Run the Spring Boot File Upload example**

```
mvn spring-boot:run
```

- **Access:** http://localhost:8080/files



Handle File Upload Exception

- We handle the case in that a request exceeds Max Upload Size. The system will throw `MaxUploadSizeExceededException` and we're gonna use **`@ControllerAdvice`** with **`@ExceptionHandler`** annotation for handling the exceptions.
- `fa.training.exception.FileUploadExceptionAdvice.java`

```java
package fa.training.exception;

import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.multipart.MaxUploadSizeExceededException;

@ControllerAdvice
public class FileUploadExceptionAdvice {
 @ExceptionHandler(MaxUploadSizeExceededException.class)
 public String handleMaxSizeException(Model model, MaxUploadSizeExceededException e) {
   model.addAttribute("message", "File is too large!");

   return "upload_form";
 }
}
```

Summary

- Introduction
- Spring Boot File Upload and Download Rest API
- Thymeleaf File Upload with Spring Boot
- Q&A

THANK YOU!