Handling File Upload with Spring MVC and Spring Boot

Estimated time needed: 3 minutes

Learning objectives:

- Explain the configurations required for handling file uploads using Spring MVC with Spring Boot
- Describe the process of creating user interfaces for file uploads using Thymeleaf templates

File uploading is a fundamental feature in modern web applications, enabling users to upload documents, images, and other files. In the Spring ecosystem, Spring Boot simplifies the implementation of this functionality, offering a streamlined and efficient approach. This reading provides a comprehensive walkthrough for handling file uploads using Spring MVC with Spring Boot, ensuring a robust and scalable solution.

Introduction

Spring Boot is a powerful framework for building web applications, providing an easy and efficient way to handle file uploads. Mastering file upload management is crucial for web developers, whether it's for an image gallery, a document management system, or a profile picture upload feature. This reading will walk you through the configurations, controllers, and templates required to implement file uploading seamlessly.

Prerequisites

Before getting started, ensure the following are in place:

- **Java Development Kit (JDK) 8 or later:** Required to run Spring Boot applications.
- **Spring Boot Installation:** Basic knowledge of Spring Boot is beneficial, and installation can be done via the Spring Initializr or your IDE.
- **IDE:** Using an Integrated Development Environment like IntelliJ IDEA or Eclipse will ease development and debugging.

With these prerequisites in place, you can now proceed to set up the Spring Boot project and configure it for file uploads.

Setting up the project

The first step is to create a new Spring Boot project using your preferred IDE or the Spring Initializr. Select the following dependencies:

- **Spring Web:** For building web applications using Spring MVC
- **Thymeleaf:** To render HTML templates
- **Spring Boot DevTools:** To enable live reloading during development

These dependencies provide the necessary tools for building a dynamic and responsive file upload feature.

Configuring the application

Spring Boot offers built-in support for handling file uploads through the MultipartFile interface, simplifying the process significantly. To enable this feature, follow these steps:

Step 1: Configure application.properties
In the src/main/resources/application.properties file, add the following configuration:

```
spring.servlet.multipart.max-file-size=5MB

spring.servlet.multipart.max-request-size=5MB
```

This configuration sets the maximum file size and request size to 5MB, ensuring only files within this limit can be uploaded. Adjust these values as needed to accommodate different file sizes.

Step 2: Create file upload controller

The next step is to create a controller to manage the file upload process. This controller will handle requests, validate inputs, and store uploaded files.

```
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 org.springframework.web.servlet.mvc.support.RedirectAttributes;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
@Controller
public class FileUploadController {
    private static String UPLOADED_FOLDER = "uploads/";
    @GetMapping("/")
    public String index() {
        return "upload";
    }
    @PostMapping("/upload")
    public String singleFileUpload(@RequestParam("file") MultipartFile file,
                                   RedirectAttributes redirectAttributes) {
        if (file.isEmpty()) {
            redirectAttributes.addFlashAttribute("message", "Please select a file to upload");
            return "redirect:/uploadStatus";
        }
        try {
            byte[] bytes = file.getBytes();
            Path path = Paths.get(UPLOADED_FOLDER + file.getOriginalFilename());
            Files.write(path, bytes);
            redirectAttributes.addFlashAttribute("message",
                    "You successfully uploaded '" + file.getOriginalFilename() + "'");
        } catch (IOException e) {
            e.printStackTrace();
        }
        return "redirect:/uploadStatus";
    }
    @GetMapping("/uploadStatus")
    public String uploadStatus() {
        return "uploadStatus";
    }
}
```

Step 3: Create Thymeleaf templates

To provide a user-friendly interface, create two Thymeleaf templates: upload.html and uploadStatus.html.

upload.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>File Upload</title>
</head>
<body>
    <h2>Upload a File</h2>
    <form method="POST" enctype="multipart/form-data" action="/upload">
        <input type="file" name="file"/>
        <button type="submit">Upload</button>
    </form>
    <p th:text="${message}"></p>
</body>
</html>
```

uploadStatus.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Upload Status</title>
</head>
<body>
    <h2>Upload Status</h2>
    <p th:text="${message}"></p>
    <a href="/">Back to upload</a>
</body>
</html>
```

Running the Application

To run the application, use your IDE's built-in Spring Boot runner or the command line:

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

Summary

This reading covered the key concepts of handling file uploads with Spring MVC and Spring Boot. The main steps in handling file upload include:

- Setting up a Spring Boot project with the necessary dependencies.
- Configuring application.properties to manage file size limits.
- Creating a FileUploadController to handle file uploads and store them efficiently.

- Designing user interfaces using Thymeleaf templates for seamless file uploading and status messaging.
- Run and test the application to ensure proper functionality.