GETTING STARTED

Uploading Files

This guide walks you through the process of creating a server application that can receive HTTP multi-part file uploads.

What you'll build

You will create a Spring Boot web application that accepts file uploads. You will also build a simple HTML interface to upload a test file.

What you'll need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 (http://www.oracle.com/technetwork/java/javase/downloads/index.html) or later
- Gradle 2.3+ (http://www.gradle.org/downloads) or Maven 3.0+ (https://maven.apache.org/download.cgi)
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS) (/guides/gs/sts)
  - IntelliJ IDEA (/guides/gs/intellij-idea/)

How to complete this guide

Like most Spring Getting Started guides (/guides), you can start from scratch and complete each step, or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Build with Gradle.

To **skip the basics**, do the following:

- Download (https://github.com/spring-guides/gs-uploading-files/archive/master.zip) and unzip the source repository for this guide, or clone it using Git (/understanding/Git):
  `git clone https://github.com/spring-guides/gs-uploading-files.git (https://github.com/spring-guides/gs-uploading-files.git)`
- cd into `gs-uploading-files/initial`
- Jump ahead to Create an Application class.

When you're finished, you can check your results against the code in `gs-uploading-files/complete`.

▸ Build with Gradle

▸ Build with Maven

▾ Build with your IDE

- Read how to import this guide straight into Spring Tool Suite (/guides/gs/sts/).
- Read how to work with this guide in IntelliJ IDEA (/guides/gs/intellij-idea).

Create an Application class

build passing (https://travis-ci.org/spring-guides/gs-uploading-files)

Get the Code

HTTPS | SSH | Subversion

https://github.com/spring-guid

DOWNLOAD ZIP (HTTPS://GITHUB.COM/SPRING-GUIDES/GS-UPLOADING-FILES/ARCHIVE/MASTER.ZIP)

GO TO REPO (HTTPS://GITHUB.COM/SPRING-GUIDES/GS-UPLOADING-FILES)

Table of contents

To start a Spring Boot MVC application, we first need a starter; here, `spring-boot-starter-thymeleaf` and `spring-boot-starter-web` are already added as dependencies. To upload files with Servlet containers, you need to register a `MultipartConfigElement` class (which would be `<multipart-config>` in web.xml). Thanks to Spring Boot, everything is auto-configured for you!

All you need to get started with this application is the following `Application` class.

`src/main/java/hello/Application.java`

```
package hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

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

}
```

As part of auto-configuring Spring MVC, Spring Boot will create a `MultipartConfigElement` bean and make itself ready for file uploads.

Create a file upload controller

The initial application already contains a few classes to deal with storing and loading the uploaded files on disk; they're all located in the `hello.storage` package. We'll use those in our new `FileUploadController`.

`src/main/java/hello/FileUploadController.java`

```
package hello;

import java.io.IOException;
import java.util.stream.Collectors;

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.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBu
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import hello.storage.StorageFileNotFoundException;
import hello.storage.StorageService;

@Controller
public class FileUploadController {

    private final StorageService storageService;

    @Autowired
    public FileUploadController(StorageService storageService) {
        this.storageService = storageService;
    }

    @GetMapping("/")
    public String listUploadedFiles(Model model) throws IOException {

        model.addAttribute("files", storageService.loadAll().map(
                path -> MvcUriComponentsBuilder.fromMethodName(FileUploadContro
                        "serveFile", path.getFileName().toString()).build().toS
                .collect(Collectors.toList()));

        return "uploadForm";
    }

    @GetMapping("/files/{filename:.+}")
    @ResponseBody
    public ResponseEntity<Resource> serveFile(@PathVariable String filename) {

        Resource file = storageService.loadAsResource(filename);
        return ResponseEntity.ok().header(HttpHeaders.CONTENT_DISPOSITION,
                "attachment; filename=\"" + file.getFilename() + "\"").body(fil
    }

    @PostMapping("/")
    public String handleFileUpload(@RequestParam("file") MultipartFile file,
            RedirectAttributes redirectAttributes) {

        storageService.store(file);
        redirectAttributes.addFlashAttribute("message",
                "You successfully uploaded " + file.getOriginalFilename() + "!"

        return "redirect:/";
    }

    @ExceptionHandler(StorageFileNotFoundException.class)
    public ResponseEntity<?> handleStorageFileNotFound(StorageFileNotFoundExcep
        return ResponseEntity.notFound().build();
```

```
    }
}
```

This class is annotated with `@Controller` so Spring MVC can pick it up and look for routes. Each method is tagged with `@GetMapping` or `@PostMapping` to tie the path and the HTTP action to a particular Controller action.

In this case:

- `GET /` looks up the current list of uploaded files from the `StorageService` and loads it into a Thymeleaf template. It calculates a link to the actual resource using `MvcUriComponentsBuilder`
- `GET /files/{filename}` loads the resource if it exists, and sends it to the browser to download using a `"Content-Disposition"` response header
- `POST /` is geared to handle a multi-part message `file` and give it to the `StorageService` for saving

> In a production scenario, you more likely would store the files in a temporary location, a database, or perhaps a NoSQL store like Mongo's GridFS (http://docs.mongodb.org/manual/core/gridfs). It's is best to NOT load up the file system of your application with content.

You will need to provide a `StorageService` for the controller to interact with a storage layer (e.g. a file system). The interface is like this:

`src/main/java/hello/storage/StorageService.java`

```
package hello.storage;

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

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

public interface StorageService {

    void init();

    void store(MultipartFile file);

    Stream<Path> loadAll();

    Path load(String filename);

    Resource loadAsResource(String filename);

    void deleteAll();

}
```

There is an example implementation of the interface in the sample app. You could copy and paste it if you want to save time.

Creating a simple HTML template

To build something of interest, the following Thymeleaf template is a nice example of uploading files as well as showing what's been uploaded.

`src/main/resources/templates/uploadForm.html`

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

        <div th:if="${message}">
                <h2 th:text="${message}"/>
        </div>

        <div>
                <form method="POST" enctype="multipart/form-data" action="/">
                        <table>
                                <tr><td>File to upload:</td><td><input type="fi
                                <tr><td></td><td><input type="submit" value="Up
                        </table>
                </form>
        </div>

        <div>
                <ul>
                        <li th:each="file : ${files}">
                                <a th:href="${file}" th:text="${file}" />
                        </li>
                </ul>
        </div>

</body>
</html>
```

This template has three parts:

- An optional message at the top where Spring MVC writes a flash-scoped messages (https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#mvc-flash-attributes).
- A form allowing the user to upload files
- A list of files supplied from the backend

Tuning file upload limits

When configuring file uploads, it is often useful to set limits on the size of files. Imagine trying to handle a 5GB file upload! With Spring Boot, we can tune its auto-configured `MultipartConfigElement` with some property settings.

Add the following properties to your existing properties settings:

`src/main/resources/application.properties`

```
spring.servlet.multipart.max-file-size=128KB
spring.servlet.multipart.max-request-size=128KB
```

The multipart settings are constrained as follows:

- `spring.http.multipart.max-file-size` is set to 128KB, meaning total file size cannot exceed 128KB.
- `spring.http.multipart.max-request-size` is set to 128KB, meaning total request size for a `multipart/form-data` cannot exceed 128KB.

Make the application executable

Although it is possible to package this service as a traditional WAR (/understanding/WAR) file for deployment to an external application server, the simpler approach demonstrated below creates a *standalone application*. You package everything in a single, executable JAR file, driven by a good old Java `main()` method. And along the way, you use Spring's support for embedding the Tomcat (/understanding/Tomcat) servlet container as the HTTP runtime,

instead of deploying to an external instance.

You also want a target folder to upload files to, so let's enhance the basic `Application` class and add a Boot `CommandLineRunner` which deletes and re-creates that folder at startup:

`src/main/java/hello/Application.java`

```
package hello;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.EnableConfigurationPropertie
import org.springframework.context.annotation.Bean;

import hello.storage.StorageProperties;
import hello.storage.StorageService;

@SpringBootApplication
@EnableConfigurationProperties(StorageProperties.class)
public class Application {

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

    @Bean
    CommandLineRunner init(StorageService storageService) {
        return (args) -> {
            storageService.deleteAll();
            storageService.init();
        };
    }
}
```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration` tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration` tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- Normally you would add `@EnableWebMvc` for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This flags the application as a web application and activates key behaviors such as setting up a `DispatcherServlet`.
- `@ComponentScan` tells Spring to look for other components, configurations, and services in the `hello` package, allowing it to find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there wasn't a single line of XML? No **web.xml** file either. This web application is 100% pure Java and you didn't have to deal with configuring any plumbing or infrastructure.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you are using Gradle, you can run the application using `./gradlew bootRun`. Or you can build the JAR file using `./gradlew build`. Then you can run the JAR file:

```
java -jar build/libs/gs-uploading-files-0.1.0.jar
```

If you are using Maven, you can run the application using `./mvnw spring-boot:run`. Or you can build the JAR file with `./mvnw clean package`. Then you can run the JAR file:

```
java -jar target/gs-uploading-files-0.1.0.jar
```

> The procedure above will create a runnable JAR. You can also opt to build a classic WAR file (/guides/gs/convert-jar-to-war/) instead.

That runs the server-side piece that receives file uploads. Logging output is displayed. The service should be up and running within a few seconds.

With the server running, you need to open a browser and visit http://localhost:8080/ (http://localhost:8080/) to see the upload form. Pick a (small) file and press "Upload" and you should see the success page from the controller. Choose a file that is too large and you will get an ugly error page.

You should then see something like this in your browser window:

```
You successfully uploaded <name of your file>!
```

Testing your application

There are multiple ways to test this particular feature in our application. Here's one example that leverages `MockMvc`, so it does not require to start the Servlet container:

`src/test/java/hello/FileUploadTests.java`

```
package hello;

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

import org.hamcrest.Matchers;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMoc
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.mock.web.MockMultipartFile;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static org.mockito.BDDMockito.given;
import static org.mockito.BDDMockito.then;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilde
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilde
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers

import hello.storage.StorageFileNotFoundException;
import hello.storage.StorageService;

@RunWith(SpringRunner.class)
@AutoConfigureMockMvc
@SpringBootTest
public class FileUploadTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private StorageService storageService;

    @Test
    public void shouldListAllFiles() throws Exception {
        given(this.storageService.loadAll())
                .willReturn(Stream.of(Paths.get("first.txt"), Paths.get("second

        this.mvc.perform(get("/")).andExpect(status().isOk())
                .andExpect(model().attribute("files",
                        Matchers.contains("http://localhost/files/first.txt",
                                "http://localhost/files/second.txt")));
    }

    @Test
    public void shouldSaveUploadedFile() throws Exception {
        MockMultipartFile multipartFile = new MockMultipartFile("file", "test.t
                "text/plain", "Spring Framework".getBytes());
        this.mvc.perform(fileUpload("/").file(multipartFile))
                .andExpect(status().isFound())
                .andExpect(header().string("Location", "/"));

        then(this.storageService).should().store(multipartFile);
    }

    @SuppressWarnings("unchecked")
    @Test
    public void should404WhenMissingFile() throws Exception {
        given(this.storageService.loadAsResource("test.txt"))
                .willThrow(StorageFileNotFoundException.class);

        this.mvc.perform(get("/files/test.txt")).andExpect(status().isNotFound(
```

```
}
```

In those tests, we're using various mocks to set up the interactions with our Controller and the `StorageService` but also with the Servlet container itself by using `MockMultipartFile`.

For an example of an integration test, please check out the `FileUploadIntegrationTests` class.

Summary

Congratulations! You have just written a web application that uses Spring to handle file uploads.

See Also

The following guides may also be helpful:

- Serving Web Content with Spring MVC (https://spring.io/guides/gs/serving-web-content/)
- Handling Form Submission (https://spring.io/guides/gs/handling-form-submission/)
- Securing a Web Application (https://spring.io/guides/gs/securing-web/)
- Building an Application with Spring Boot (https://spring.io/guides/gs/spring-boot/)

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines (https://github.com/spring-guides/getting-started-guides/wiki).

> All guides are released with an ASLv2 license for the code, and an Attribution, NoDerivatives creative commons license (https://creativecommons.org/licenses/by-nd/3.0/) for the writing.