

# 11. 파일 업로드

#인강/5. 스프링 MVC 2/강의#

## 목차

- 11. 파일 업로드 - 파일 업로드 소개
- 11. 파일 업로드 - 프로젝트 생성
- 11. 파일 업로드 - 서블릿과 파일 업로드1
- 11. 파일 업로드 - 서블릿과 파일 업로드2
- 11. 파일 업로드 - 스프링과 파일 업로드
- 11. 파일 업로드 - 예제로 구현하는 파일 업로드, 다운로드
- 11. 파일 업로드 - 정리

## 파일 업로드 소개

일반적으로 사용하는 HTML Form을 통한 파일 업로드를 이해하려면 먼저 폼을 전송하는 다음 두 가지 방식의 차이를 이해해야 한다.

### HTML 폼 전송 방식

- application/x-www-form-urlencoded
- multipart/form-data

### application/x-www-form-urlencoded 방식

## HTML Form 데이터 전송 POST 전송 - 저장

username:  age:  전송

```
<form action="/save" method="post">
  <input type="text" name="username" />
  <input type="text" name="age" />
  <button type="submit">전송</button>
</form>
```

웹 브라우저가 생성한 요청 HTTP 메시지

```
POST /save HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded
username=kim&age=20
```



application/x-www-form-urlencoded 방식은 HTML 폼 데이터를 서버로 전송하는 가장 기본적인 방법이다. Form 태그에 별도의 enctype 옵션이 없으면 웹 브라우저는 요청 HTTP 메시지의 헤더에 다음

내용을 추가한다.

```
Content-Type: application/x-www-form-urlencoded
```

그리고 폼에 입력한 전송할 항목을 HTTP Body에 문자로 `username=kim&age=20` 와 같이 `&` 로 구분해서 전송한다.

파일을 업로드 하려면 파일은 문자가 아니라 바이너리 데이터를 전송해야 한다. 문자를 전송하는 이 방식으로 파일을 전송하기는 어렵다. 그리고 또 한가지 문제가 더 있는데, 보통 폼을 전송할 때 파일만 전송하는 것이 아니라는 점이다.

다음 예를 보자.

- 이름
- 나이
- 첨부파일

여기에서 이름과 나이도 전송해야 하고, 첨부파일도 함께 전송해야 한다. 문제는 이름과 나이는 문자로 전송하고, 첨부파일은 바이너리로 전송해야 한다는 점이다. 여기에서 문제가 발생한다. **문자와 바이너리를 동시에 전송**해야 하는 상황이다.

이 문제를 해결하기 위해 HTTP는 `multipart/form-data` 라는 전송 방식을 제공한다.

## multipart/form-data 방식

# HTML Form 데이터 전송

## multipart/form-data

username: kim  
age: 20  
file: 파일 선택 intro.png  
전송

```
<form action="/save" method="post" enctype="multipart/form-data">  
  <input type="text" name="username" />  
  <input type="text" name="age" />  
  <input type="file" name="file1" />  
  <button type="submit">전송</button>  
</form>
```

웹 브라우저가 생성한 요청 HTTP 메시지

```
POST /save HTTP/1.1  
Host: localhost:8080  
Content-Type: multipart/form-data; boundary=-----XXX  
Content-Length: 10457  
  
-----XXX  
Content-Disposition: form-data; name="username"  
  
kim  
-----XXX  
Content-Disposition: form-data; name="age"  
  
20  
-----XXX  
Content-Disposition: form-data; name="file1"; filename="intro.png"  
Content-Type: image/png  
  
109238a9o0p3eqwokjasd09ou3oirjwoe9u34ouief...  
-----XXX--
```

끝에는 -- 추가

이 방식을 사용하려면 Form 태그에 별도의 `enctype="multipart/form-data"` 를 지정해야 한다.

multipart/form-data 방식은 다른 종류의 여러 파일과 폼의 내용 함께 전송할 수 있다. (그래서 이름이 multipart 이다.)

폼의 입력 결과로 생성된 HTTP 메시지를 보면 각각의 전송 항목이 구분이 되어있다. Content-Disposition이라는 항목별 헤더가 추가되어 있고 여기에 부가 정보가 있다. 예제에서는 username, age, file1이 각각 분리되어 있고, 폼의 일반 데이터는 각 항목별로 문자가 전송되고, 파일의 경우 파일 이름과 Content-Type이 추가되고 바이너리 데이터가 전송된다.

multipart/form-data는 이렇게 각각의 항목을 구분해서, 한번에 전송하는 것이다.

## Part

multipart/form-data는 application/x-www-form-urlencoded와 비교해서 매우 복잡하고 각각의 부분( Part )로 나누어져 있다. 그렇다면 이렇게 복잡한 HTTP 메시지를 서버에서 어떻게 사용할 수 있을까?

## 참고

multipart/form-data와 폼 데이터 전송에 대한 더 자세한 내용은 모든 개발자를 위한 HTTP 웹 기본 지식 강의를 참고하자.

## 프로젝트 생성

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

<https://start.spring.io>

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.4.x
- Project Metadata
  - Group: hello
  - Artifact: upload
  - Name: upload
  - Package name: **hello.upload**
  - Packaging: **Jar**
  - Java: 11

- Dependencies: **Spring Web, Lombok , Thymeleaf**

## build.gradle

```
plugins {  
    id 'org.springframework.boot' version '2.4.5'  
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'  
    id 'java'  
}  
  
group = 'hello'  
version = '0.0.1-SNAPSHOT'  
sourceCompatibility = '11'  
  
configurations {  
    compileOnly {  
        extendsFrom annotationProcessor  
    }  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    compileOnly 'org.projectlombok:lombok'  
    annotationProcessor 'org.projectlombok:lombok'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}  
  
test {  
    useJUnitPlatform()  
}
```

- 동작 확인
  - 기본 메인 클래스 실행( `UploadApplication.main()` )
  - <http://localhost:8080> 호출해서 Whitelabel Error Page가 나오면 정상 동작

편의상 `index.html` 을 추가해두자.

`resources/static/index.html`

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<ul>
  <li>상품 관리
    <ul>
      <li><a href="/servlet/v1/upload">서블릿 파일 업로드1</a></li>
      <li><a href="/servlet/v2/upload">서블릿 파일 업로드2</a></li>
      <li><a href="/spring/upload">스프링 파일 업로드</a></li>
      <li><a href="/items/new">상품 - 파일, 이미지 업로드</a></li>
    </ul>
  </li>
</ul>
</body>
</html>
```

## 서블릿과 파일 업로드1

먼저 서블릿을 통한 파일 업로드를 코드와 함께 알아보자.

### ServletUploadControllerV1

```
package hello.upload.controller;
```

```

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.Part;
import java.io.IOException;
import java.util.Collection;

@Slf4j
@Controller
@RequestMapping("/servlet/v1")
public class ServletUploadControllerV1 {

    @GetMapping("/upload")
    public String newFile() {
        return "upload-form";
    }

    @PostMapping("/upload")
    public String saveFileV1(HttpServletRequest request) throws
ServletException, IOException {
        log.info("request={}", request);

        String itemName = request.getParameter("itemName");
        log.info("itemName={}", itemName);

        Collection<Part> parts = request.getParts();
        log.info("parts={}", parts);

        return "upload-form";
    }
}

```

`request.getParts()` : `multipart/form-data` 전송 방식에서 각각 나누어진 부분을 받아서 확인할 수 있다.

`resources/templates/upload-form.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

<div class="container">

    <div class="py-5 text-center">
        <h2>상품 등록 폼</h2>
    </div>

    <h4 class="mb-3">상품 입력</h4>

    <form th:action method="post" enctype="multipart/form-data">
        <ul>
            <li>상품명 <input type="text" name="itemName"></li>
            <li>파일<input type="file" name="file" ></li>
        </ul>
        <input type="submit"/>
    </form>

</div> <!-- /container -->
</body>
</html>
```

테스트를 진행하기 전에 먼저 다음 옵션들을 추가하자.

`application.properties`

```
logging.level.org.apache.coyote.http11=debug
```

이 옵션을 사용하면 HTTP 요청 메시지를 확인할 수 있다.

## 실행

<http://localhost:8080/servlet/v1/upload>

실행해보면 `logging.level.org.apache.coyote.http11` 옵션을 통한 로그에서 `multipart/form-data` 방식으로 전송된 것을 확인할 수 있다.

## 결과 로그

```
Content-Type: multipart/form-data; boundary=-----xxxx

-----xxxx
Content-Disposition: form-data; name="itemName"

Spring
-----xxxx
Content-Disposition: form-data; name="file"; filename="test.data"
Content-Type: application/octet-stream

sdklajkljdf...
```

## 멀티파트 사용 옵션

### 업로드 사이즈 제한

```
spring.servlet.multipart.max-file-size=1MB
spring.servlet.multipart.max-request-size=10MB
```

큰 파일을 무제한 업로드하게 둘 수는 없으므로 업로드 사이즈를 제한할 수 있다.

사이즈를 넘으면 예외(`SizeLimitExceededException`)가 발생한다.

`max-file-size`: 파일 하나의 최대 사이즈, 기본 1MB



`max-request-size`: 멀티파트 요청 하나에 여러 파일을 업로드 할 수 있는데, 그 전체 합이다. 기본 10MB

### spring.servlet.multipart.enabled 끄기

```
spring.servlet.multipart.enabled=false
```

#### 결과 로그

```
request=org.apache.catalina.connector.RequestFacade@xxx  
itemName=null  
parts=[]
```

멀티파트는 일반적인 폼 요청인 `application/x-www-form-urlencoded` 보다 훨씬 복잡하다.

`spring.servlet.multipart.enabled` 옵션을 끄면 서블릿 컨테이너는 멀티파트와 관련된 처리를 하지 않는다.

그래서 결과 로그를 보면 `request.getParameter("itemName")`, `request.getParts()` 의 결과가 비어있다.

### spring.servlet.multipart.enabled 켜기

```
spring.servlet.multipart.enabled=true (기본 true)
```

이 옵션을 켜면 스프링 부트는 서블릿 컨테이너에게 멀티파트 데이터를 처리하라고 설정한다. 참고로 기본 값은 `true` 이다.

```
request=org.springframework.web.multipart.support.StandardMultipartHttpServletRequest  
itemName=Spring  
parts=[ApplicationPart1, ApplicationPart2]
```

`request.getParameter("itemName")` 의 결과도 잘 출력되고, `request.getParts()` 에도 요청한 두 가지 멀티파트의 부분 데이터가 포함된 것을 확인할 수 있다. 이 옵션을 켜면 복잡한 멀티파트 요청을 처리해서 사용할 수 있게 제공한다.

로그를 보면 `HttpServletRequest` 객체가 `RequestFacade` → `StandardMultipartHttpServletRequest` 로 변한 것을 확인할 수 있다.

## 참고

`spring.servlet.multipart.enabled` 옵션을 켜면 스프링의 `DispatcherServlet` 에서 멀티파트 리졸버(`MultipartResolver`)를 실행한다.

멀티파트 리졸버는 멀티파트 요청인 경우 서블릿 컨테이너가 전달하는 일반적인 `HttpServletRequest` 를 `MultipartHttpServletRequest` 로 변환해서 반환한다.

`MultipartHttpServletRequest` 는 `HttpServletRequest` 의 자식 인터페이스이고, 멀티파트와 관련된 추가 기능을 제공한다.

스프링이 제공하는 기본 멀티파트 리졸버는 `MultipartHttpServletRequest` 인터페이스를 구현한 `StandardMultipartHttpServletRequest` 를 반환한다.

이제 컨트롤러에서 `HttpServletRequest` 대신에 `MultipartHttpServletRequest` 를 주입받을 수 있는데, 이것을 사용하면 멀티파트와 관련된 여러가지 처리를 편리하게 할 수 있다. 그런데 이후 강의에서 설명할 `MultipartFile` 이라는 것을 사용하는 것이 더 편하기 때문에 `MultipartHttpServletRequest` 를 잘 사용하지는 않는다. 더 자세한 내용은 `MultipartResolver` 를 검색해보자.

## 서블릿과 파일 업로드2

서블릿이 제공하는 `Part` 에 대해 알아보고 실제 파일도 서버에 업로드 해보자.

먼저 파일을 업로드를 하려면 실제 파일이 저장되는 경로가 필요하다.

해당 경로에 실제 폴더를 만들어두자.

그리고 다음에 만들어진 경로를 입력해두자.

### application.properties

```
file.dir=파일 업로드 경로 설정(예): /Users/kimyoungha/study/file/
```

### 주의

1. 꼭 해당 경로에 실제 폴더를 미리 만들어두자.
2. `application.properties` 에서 설정할 때 마지막에 `/` (슬래시)가 포함된 것에 주의하자.

## ServletUploadControllerV2

```
package hello.upload.controller;
```

```

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Controller;
import org.springframework.util.StreamUtils;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.Part;
import java.io.IOException;
import java.io.InputStream;
import java.nio.charset.StandardCharsets;
import java.util.Collection;

@Slf4j
@Controller
@RequestMapping("/servlet/v2")
public class ServletUploadControllerV2 {

    @Value("${file.dir}")
    private String fileDir;

    @GetMapping("/upload")
    public String newFile() {
        return "upload-form";
    }

    @PostMapping("/upload")
    public String saveFileV1(HttpServletRequest request) throws
ServletException, IOException {
        log.info("request={}", request);

        String itemName = request.getParameter("itemName");
        log.info("itemName={}", itemName);

```

```

Collection<Part> parts = request.getParts();
log.info("parts={}", parts);
for (Part part : parts) {

    log.info("==== PART ====");
    log.info("name={}", part.getName());
    Collection<String> headerNames = part.getHeaderNames();
    for (String headerName : headerNames) {
        log.info("header {}: {}", headerName,
part.getHeader(headerName));
    }
    //편의 메서드
    //content-disposition; filename
    log.info("submittedFileName={}", part.getSubmittedFileName());
    log.info("size={}", part.getSize()); //part body size

    //데이터 읽기
    InputStream inputStream = part.getInputStream();
    String body = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);
    log.info("body={}", body);

    //파일에 저장하기
    if (StringUtils.hasText(part.getSubmittedFileName())) {
        String fullPath = fileDir + part.getSubmittedFileName();
        log.info("파일 저장 fullPath={}", fullPath);
        part.write(fullPath);
    }
}

return "upload-form";
}
}

```

```
@Value("${file.dir}")
private String fileDir;
```

application.properties에서 설정한 file.dir의 값을 주입한다.

멀티파트 형식은 전송 데이터를 하나하나 각각 부분( Part )으로 나누어 전송한다. parts에는 이렇게 나누어진 데이터가 각각 담긴다.

서블릿이 제공하는 Part는 멀티파트 형식을 편리하게 읽을 수 있는 다양한 메서드를 제공한다.

### Part 주요 메서드

part.getSubmittedFileName(): 클라이언트가 전달한 파일명

part.getInputStream(): Part의 전송 데이터를 읽을 수 있다.

part.write(...): Part를 통해 전송된 데이터를 저장할 수 있다.

### 실행

<http://localhost:8080/servlet/v2/upload>

다음 내용을 전송했다.

- itemName: 상품A
- file: 스크린샷.png

### 결과 로그

```
==== PART ====
name=itemName
header content-disposition: form-data; name="itemName"
submittedFileName=null
size=7
body=상품A
==== PART ====
name=file
header content-disposition: form-data; name="file"; filename="스크린샷.png"
header content-type: image/png
submittedFileName=스크린샷.png
size=112384
body=qwlkjek2ljlese...
파일 저장 fullPath=/Users/kimyoungha/study/file/스크린샷.png
```

파일 저장 경로에 가보면 실제 파일이 저장된 것을 확인할 수 있다. 만약 저장이 되지 않았다면 파일 저장 경로를 다시 확인하자.

## 참고

큰 용량의 파일을 업로드를 테스트 할 때는 로그가 너무 많이 남아서 다음 옵션을 끄는 것이 좋다.

```
logging.level.org.apache.coyote.http11=debug
```

다음 부분도 파일의 바이너리 데이터를 모두 출력하므로 끄는 것이 좋다.

```
log.info("body={}", body);
```

서블릿이 제공하는 `Part` 는 편하기는 하지만, `HttpServletRequest` 를 사용해야 하고, 추가로 파일 부분만 구분하려면 여러가지 코드를 넣어야 한다. 이번에는 스프링이 이 부분을 얼마나 편리하게 제공하는지 확인해보자.

## 스프링과 파일 업로드

스프링은 `MultipartFile` 이라는 인터페이스로 멀티파트 파일을 매우 편리하게 지원한다.

### SpringUploadController

```
package hello.upload.controller;

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
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 javax.servlet.http.HttpServletRequest;
import java.io.File;
import java.io.IOException;

@Slf4j
```

```

@Controller
@RequestMapping("/spring")
public class SpringUploadController {

    @Value("${file.dir}")
    private String fileDir;

    @GetMapping("/upload")
    public String newFile() {
        return "upload-form";
    }

    @PostMapping("/upload")
    public String saveFile(@RequestParam String itemName,
                           @RequestParam MultipartFile file, HttpServletRequest
request) throws IOException {

        log.info("request={}", request);
        log.info("itemName={}", itemName);
        log.info("multipartFile={}", file);

        if (!file.isEmpty()) {
            String fullPath = fileDir + file.getOriginalFilename();
            log.info("파일 저장 fullPath={}", fullPath);
            file.transferTo(new File(fullPath));
        }

        return "upload-form";
    }
}

```

코드를 보면 스프링 답게 딱 필요한 부분의 코드만 작성하면 된다.

`@RequestParam MultipartFile file`

업로드하는 HTML Form의 name에 맞추어 `@RequestParam`을 적용하면 된다. 추가로

`@ModelAttribute`에서도 `MultipartFile`을 동일하게 사용할 수 있다.

## MultipartFile 주요 메서드

`file.getOriginalFilename()` : 업로드 파일 명

`file.transferTo(...)` : 파일 저장

## 실행

<http://localhost:8080/spring/upload>

## 실행 로그

```
request=org.springframework.web.multipart.support.StandardMultipartHttpServletRequest@5c022dc6
itemName=상품A
multipartFile=org.springframework.web.multipart.support.StandardMultipartHttpServletRequest$StandardMultipartFile@274ba730
파일 저장 fullPath=/Users/kimyoungha/study/file/스크린샷.png
```

## 예제로 구현하는 파일 업로드, 다운로드

실제 파일이나 이미지를 업로드, 다운로드 할 때는 몇가지 고려할 점이 있는데, 구체적인 예제로 알아보자.

### 요구사항

- 상품을 관리
  - 상품 이름
  - 첨부파일 하나
  - 이미지 파일 여러개
- 첨부파일을 업로드 다운로드 할 수 있다.
- 업로드한 이미지를 웹 브라우저에서 확인할 수 있다.

### Item - 상품 도메인

```
package hello.upload.domain;

import lombok.Data;
```



```
import java.util.List;

@Data
public class Item {

    private Long id;
    private String itemName;
    private UploadFile attachFile;
    private List<UploadFile> imageFiles;
}
```

### ItemRepository - 상품 리포지토리

```
package hello.upload.domain;

import org.springframework.stereotype.Repository;
import java.util.HashMap;
import java.util.Map;

@Repository
public class ItemRepository {

    private final Map<Long, Item> store = new HashMap<>();
    private long sequence = 0L;

    public Item save(Item item) {
        item.setId(++sequence);
        store.put(item.getId(), item);
        return item;
    }

    public Item findById(Long id) {
        return store.get(id);
    }
}
```

## UploadFile - 업로드 파일 정보 보관

```
package hello.upload.domain;

import lombok.Data;

@Data
public class UploadFile {

    private String uploadFileName;
    private String storeFileName;

    public UploadFile(String uploadFileName, String storeFileName) {
        this.uploadFileName = uploadFileName;
        this.storeFileName = storeFileName;
    }
}
```

uploadFileName : 고객이 업로드한 파일명

storeFileName : 서버 내부에서 관리하는 파일명

고객이 업로드한 파일명으로 서버 내부에 파일을 저장하면 안된다. 왜냐하면 서로 다른 고객이 같은 파일이름을 업로드 하는 경우 기존 파일 이름과 충돌이 날 수 있다. 서버에서는 저장할 파일명이 겹치지 않도록 내부에서 관리하는 별도의 파일명이 필요하다.

## FileStore - 파일 저장과 관련된 업무 처리

```
package hello.upload.file;

import hello.upload.domain.UploadFile;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.multipart.MultipartFile;
```

```

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

@Component
public class FileStore {

    @Value("${file.dir}")
    private String fileDir;

    public String getFullPath(String filename) {
        return fileDir + filename;
    }

    public List<UploadFile> storeFiles(List<MultipartFile> multipartFiles)
throws IOException {
        List<UploadFile> storeFileResult = new ArrayList<>();
        for (MultipartFile multipartFile : multipartFiles) {
            if (!multipartFile.isEmpty()) {
                storeFileResult.add(storeFile(multipartFile));
            }
        }
        return storeFileResult;
    }

    public UploadFile storeFile(MultipartFile multipartFile) throws IOException
{
        if (multipartFile.isEmpty()) {
            return null;
        }

        String originalFilename = multipartFile.getOriginalFilename();
        String storeFileName = createStoreFileName(originalFilename);
        multipartFile.transferTo(new File(getFullPath(storeFileName)));
        return new UploadFile(originalFilename, storeFileName);
    }
}

```

```

private String createStoreFileName(String originalFilename) {
    String ext = extractExt(originalFilename);
    String uuid = UUID.randomUUID().toString();
    return uuid + "." + ext;
}

private String extractExt(String originalFilename) {
    int pos = originalFilename.lastIndexOf(".");
    return originalFilename.substring(pos + 1);
}
}

```

멀티파트 파일을 서버에 저장하는 역할을 담당한다.

- `createStoreFileName()` : 서버 내부에서 관리하는 파일명은 유일한 이름을 생성하는 `UUID` 를 사용해서 충돌하지 않도록 한다.
- `extractExt()` : 확장자를 별도로 추출해서 서버 내부에서 관리하는 파일명에도 붙여준다. 예를 들어서 고객이 `a.png` 라는 이름으로 업로드 하면 `51041c62-86e4-4274-801d-614a7d994edb.png` 와 같이 저장한다.

## ItemForm

```

package hello.upload.controller;

import lombok.Data;
import org.springframework.web.multipart.MultipartFile;

import java.util.List;

@Data
public class ItemForm {
    private Long itemId;
    private String itemName;
    private List<MultipartFile> imageFiles;
    private MultipartFile attachFile;
}

```

상품 저장용 폼이다.

`List<MultipartFile> imageFiles`: 이미지를 다중 업로드 하기 위해 `MultipartFile` 를 사용했다.  
`MultipartFile attachFile`: 멀티파트는 `@ModelAttribute` 에서 사용할 수 있다.

## ItemController

```
package hello.upload.controller;

import hello.upload.domain.UploadFile;
import hello.upload.domain.Item;
import hello.upload.domain.ItemRepository;
import hello.upload.file.FileStore;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.io.*;
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.*;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.springframework.web.util.UriUtils;

import java.io.IOException;
import java.net.MalformedURLException;
import java.nio.charset.StandardCharsets;
import java.util.List;

@Slf4j
@Controller
@RequiredArgsConstructor
public class ItemController {

    private final ItemRepository itemRepository;
    private final FileStore fileStore;

    @GetMapping("/items/new")
    public String newItem(@ModelAttribute ItemForm form) {
        return "item-form";
    }
}
```

```

@PostMapping("/items/new")
public String saveItem(@ModelAttribute ItemForm form, RedirectAttributes
redirectAttributes) throws IOException {

    UploadFile attachFile = fileStore.storeFile(form.getAttachFile());
    List<UploadFile> storeImageFiles =
fileStore.storeFiles(form.getImageFiles());

    //데이터베이스에 저장
    Item item = new Item();
    item.setItemName(form.getItemName());
    item.setAttachFile(attachFile);
    item.setImageFiles(storeImageFiles);
    itemRepository.save(item);

    redirectAttributes.addAttribute("itemId", item.getId());

    return "redirect:/items/{itemId}";
}

@GetMapping("/items/{id}")
public String items(@PathVariable Long id, Model model) {
    Item item = itemRepository.findById(id);
    model.addAttribute("item", item);
    return "item-view";
}

@ResponseBody
@GetMapping("/images/{filename}")
public Resource downloadImage(@PathVariable String filename) throws
MalformedURLException {
    return new UrlResource("file:" + fileStore.getFullPath(filename));
}

@GetMapping("/attach/{itemId}")
public ResponseEntity<Resource> downloadAttach(@PathVariable Long itemId)
throws MalformedURLException {

```

```

        Item item = itemRepository.findById(itemId);
        String storeFileName = item.getAttachFile().getStoreFileName();
        String uploadFileName = item.getAttachFile().getUploadFileName();

        UrlResource resource = new UrlResource("file:" +
fileStore.getFullPath(storeFileName));

        log.info("uploadFileName={}", uploadFileName);
        String encodedUploadFileName = UriUtils.encode(uploadFileName,
StandardCharsets.UTF_8);
        String contentDisposition = "attachment; filename=\"" +
encodedUploadFileName + "\"";
        return ResponseEntity.ok()
                .header(HttpHeaders.CONTENT_DISPOSITION, contentDisposition)
                .body(resource);
    }
}

```

- `@GetMapping("/items/new")` : 등록 폼을 보여준다.
- `@PostMapping("/items/new")` : 품의 데이터를 저장하고 보여주는 화면으로 리다이렉트 한다.
- `@GetMapping("/items/{id}")` : 상품을 보여준다.
- `@GetMapping("/images/{filename}")` : `<img>` 태그로 이미지를 조회할 때 사용한다. `UrlResource` 로 이미지 파일을 읽어서 `@ResponseBody` 로 이미지 바이너리를 반환한다.
- `@GetMapping("/attach/{itemId}")` : 파일을 다운로드 할 때 실행한다. 예제를 더 단순화 할 수 있지만, 파일 다운로드 시 권한 체크같은 복잡한 상황까지 가정한다 생각하고 이미지 `id` 를 요청하도록 했다. 파일 다운로드시에는 고객이 업로드한 파일 이름으로 다운로드 하는게 좋다. 이때는 `Content-Disposition` 헤더에 `attachment; filename="업로드 파일명"` 값을 주면 된다.

## 등록 폼 뷰

resources/templates/item-form.html

```

<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

```

```

<div class="container">

    <div class="py-5 text-center">
        <h2>상품 등록</h2>
    </div>

    <form th:action method="post" enctype="multipart/form-data">
        <ul>
            <li>상품명 <input type="text" name="itemName"></li>
            <li>첨부파일<input type="file" name="attachFile" ></li>
            <li>이미지 파일들<input type="file" multiple="multiple"
name="imageFiles" ></li>
        </ul>
        <input type="submit"/>
    </form>

</div> <!-- /container -->
</body>
</html>

```

다중 파일 업로드를 하려면 `multiple="multiple"` 옵션을 주면 된다.

`ItemForm` 의 다음 코드에서 여러 이미지 파일을 받을 수 있다.

```
private List<MultipartFile> imageFiles;
```

## 조회 뷰

`resources/templates/item-view.html`

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

    <div class="container">

        <div class="py-5 text-center">

```



```

    <h2>상품 조회</h2>

</div>

상품명: <span th:text="${item.itemName}">상품명</span><br/>
첨부파일: <a th:href="${item.attachFile}" th:href="/attach/${item.id}"
th:text="${item.getAttachFile().getUploadFileName()}" /><br/>
    

</div> <!-- /container -->
</body>
</html>

```

첨부 파일은 링크로 걸어두고, 이미지는 `<img>` 태그를 반복해서 출력한다.

## 실행

<http://localhost:8080/items/new>

실행해보면 하나의 첨부파일을 다운로드 업로드 하고, 여러 이미지 파일을 한번에 업로드 할 수 있다.

## 정리