10. 파일 업로드 및 다운로드

MultipartFile 처리

Spring Boot에서 `MultipartFile`을 사용하면 **파일 업로드 요청을 간단하게 처리**할 수 있습니다.
사용자는 HTML `<form>`이나 REST API를 통해 파일을 전송하고,
서버는 `MultipartFile` 객체로 해당 파일을 받고 **저장, 변환, 처리**할 수 있습니다.

아래는 `MultipartFile`의 **기본 개념, 요청 처리, 저장 전략, 실무 예시, 주의사항**까지 전부 정리한 실무 가이드입니다.

✅ 1. Multipart란?

> HTTP 요청에서 `Content-Type: multipart/form-data` 형식으로
> 텍스트 + 파일 등 여러 종류의 데이터를 **한 번에 전송**하는 방식입니다.

✅ 2. 기본 구조 및 의존성

Spring Boot는 기본적으로 `spring-boot-starter-web`에 파일 업로드 관련 처리를 포함하고 있어
별도의 의존성 추가가 필요 없습니다.

✅ 3. application.yml 설정

```yaml
spring:
  servlet:
    multipart:
      enabled: true
      max-file-size: 10MB
      max-request-size: 20MB
```

- `max-file-size`: 단일 파일 최대 크기
- `max-request-size`: 요청 전체 크기 제한 (여러 파일 포함 시)

✅ 4. 컨트롤러에서 파일 받기

🔹 단일 파일

```
@PostMapping("/upload")
public ResponseEntity<String> upload(@RequestParam("file") MultipartFile file) throws
IOException {
    String originalName = file.getOriginalFilename();
    long size = file.getSize();
    byte[] content = file.getBytes(); // 또는 file.getInputStream()

    // 저장, 처리 등 수행

```

```
    return ResponseEntity.ok("파일 업로드 성공: " + originalName);
}
```

🔹 여러 파일

```
@PostMapping("/upload-multiple")
public ResponseEntity<?> uploadMultiple(@RequestParam("files") List<MultipartFile>
files) {
    for (MultipartFile file : files) {
        // 파일별 처리
    }
    return ResponseEntity.ok("업로드 완료");
}
```

🔹 DTO로 받기

```
@Data
public class UploadRequest {
    private String title;
    private MultipartFile file;
}
```

→ `@ModelAttribute`를 활용하여 함께 받기 가능 (단, `@RequestBody`와는 호환되지 않음)

✅ 5. 파일 저장 예시

```
public void saveFile(MultipartFile file, String uploadDir) throws IOException {
    String filename = UUID.randomUUID() + "_" + file.getOriginalFilename();
    Path path = Paths.get(uploadDir + File.separator + filename);
    Files.createDirectories(path.getParent());
    file.transferTo(path);
}
```

- `transferTo()`는 로컬 파일 시스템에 직접 저장
- 저장 전에 `파일 이름 인코딩, 확장자 검사, 크기 검증` 필요

✅ 6. 클라이언트 요청 예시 (HTML)

```
<form method="post" action="/upload" enctype="multipart/form-data">
  <input type="file" name="file">
  <input type="submit" value="업로드">
</form>
```

✅ 7. 클라이언트 요청 예시 (Postman)

- `POST /upload`
- `Body → form-data → key=file (type: file)` → 파일 선택

✅ 8. 실무 시 고려사항

항목	설명
파일 크기 제한	`application.yml` 설정 또는 컨트롤러에서 확인
파일 이름 정규화	UUID, timestamp 등으로 변경 권장
저장 위치 분리	OS 별 디렉터리 구조 주의 (ex: `/tmp`, `/uploads`)
확장자 검증	`.jpg`, `.pdf` 등 허용 확장자만 허용
보안 위험	`.jsp`, `.exe`, `.sh` 등 실행 파일은 저장하지 않도록 필터링
임시 파일 처리	서버 재시작 시 임시파일 자동 삭제됨 (`/tmp`)

✅ 9. 파일 다운로드 처리 예

```
@GetMapping("/download/{filename}")
public ResponseEntity<Resource> download(@PathVariable String filename) throws
IOException {
    Path path = Paths.get("uploads").resolve(filename).normalize();
    Resource resource = new UrlResource(path.toUri());

    return ResponseEntity.ok()
        .contentType(MediaType.APPLICATION_OCTET_STREAM)
        .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" +
resource.getFilename() + "\"")
        .body(resource);
}
```

✅ 10. 결론 요약

항목	설명
파일 수신 방식	`@RequestParam MultipartFile file`
여러 파일	`List<MultipartFile>` 처리
DTO 수신	`@ModelAttribute` 이용
파일 저장	`file.transferTo(Path)`

항목	설명
설정 관리	`spring.servlet.multipart.*`
보안 필터링	확장자, 경로, 크기 제한 등 강력히 검증 필요

대용량 파일 스트리밍

Spring Boot에서 대용량 파일을 처리할 때는 일반적인 `MultipartFile` 방식으로 메모리에 올리는 것보다,
스트리밍(Streaming)을 사용하여 **메모리 부담 없이** 데이터를 주고받는 것이 중요합니다.
특히 **수십~수백 MB 이상의 파일 업로드/다운로드**에서는 **InputStream/OutputStream 기반 처리**가 핵심입니다.

아래는 **대용량 파일의 업로드 및 다운로드 스트리밍 처리 전략**을 정리한 실무 가이드입니다.

✅ 1. 스트리밍이란?

> 파일 전체를 메모리에 적재하지 않고, **일정 크기씩 나누어 순차적으로 읽고/쓰기** 하는 방식.
> `InputStream`/`OutputStream` 기반으로 처리하며, **메모리 사용량을 일정하게 유지**한다.

✅ 2. 대용량 다운로드 스트리밍 (`OutputStream` 활용)

```
@GetMapping("/download/{filename}")
public void downloadFile(@PathVariable String filename, HttpServletResponse response)
throws IOException {
    File file = new File("files/" + filename);
    response.setContentType("application/octet-stream");
    response.setHeader("Content-Disposition", "attachment; filename=\"" +
file.getName() + "\"");
    response.setContentLengthLong(file.length());

    try (InputStream in = new FileInputStream(file);
         OutputStream out = response.getOutputStream()) {

        byte[] buffer = new byte[8192];
        int bytesRead;

        while ((bytesRead = in.read(buffer)) != -1) {
            out.write(buffer, 0, bytesRead);
        }
    }
}
```

🔸 특징

- 브라우저에 직접 스트리밍 전송
- 메모리 사용 최소화
- `application/octet-stream`으로 바이너리 전송

✅ 3. 대용량 업로드 스트리밍 (`InputStream` 활용)

```
@PostMapping("/upload")
public ResponseEntity<String> upload(HttpServletRequest request) throws IOException {
    String uploadPath = "uploads/large-file.dat";
    try (InputStream in = request.getInputStream();
         OutputStream out = new FileOutputStream(uploadPath)) {

        byte[] buffer = new byte[8192];
        int bytesRead;

        while ((bytesRead = in.read(buffer)) != -1) {
            out.write(buffer, 0, bytesRead);
        }
    }

    return ResponseEntity.ok("업로드 완료");
}
```

🔸 특징

- Multipart 형태가 아닌 **raw binary** 요청에 적합
- 프론트엔드에서는 `Content-Type: application/octet-stream`으로 전송 필요
- 실무에서는 `StreamingHttpMessageConverter`, `StreamingResponseBody` 사용 가능

✅ 4. MultipartFile로 스트리밍 저장 (대용량도 가능하나 제한 있음)

```
@PostMapping("/upload-multipart")
public ResponseEntity<?> upload(@RequestParam MultipartFile file) throws IOException {
    try (InputStream in = file.getInputStream();
         OutputStream out = new FileOutputStream("uploads/" +
file.getOriginalFilename())) {

        byte[] buffer = new byte[8192];
        int bytesRead;
        while ((bytesRead = in.read(buffer)) != -1) {
            out.write(buffer, 0, bytesRead);
        }
    }
    return ResponseEntity.ok("저장 완료");
}
```

🔸 주의사항

- MultipartFile은 메모리에 올라가는 한계 설정이 `application.yml`에 따라 제한됨
- 10MB 이상은 `DiskFileItem`으로 임시 파일 처리됨

✅ 5. `StreamingResponseBody` 활용 (Spring 전용 스트리밍)

```
@GetMapping("/stream/{filename}")
public ResponseEntity<StreamingResponseBody> stream(@PathVariable String filename)
throws IOException {
    File file = new File("files/" + filename);

    StreamingResponseBody stream = outputStream -> {
        try (InputStream in = new FileInputStream(file)) {
            byte[] buffer = new byte[8192];
            int bytesRead;
            while ((bytesRead = in.read(buffer)) != -1) {
                outputStream.write(buffer, 0, bytesRead);
            }
        }
    };

    return ResponseEntity.ok()
        .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=" +
file.getName())
        .contentType(MediaType.APPLICATION_OCTET_STREAM)
        .body(stream);
}
```

🔸 장점

- Spring MVC 비동기 지원 (`AsyncContext`)
- 요청 스레드를 블로킹하지 않음
- **실시간 처리 / 대규모 병렬 다운로드**에 적합

✅ 6. Spring WebFlux (Reactive 기반 스트리밍)

> Spring WebFlux 환경에서는 `Flux<DataBuffer>` 기반으로 완전 비동기 스트리밍 가능

```
@GetMapping(value = "/download", produces = MediaType.APPLICATION_OCTET_STREAM_VALUE)
public Mono<ResponseEntity<Flux<DataBuffer>>> download() {
    File file = new File("files/large.dat");
    Flux<DataBuffer> flux = DataBufferUtils.read(file.toPath(), new
DefaultDataBufferFactory(), 4096);
    return Mono.just(ResponseEntity.ok(flux));
}
```

✅ 7. 파일 업로드 제한 설정

```
spring:
  servlet:
    multipart:
      max-file-size: 2GB
      max-request-size: 2GB
```

- Tomcat 레벨 설정도 필요할 수 있음 (server.tomcat.max-swallow-size)

✅ 8. 실무 팁 요약

목적	방법
다운로드 스트리밍	`OutputStream`, `StreamingResponseBody`
업로드 스트리밍	`HttpServletRequest.getInputStream()`
대용량 업로드/다운로드	Chunked 방식, 병렬 분할 업로드 고려
메모리 최소화	`byte[] buffer` 기반 처리
다국어 파일 이름	`URLEncoder.encode()`로 처리
보안	파일 이름 검증, 경로 검증 (경로 traversal 방지) 필수

✅ 결론 요약

항목	설명
기본 전략	InputStream / OutputStream 기반 처리
Spring 전용	`StreamingResponseBody` (MVC), `Flux<DataBuffer>` (WebFlux)
Multipart 사용 시	10MB 이상이면 디스크 저장 → 스트리밍 유사
장점	메모리 사용 제한, 실시간 처리 가능
실무 활용	영상/로그 다운로드, 대형 이미지 업로드, 파일 백업 API 등에 적합

S3, Cloud Storage 연동

Spring Boot에서 **Amazon S3** 또는 **Google Cloud Storage (GCS)** 와 같은 클라우드 스토리지를 연동하면,
파일을 **안전하게 업로드/다운로드/삭제/리스트화** 할 수 있습니다.
이러한 연동은 로컬 저장소보다 확장성과 접근성이 뛰어나 **실무에서 매우 널리 사용됩니다.**

아래는 **S3와 GCS 연동을 각각 따로 정리**한 실전 가이드입니다.

✅ 1. Amazon S3 연동

📌 의존성 추가 (Gradle)

```
implementation 'software.amazon.awssdk:s3'
```

> AWS SDK v2 사용 (v1보다 경량, 비동기 지원 우수)

📌 application.yml 설정

```
cloud:
  aws:
    s3:
      bucket: my-bucket-name
    credentials:
      access-key: YOUR_ACCESS_KEY
      secret-key: YOUR_SECRET_KEY
    region:
      static: ap-northeast-2
```

📌 S3Client Bean 등록

```
@Configuration
public class S3Config {

    @Value("${cloud.aws.region.static}")
    private String region;

    @Value("${cloud.aws.credentials.access-key}")
    private String accessKey;

    @Value("${cloud.aws.credentials.secret-key}")
    private String secretKey;

    @Bean
    public S3Client s3Client() {
        AwsBasicCredentials credentials = AwsBasicCredentials.create(accessKey,
secretKey);

        return S3Client.builder()
            .credentialsProvider(StaticCredentialsProvider.create(credentials))
            .region(Region.of(region))
            .build();
    }
}
```

📌 업로드 예제

```
public void uploadFile(String key, MultipartFile file) throws IOException {
    PutObjectRequest request = PutObjectRequest.builder()
        .bucket("my-bucket-name")
        .key(key)
        .contentType(file.getContentType())
        .build();

    s3Client.putObject(request, RequestBody.fromInputStream(file.getInputStream(),
file.getSize()));
}
```

📌 다운로드 예제

```
public byte[] downloadFile(String key) {
    GetObjectRequest request = GetObjectRequest.builder()
        .bucket("my-bucket-name")
        .key(key)
        .build();

    ResponseBytes<GetObjectResponse> objectBytes = s3Client.getObjectAsBytes(request);
    return objectBytes.asByteArray();
}
```

📌 삭제 예제

```
public void deleteFile(String key) {
    DeleteObjectRequest request = DeleteObjectRequest.builder()
        .bucket("my-bucket-name")
        .key(key)
        .build();
    s3Client.deleteObject(request);
}
```

✅ 2. Google Cloud Storage (GCS) 연동

📌 의존성 추가

```
implementation 'com.google.cloud:google-cloud-storage:2.33.0'
```

📌 서비스 계정 키 설정

1. GCP 콘솔 → IAM & 관리자 → 서비스 계정 생성
2. 키를 JSON으로 다운로드
3. `GOOGLE_APPLICATION_CREDENTIALS` 환경변수로 경로 지정 또는 직접 로딩

📌 application.yml 예시

```yaml
gcs:
  bucket: my-gcs-bucket
  credentials: /path/to/key.json
```

📌 GCSClient 등록

```java
@Configuration
public class GCSConfig {

    @Value("${gcs.credentials}")
    private String credentialsPath;

    @Bean
    public Storage gcsStorage() throws IOException {
        GoogleCredentials credentials = GoogleCredentials.fromStream(new FileInputStream(credentialsPath));
        return StorageOptions.newBuilder().setCredentials(credentials).build().getService();
    }
}
```

📌 파일 업로드 예제

```java
public void uploadToGCS(String filename, MultipartFile file) throws IOException {
    BlobId blobId = BlobId.of("my-gcs-bucket", filename);
    BlobInfo blobInfo = BlobInfo.newBuilder(blobId).setContentType(file.getContentType()).build();

    gcsStorage.create(blobInfo, file.getBytes());
}
```

📌 다운로드 예제

```java
public byte[] downloadFromGCS(String filename) {
    Blob blob = gcsStorage.get(BlobId.of("my-gcs-bucket", filename));
    return blob.getContent();
}
```

📌 삭제 예제

```
public void deleteFromGCS(String filename) {
    gcsStorage.delete(BlobId.of("my-gcs-bucket", filename));
}
```

✅ 3. S3 vs GCS 비교

항목	S3	GCS
API 명	`S3Client` (SDK v2)	`Storage`
인증	Access Key / Secret Key	서비스 계정 JSON
URI 형식	`s3://bucket/key`	`gs://bucket/key`
권한 관리	IAM + Bucket Policy	IAM + ACL
Spring 연동성	S3 → spring-cloud-aws 지원 있음	GCS는 직접 연동 필요

✅ 4. 실무 팁 요약

목적	전략
대용량 파일 처리	스트리밍으로 전송 (`InputStream`, `fromInputStream()`)
업로드 경로 관리	버킷/폴더 구조 → `bucket/folder/filename.ext`
임시 공개 URL 발급	S3: `PresignedUrl`, GCS: `SignedUrl`
파일명 충돌 방지	UUID, timestamp 등으로 식별자 생성
테스트	`LocalStack (S3)`, `Fake GCS server`, `TestContainers` 활용 가능

✅ 결론 요약

항목	설명
파일 저장 방식	클라우드 스토리지를 외부 파일 시스템처럼 사용
S3 연동	`S3Client`, `PutObjectRequest`, `GetObjectRequest` 중심
GCS 연동	`Storage`, `BlobId`, `BlobInfo` 중심
실무 권장	이미지/영상/대용량 로그 저장소 등으로 적합
보안	버킷 권한 분리, ACL 제한, 업로드 확장자 검사 필수