Name : Abishek Kumar Sah

USN : 21BTRCS256

Sec : VII Sem ‘D’ Sec

REST APIS USING SPRING BOOT ASSIGNMENT 1

Part – A : Answer

1. Creating a Spring Boot Project Using Spring Initializer:

To create a Spring Boot project, we can use the **Spring Initializer** tool, which provides a quick way to generate a Spring Boot project structure with dependencies.

Steps:

1. Visit Spring Initializer.
2. Select **Maven** as the project type and **Java** as the language.
3. Choose the appropriate **Spring Boot version** (for example, 3.1.0).
4. Fill in the **Group**, **Artifact**, and **Name** fields. Example:
    - **Group:** com.example
    - **Artifact:** demo
    - **Name:** Spring Boot Demo
5. Select dependencies. For this project, add:
    - **Spring Web** (for creating the REST endpoint).
6. Click **Generate** to download the project as a .zip file.
7. Extract the .zip file and import it into IntelliJ.

2. Setting up Maven for Managing Dependencies:

Maven is a build automation tool used primarily for Java projects. It manages dependencies and compiles, packages, and runs the project.

When you create the Spring Boot project using Spring Initializer, Maven is automatically set up. It uses a file named pom.xml (Project Object Model) to manage dependencies and project configurations.

Here's a simple structure for pom.xml:

```
<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
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```xml
<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>demo</name>

<description>Spring Boot Demo</description>


<parent>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-parent</artifactId>

  <version>3.1.0</version>

  <relativePath/> <!-- lookup parent from repository -->

</parent>


<properties>

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

</properties>


<dependencies>

  <!-- Spring Boot Web Dependency -->

  <dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

  </dependency>


  <!-- Test Dependencies -->

  <dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-test</artifactId>

    <scope>test</scope>
```

```
        </dependency>
    </dependencies>

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

3. Creating a Simple REST Endpoint:

Now, let's create a simple REST controller that returns a "Hello World" message.

In the src/main/java/com/example/demo directory, create a new class HelloController.java:

```
package com.example.demo;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello World!";
    }
}
```

4. Role of pom.xml **in the Project:**

The pom.xml file is crucial for managing the project's dependencies, plugins, and build lifecycle in a Maven-based project. It defines:

- **Group ID, Artifact ID, and Version**: These uniquely identify the project.
- **Dependencies**: Third-party libraries and frameworks (like Spring Boot) that the project needs to run. In this case, we added the spring-boot-starter-web dependency, which provides all necessary libraries for building a web application with Spring MVC.
- **Parent**: Defines a base project configuration (in this case, Spring Boot's starter parent) that provides default settings and plugins.
- **Plugins**: Special tools to help with packaging, testing, and running the project.

5. How Maven Manages Classpath Dependencies:

Maven automatically resolves and downloads the necessary dependencies declared in pom.xml from the central Maven repository (or other specified repositories) and adds them to the project's classpath.

In the example above:

- By adding the spring-boot-starter-web dependency, Maven downloads Spring's web framework, including dependencies like Tomcat (an embedded servlet container), Jackson (for JSON serialization), and other necessary libraries.

Maven ensures that the correct versions of dependencies are fetched, and it manages transitive dependencies (dependencies of dependencies) as well.

6. Running the Spring Boot Application:

To run the project:

1. In your IDE, open the DemoApplication.java file inside src/main/java/com/example/demo/. This file is the entry point for the Spring Boot application.

```
package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication {

  public static void main(String[] args) {

    SpringApplication.run(DemoApplication.class, args);

  }

}
```

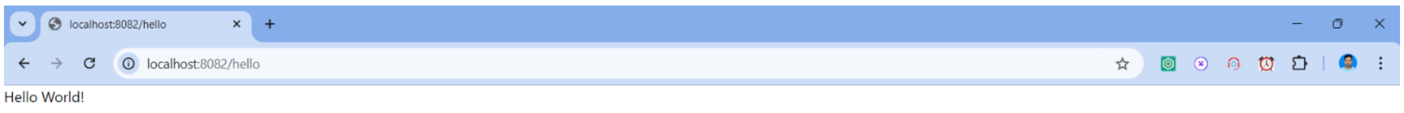
2. Right-click and select **Run DemoApplication**.

3. The application will start, using http://localhost:8082/hello in my browser, which will display "Hello World!".

Conclusion:

By using Spring Initializer and Maven, we can quickly bootstrap a Spring Boot project. The pom.xml file is essential for managing the project's dependencies, build configuration, and plugins. Maven simplifies the process of adding libraries and managing the classpath, allowing developers to focus on writing code without worrying about dependency conflicts.

Output which displays in the Browser:



```
Hello World!
```

Part – B : Answer

Comparing Maven and Gradle in the Context of a Spring Boot Project

1. Build Configuration:

- **Maven**:
  - XML-based configuration in pom.xml.
  - Declarative and rigid structure.
  - Each dependency is declared within <dependencies> tags.
- **Gradle**:
  - Groovy/Kotlin-based configuration in build.gradle.
  - More concise and flexible.
  - Dependencies are declared in a block structure like dependencies { }.

2. Performance:

- **Maven**:
  - Executes tasks sequentially, which may result in slower builds.
  - No built-in support for incremental builds or task avoidance.
- **Gradle**:
  - Supports parallel and incremental builds, which results in faster build times.
  - Supports task avoidance, meaning it only runs tasks if necessary.

3. Dependency Management:

- **Maven**:
  - Centralized dependency management.
  - Repositories and dependencies are listed within <repositories> and <dependencies> in pom.xml.
- **Gradle**:
  - Flexible dependency resolution.
  - Uses repositories like mavenCentral() or jcenter() with dependencies { } blocks for declaring dependencies.

4. Plugin Support:

- **Maven**:
  - Uses plugins defined within the <build> section of pom.xml.
  - Large ecosystem of plugins but can be more verbose to configure.

- **Gradle**:
    - Easier plugin management with the plugins block.
    - More modern and versatile plugin system.

5. Learning Curve:

- **Maven**:
    - Easier to learn due to its straightforward, declarative nature.
    - Popular in enterprise projects.
- **Gradle**:
    - More flexible, but the learning curve is slightly steeper because of its scripting capabilities.
    - Preferred in modern projects due to flexibility and performance improvements.

Creating a Simple Spring Boot Application with Gradle

Now, let's create a **Spring Boot** application using **Gradle** with dependencies for Spring Boot Starter Web and H2 Database.

Step-by-Step Guide:

1. **Initialize the Project Using Spring Initializr:**

    You can either manually set up the project or use the Spring Initializr tool at https://start.spring.io/, selecting **Gradle Project**, and dependencies like:

    - **Spring Web**
    - **H2 Database**

2. **Add Dependencies in** build.gradle**:**
    Here's an example of how to configure your build.gradle file for a Spring Boot application with Spring Web and H2 Database dependencies.

```
plugins {
  id 'org.springframework.boot' version '3.3.4'
  id 'io.spring.dependency-management' version '1.1.3'
  id 'java'
}


group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'
```

```
repositories {
  mavenCentral()
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-web'
  implementation 'com.h2database:h2'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
  useJUnitPlatform()
}
```

Explanation of build.gradle **Structure:**

1. **Plugins Section:**

```
plugins {
  id 'org.springframework.boot' version '3.3.4'
  id 'io.spring.dependency-management' version '1.1.3'
  id 'java'
}
```

- org.springframework.boot: Applies the Spring Boot plugin, which makes it easier to build Spring Boot apps.
- io.spring.dependency-management: This plugin manages dependencies so you don't have to specify their versions manually.
- java: Applies the Java plugin to compile and package Java code.

2. **Group and Version:**

```
group = 'com.example'
version = '0.0.1-SNAPSHOT'
```

- Defines the group ID and version for the project (similar to Maven's groupId and version).

3. **Repositories:**

```
repositories {
  mavenCentral()
}
```

- Specifies where to resolve dependencies. In this case, it's using mavenCentral.

4. **Dependencies Section:**

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-web'
  implementation 'com.h2database:h2'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- implementation: These dependencies are required to run the application.
  - spring-boot-starter-web: This provides Spring MVC and embedded Tomcat for building REST APIs.
  - h2: Provides an in-memory H2 database for development and testing.
- testImplementation: These dependencies are only required for testing. spring-boot-starter-test includes libraries like JUnit and Mockito.

5. **Tasks Section:**

```
tasks.named('test') {
  useJUnitPlatform()
}
```

- Configures the test task to use JUnit 5 (the default for Spring Boot).

Conclusion:

- **Maven** is simpler, with a declarative approach and is widely adopted in enterprise environments.
- **Gradle** is more flexible, offers better performance, and is more modern, but it comes with a steeper learning curve.

Part – C : Answer

To implement a file upload and download feature in Spring Boot, you'll need to create two REST endpoints: one for uploading a file to the server and another for downloading it. We'll use Spring's built-in support for handling multipart file uploads and provide a simple way to save and retrieve files on the server.

Steps to Implement File Upload and Download in Spring Boot

1. **Set up the Spring Boot project**: Create a Spring Boot project with **Spring Web** as a dependency. You can use Maven or Gradle for this project. The key dependency required is:
    - **Spring Boot Starter Web**
2. **Create a directory to store files**: Files will be uploaded and stored in a specific directory on the server.
3. **Implement REST APIs**:
    - An API to upload a file (POST request).
    - An API to download a file (GET request).

Code Implementation

1. **Configure application.properties**

   In your application.properties file, specify the location where the files will be uploaded and stored:

```
file.upload-dir=C:/uploads

server.port=8083
```

   This configuration will allow the application to upload files to the C:/uploads directory.

2. **FileController.java**

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

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;

@RestController
@RequestMapping("/api/files")
public class FileController {

  private final Path fileStorageLocation;

  public FileController(@Value("${file.upload-dir}") String uploadDir) throws IOException {
    this.fileStorageLocation = Paths.get(uploadDir).toAbsolutePath().normalize();
    Files.createDirectories(this.fileStorageLocation);
  }

  // 1. Upload a file
  @PostMapping("/upload")
  public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile file) {
    try {
      // Save the file to the server
      Path filePath = this.fileStorageLocation.resolve(file.getOriginalFilename());
      Files.copy(file.getInputStream(), filePath, StandardCopyOption.REPLACE_EXISTING);
      return ResponseEntity.status(HttpStatus.OK).body("File uploaded successfully: " +
file.getOriginalFilename());
    } catch (IOException e) {
      return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Failed to
upload file: " + e.getMessage());
    }
  }

  // 2. Download a file
  @GetMapping("/download/{fileName:.+}")
  public ResponseEntity<Resource> downloadFile(@PathVariable String fileName) {
    try {
      Path filePath = this.fileStorageLocation.resolve(fileName).normalize();
      Resource resource = new UrlResource(filePath.toUri());
      if (resource.exists()) {
        return ResponseEntity.ok()
            .contentType(MediaType.parseMediaType(Files.probeContentType(filePath)))
            .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" +
resource.getFilename() + "\"")
            .body(resource);
      } else {
```

```
                return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null);
            }
        } catch (MalformedURLException ex) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
        } catch (IOException ex) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
        }
    }
}
```

3. Explanation of Code

1. **File Upload API (POST /api/files/upload)**:
    - The @PostMapping("/upload") method handles the file upload.
    - The @RequestParam("file") MultipartFile captures the uploaded file.
    - The file is saved to a directory specified by file.upload-dir in the application.properties file.
    - The Files.copy() method writes the uploaded file to the target directory.
    - If successful, the method returns a success message; otherwise, it returns an error.
2. **File Download API (GET /api/files/download/{fileName})**:
    - The @GetMapping("/download/{fileName}") method handles file downloading.
    - The file path is normalized, and UrlResource is used to retrieve the file.
    - If the file exists, it's returned to the client with the correct content type and Content-Disposition header (for the browser to download it as an attachment).
    - If the file does not exist, the method returns a 404 (Not Found) status.

4. SpringBootFileUploadDownloadApplication.java

This is the main class for running your Spring Boot application:

```
package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringBootFileUploadDownloadApplication {

  public static void main(String[] args) {

    SpringApplication.run(SpringBootFileUploadDownloadApplication.class, args);

  }
}
```

Steps to Test the File Upload and Download Endpoints

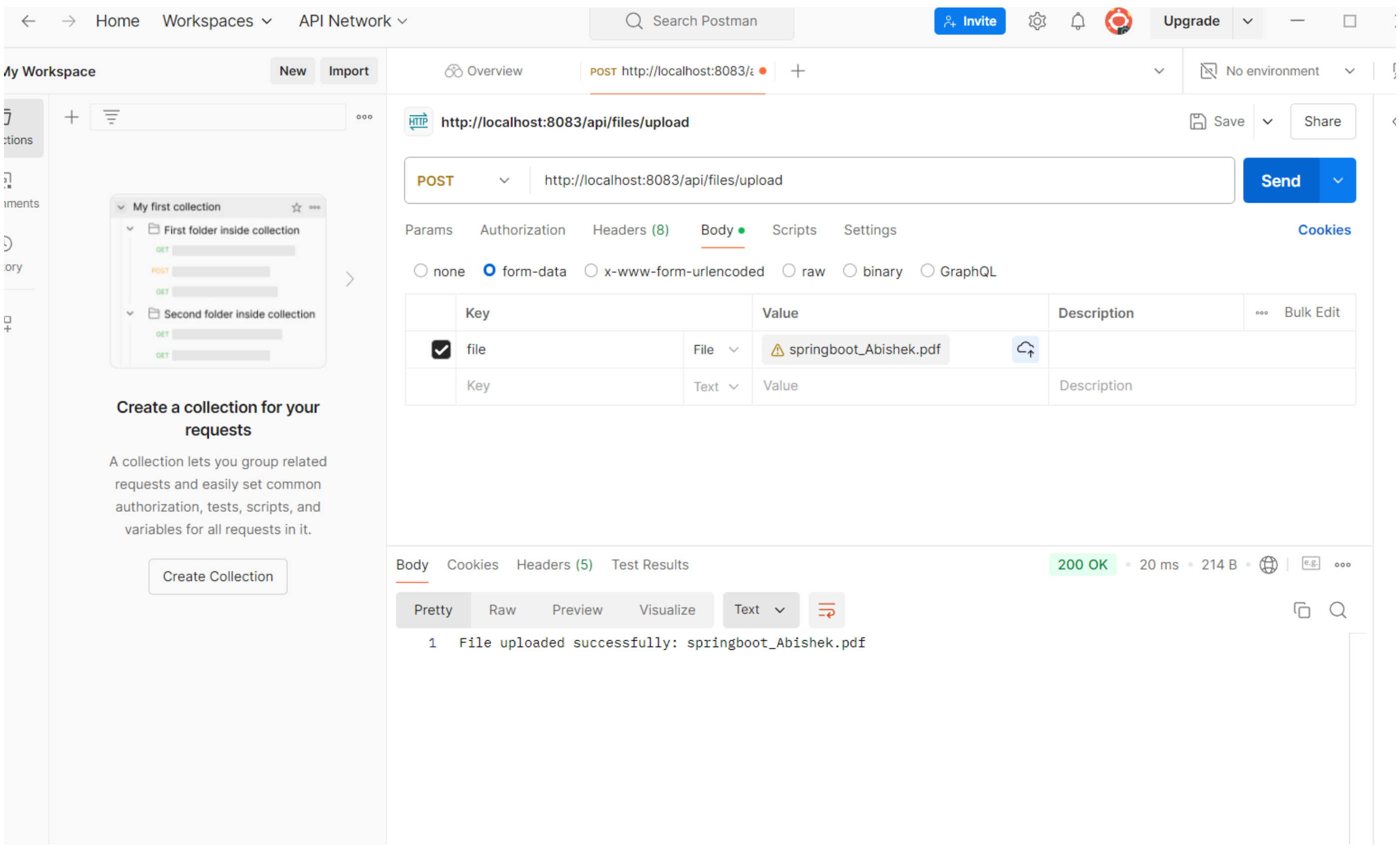
1. Upload a File Using Postman

Here's how you can test the file upload feature:

1. **Open Postman**.
2. **Set the HTTP method to** POST:
   - In the request box (next to the URL bar), select POST from the dropdown.
3. **Enter the URL**:
   - Type this URL into the request URL bar:

     http://localhost:8083/upload
4. **Set the Body to** form-data:
   - Scroll down and click on the **Body** tab below the request URL.
   - Select **form-data**.
5. **Add a Key for the File**:
   - In the form-data section, add a new row.
   - Under the Key column, type file.
   - On the same row, in the Value column, you'll see a dropdown. Change the dropdown from **Text** to **File**.
   - Click on the **Select Files** button and choose a file from your system to upload.
6. **Send the Request**:
   - Click the **Send** button on the top right corner.
7. **Check the Response**:
   - If everything works fine, you should receive a response from the server that looks something like:

     ```
     {
       "message": "File uploaded successfully: your_file_name"
     }
     ```


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
My Workspace
New
Import
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection
Overview
No environment
Save
Share
Send
Cookies

POST http://localhost:8083/api/files/upload

Body: form-data

Key		Value	Description
file	File	springboot_Abishek.pdf	

200 OK · 20 ms · 214 B

```
File uploaded successfully: springboot_Abishek.pdf
```

2. Download the Uploaded File Using Postman

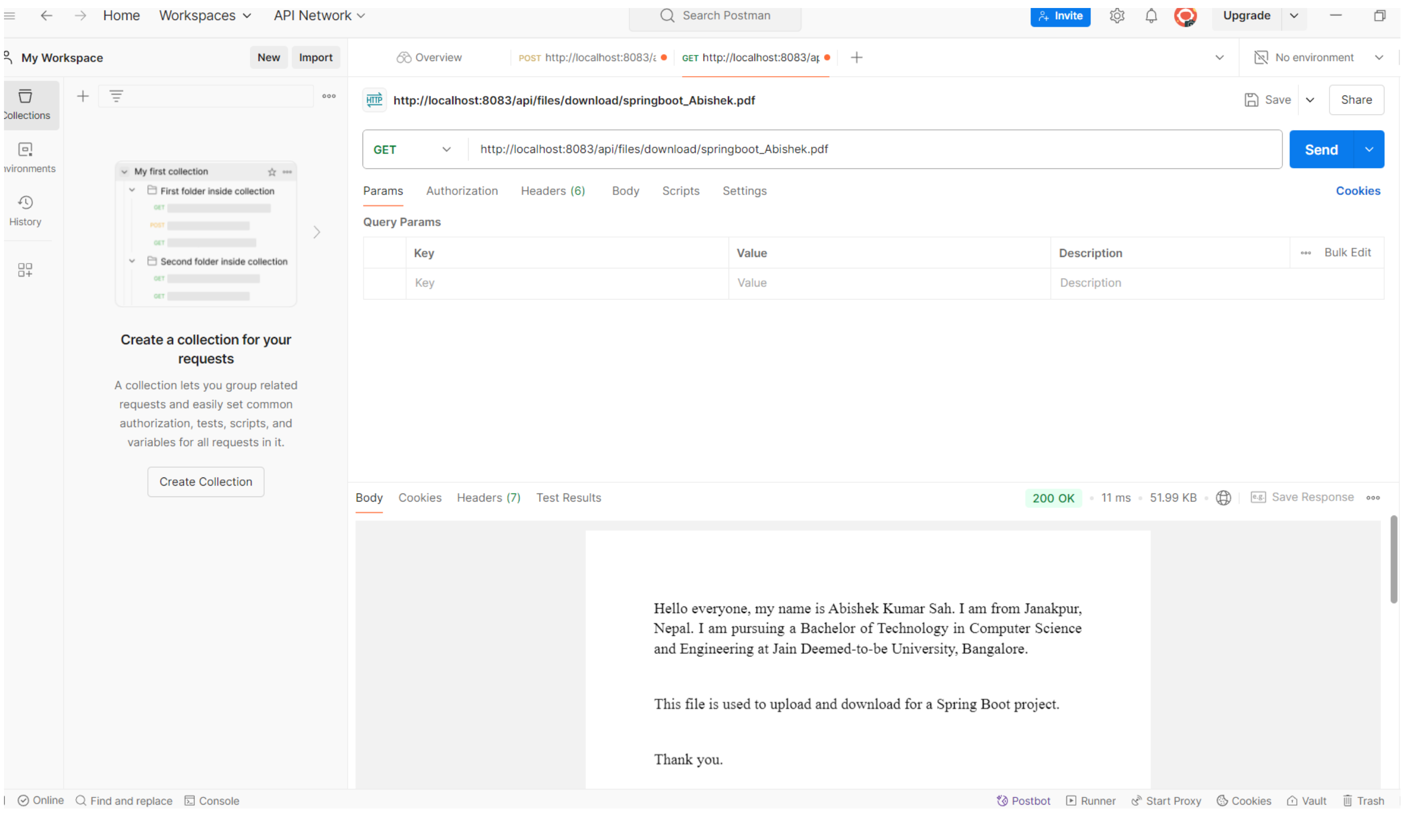
Now, let's test the file download functionality.

1. **Set the HTTP method to** GET:
    - Change the request method to GET.
2. **Enter the URL**:
    - Use the following URL format for downloading the file:

      http://localhost:8083/download/{filename}
    - Replace {filename} with the name of the file you uploaded.

   For example:

   http://localhost:8083/download/myfile.txt
3. **Send the Request**:
    - Click **Send** to make the GET request.
4. **Check the Response**:
    - If successful, the file will be downloaded, and you'll see the response. Postman will show the file content, or it will prompt you to save the file.



Create a collection for your requests

A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.

Create Collection

http://localhost:8083/api/files/download/springboot_Abishek.pdf

GET http://localhost:8083/api/files/download/springboot_Abishek.pdf

Send

Query Params

Key	Value	Description
Key	Value	Description

200 OK · 11 ms · 51.99 KB

Hello everyone, my name is Abishek Kumar Sah. I am from Janakpur, Nepal. I am pursuing a Bachelor of Technology in Computer Science and Engineering at Jain Deemed-to-be University, Bangalore.

This file is used to upload and download for a Spring Boot project.

Thank you.

How the File is Stored on the Server

- **File Upload**: The uploaded file is stored in the directory specified in application.properties. This path is defined in the FileController class. When a file is uploaded, it's saved in the target location using the Files.copy() method. The file is written directly to the filesystem of the server.
- **File Download**: The file is retrieved from the filesystem using UrlResource, which represents a file-based resource in Spring. The file is then served to the client with appropriate headers to ensure the browser downloads it correctly.

Conclusion

You've implemented a file upload and download feature in a Spring Boot application. This simple example demonstrates how files can be stored and retrieved from a server using Spring's multipart file handling and REST APIs. You can extend this functionality by adding validation, file size limits, or database entries to track uploaded files.

THANK YOU!!!