4. 웹 개발 기초

Spring MVC 개념

- ## DispatcherServlet 구조

Spring MVC와 Spring Boot에서의 **DispatcherServlet**은 전체 웹 애플리케이션의 **프론트 컨트롤러(Front Controller)** 역할을 한다.
모든 HTTP 요청은 이 DispatcherServlet을 통해 흐르고,
Spring은 이를 통해 **Controller**, **View**, **Model**, **Exception 처리**를 연결하는 구조로 동작한다.

아래에 DispatcherServlet의 구조와 동작 흐름, 주요 컴포넌트, Spring Boot에서의 자동 설정까지 모두 정리한다.

---

1. DispatcherServlet이란?

- Spring Web MVC의 **핵심 서블릿**
- 모든 HTTP 요청을 가로채어 **요청 흐름을 중앙에서 제어**
- 클라이언트 → DispatcherServlet → Controller → Service → View 로직 수행

DispatcherServlet은 Java EE의 `javax.servlet.http.HttpServlet`을 상속한 클래스다.

---

2. 전체 요청 흐름 구조

```
[Client HTTP Request]
        ↓
[DispatcherServlet]  ← 웹 애플리케이션의 단일 진입점
        ↓
[HandlerMapping] ← 어떤 컨트롤러를 실행할지 결정
        ↓
[HandlerAdapter] ← 실제로 컨트롤러 메서드 호출
        ↓
[Controller (Handler)]
        ↓
[ModelAndView 반환]
        ↓
[ViewResolver] ← 어떤 뷰(html, json 등)를 선택할지 결정
        ↓
[View 렌더링 (Thymeleaf, JSP 등)]
        ↓
[HTTP Response 반환]
```

---

3. DispatcherServlet의 주요 역할

역할	설명
요청 수신	모든 요청은 DispatcherServlet을 통함 (`/*`, `/api/*`)
컨트롤러 찾기	HandlerMapping으로 어떤 Controller에 전달할지 결정
컨트롤러 실행	HandlerAdapter가 컨트롤러 메서드 실행
모델 반환 처리	Controller가 반환한 ModelAndView 처리
뷰 이름 → 뷰 객체	ViewResolver 사용
응답 렌더링	View 객체의 render 메서드로 HTML, JSON 등을 생성

---

4. 주요 컴포넌트 설명

1) HandlerMapping

- URL → 어떤 컨트롤러 메서드로 매핑할지 결정
- Spring Boot에서는 기본적으로 `RequestMappingHandlerMapping` 사용

예:

```
@GetMapping("/hello")
public String hello() {
    return "hello";
}
```

→ `/hello` 요청이 들어오면 `hello()` 메서드가 매핑됨

---

2) HandlerAdapter

- HandlerMapping이 반환한 핸들러를 실제로 실행해주는 역할
- `@Controller`, `@RestController` 등을 지원

---

3) ViewResolver

- `ModelAndView` 또는 String으로 반환된 뷰 이름을 **물리적인 템플릿 파일로 변환**
- 예: `"home"` → `/templates/home.html`, `/WEB-INF/views/home.jsp`

Spring Boot에서는 다음과 같은 설정으로 ViewResolver 경로가 정해짐:

```
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

또는 Thymeleaf:

```
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
```

---

4) View

- 렌더링 책임 (HTML, JSON 등)
- `render(model, request, response)` 메서드를 통해 실제 응답을 생성

---

5. DispatcherServlet과 Spring Boot

Spring Boot는 DispatcherServlet을 자동으로 등록한다. (기본 경로 `/`)

```
@Bean
public DispatcherServlet dispatcherServlet() {
    return new DispatcherServlet();
}
```

이 설정은 내부적으로 다음 경로에서 자동 구성된다:

```
spring-boot-autoconfigure
└── WebMvcAutoConfiguration
    └── DispatcherServletAutoConfiguration
```

경로 변경도 가능:

```
spring.mvc.servlet.path=/api
```

6. 예외 처리도 DispatcherServlet이 담당

- Controller에서 예외가 발생하면, DispatcherServlet이 이를 **ExceptionResolver**로 전달
- `@ExceptionHandler`, `@ControllerAdvice`, `ErrorController` 등을 통해 공통 예외 처리 가능

7. DispatcherServlet 생명주기

단계	설명
초기화	`DispatcherServlet.init()` 시, HandlerMapping, ViewResolver 등을 초기화
요청 처리	HTTP 요청 → `doDispatch()` 호출 → 전체 흐름 제어
종료	서블릿 컨테이너 종료 시 cleanup 수행

8. 전체 코드 흐름 요약

```
@Controller
public class HelloController {
    @GetMapping("/hello")
    public String hello(Model model) {
        model.addAttribute("name", "Spring");
        return "hello"; // hello.html
    }
}
```

→ `/hello` 요청

→ DispatcherServlet

→ HandlerMapping → HelloController.hello()

→ Model 생성 및 View 이름 반환

→ ViewResolver → hello.html 템플릿 찾기

→ View.render() → 응답 HTML 생성

→ HTTP 응답 반환

결론

DispatcherServlet은 Spring MVC의 핵심이며,
모든 HTTP 요청의 흐름을 제어하는 **중앙 관제탑**이다.
View 렌더링, 예외 처리, Handler 연결 모두 이 한 곳을 통해 이루어진다.

이 구조를 정확히 이해하면, Spring MVC의 동작 원리와 커스터마이징 지점들(Interceptor, Filter, HandlerInterceptor 등)을 효과적으로 활용할 수 있다.

- # Controller, Service, Repository 계층 설계

Spring Boot에서는 전형적으로 **3계층 아키텍처 (Controller - Service - Repository)**를 따라 애플리케이션을 구성한다.
이 구조는 각 계층의 **역할을 명확히 분리**하고, **책임과 변경의 파급을 최소화**하는 데 목적이 있다.

---

1. 계층 아키텍처 구조

```
[Client HTTP Request]
       ↓
@Controller  ← 웹 요청 처리 (입력/출력, API 응답 포맷, DTO 변환 등)
       ↓
@Service     ← 비즈니스 로직 처리, 트랜잭션 관리
       ↓
@Repository  ← 데이터 접근 (JPA, MyBatis, JDBC 등)
```

---

2. 각 계층의 책임과 역할

1) Controller 계층

- 웹 요청의 **진입점**
- 사용자로부터 들어온 데이터를 수신 및 검증
- `@RestController`, `@Controller` 사용
- View 혹은 JSON 반환

역할 요약

- HTTP 요청/응답 처리
- DTO 변환
- Service 계층 호출

예시

```
@RestController
@RequestMapping("/api/members")
public class MemberController {
    private final MemberService memberService;

    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }

    @PostMapping
    public ResponseEntity<MemberDto> join(@RequestBody MemberDto dto) {
        Member member = memberService.join(dto);
        return ResponseEntity.ok(MemberDto.from(member));
    }
}
```

---

2) Service 계층

- 핵심 **비즈니스 로직** 담당
- 트랜잭션 처리 (`@Transactional`)
- 도메인 로직 수행
- 여러 Repository 조합 및 외부 API 호출 포함

역할 요약

- 로직 조합, 검증
- 트랜잭션 경계 정의
- Service 간 호출 조율

예시

```
@Service
public class MemberService {
```

```
    private final MemberRepository memberRepository;

    public MemberService(MemberRepository memberRepository) {
        this.memberRepository = memberRepository;
    }

    @Transactional
    public Member join(MemberDto dto) {
        if (memberRepository.existsByEmail(dto.getEmail())) {
            throw new IllegalArgumentException("이미 가입된 이메일입니다.");
        }
        Member member = new Member(dto.getEmail(), dto.getName());
        return memberRepository.save(member);
    }
}
```

---

3) Repository 계층

- **DB 또는 외부 저장소와 직접 통신**
- Spring Data JPA, MyBatis, JDBC, MongoDB 등 사용
- `@Repository` 애노테이션 사용
- 예외 자동 변환 기능 포함 (`PersistenceExceptionTranslationPostProcessor`)

역할 요약

- CRUD
- 쿼리 메서드 정의
- JPQL, QueryDSL, Native SQL 사용

예시

```
@Repository
public interface MemberRepository extends JpaRepository<Member, Long> {
    boolean existsByEmail(String email);
}
```

---

3. DTO와 Entity 분리

항목	DTO	Entity
사용 목적	외부와 데이터 송수신	DB 테이블 매핑
위치	Controller ↔ Client	Repository ↔ DB
수정 자유도	자유롭게 필드 구성 가능	정형화되어 있어 제한적

예시

```
public class MemberDto {
    private String name;
    private String email;

    public static MemberDto from(Member member) {
        return new MemberDto(member.getName(), member.getEmail());
    }
}
```

---

4. 계층 간 의존성 흐름 (단방향)

```
Controller → Service → Repository
```

반대 방향으로 참조하면 순환참조 발생 가능성 있음 (Spring Boot는 자동 탐지하여 오류 발생)

---

5. 트랜잭션 처리 위치

- 트랜잭션은 반드시 **Service 계층에서 시작**
- Controller에는 `@Transactional`을 **절대 사용하지 않음**

```
@Service
@Transactional
public class OrderService { ... }
```

---

6. 테스트 전략

계층	테스트 유형	설명
Controller	WebMvcTest	요청-응답, 유효성 검증
Service	UnitTest	순수 자바 테스트 (Mockito 등)
Repository	DataJpaTest	실제 DB와 쿼리 테스트

---

7. 계층 분리의 이점

장점	설명
관심사 분리	각 계층이 명확한 책임을 가짐
유지보수성	로직 변경이 다른 계층에 영향 주지 않음
테스트 용이성	각 계층별로 독립적 테스트 가능
유연한 확장	외부 API → Repository처럼 다룰 수 있음

---

8. 정리 예시

```
- com.example
  ├── controller/
  |   └── MemberController.java
  ├── service/
  |   └── MemberService.java
  ├── repository/
  |   └── MemberRepository.java
  ├── domain/
  |   └── Member.java
  ├── dto/
  |   └── MemberDto.java
```

결론

Spring Boot의 Controller-Service-Repository 구조는 애플리케이션을 **명확한 책임 단위로 나누고**,
로직, 데이터 접근, 표현 계층을 분리함으로써 유지보수성과 확장성을 확보하는 매우 중요한 설계 원칙이다.

@RestController vs @Controller

`@RestController`와 `@Controller`는 Spring MVC에서 **웹 요청을 처리하는 클래스에 붙는 애노테이션**이다.
두 애노테이션은 **동일한 계층(Controller 계층)에 위치**하지만,
주로 **응답 방식(View vs JSON)**에서 명확하게 다르다.

---

1. 기본 정의

애노테이션	구성	용도
`@Controller`	`@Component` + View 반환	JSP/Thymeleaf 등 HTML View 렌더링에 사용
`@RestController`	`@Controller` + `@ResponseBody`	JSON, XML 등 데이터 API 응답 전용

---

2. 내부 구조 차이

```
// 내부적으로 이렇게 구성되어 있음
@RestController = @Controller + @ResponseBody
```

즉, `@RestController`는 컨트롤러의 모든 메서드에 `@ResponseBody`가 자동 적용됨.
반면, `@Controller`는 뷰(View)를 반환하며, JSON 응답 시에는 반드시 `@ResponseBody`를 추가해야 함.

---

3. 응답 방식 차이

애노테이션	반환값 처리 방식	예시
`@Controller`	뷰 이름 반환 → ViewResolver가 HTML 렌더링	`"home"` → `home.html`
`@RestController`	객체를 반환 → JSON 변환 (Jackson 등 사용)	`return new User("kim")` → `{ "name": "kim" }`

---

4. 사용 예시 비교

1) @Controller 예시 (View 반환)

```
@Controller
public class PageController {

    @GetMapping("/home")
    public String home(Model model) {
        model.addAttribute("message", "Hello");
        return "home"; // → home.html 렌더링
    }
}
```

→ ViewResolver를 통해 `resources/templates/home.html`이 렌더링됨 (Thymeleaf, JSP 등)

---

2) @RestController 예시 (JSON 반환)

```
@RestController
public class ApiController {

    @GetMapping("/api/user")
    public User getUser() {
        return new User("kim", 30);
    }
}
```

→ `{"name":"kim", "age":30}` 형태의 JSON 자동 반환됨
→ `@ResponseBody` 없이도 자동 직렬화 적용

---

5. `@Controller`에서 JSON을 반환하고 싶다면?

```
@Controller
public class ApiController {

    @ResponseBody
    @GetMapping("/api/data")
    public Map<String, String> data() {
        return Map.of("result", "ok");
    }
}
```

→ `@ResponseBody`를 명시적으로 추가해야 JSON 반환됨

---

6. 사용 목적에 따른 선택 기준

목적	추천 애노테이션
HTML 페이지, 템플릿 출력	`@Controller`
RESTful API (JSON/XML)	`@RestController`
Ajax 요청 응답 전용	`@RestController`
비동기 요청 처리	`@RestController`
Thymeleaf/JSP 통합	`@Controller`

---

7. Spring Boot의 기본 설정 흐름

- Spring Boot는 기본적으로 **Thymeleaf가 있으면 View 렌더링 컨트롤러 우선**
- 의존성에 `spring-boot-starter-web`만 포함하면 JSON 기반 API 컨트롤러로 동작

---

8. REST와 REST Controller는 다르다

- `@RestController`는 REST API 스타일 응답을 위한 기술적 도구일 뿐,
  REST 아키텍처를 보장하거나 강제하는 것은 아님
- REST 설계 원칙: URI 설계, 상태 코드, HATEOAS 등은 별개로 관리해야 한다

결론

차이점	@Controller	@RestController
응답 방식	HTML View	JSON/XML 등 데이터
자동 직렬화	❌ (직접 `@ResponseBody` 필요)	⭕

Spring Boot에서는 **웹 페이지 렌더링이 목적이면** `@Controller`,
데이터 API가 목적이면 `@RestController`를 사용하는 것이 명확하고 직관적인 설계 방식이다.

요청 매핑: `@RequestMapping`, `@GetMapping` 등

Spring MVC에서는 **HTTP 요청을 처리할 메서드에 경로와 HTTP 메서드 타입을 연결**하기 위해
`@RequestMapping`과 단축형 애노테이션인 `@GetMapping`, `@PostMapping`, `@PutMapping` 등을 사용한다.
이들 애노테이션은 Controller 계층에서 **URL과 메서드의 매핑 규칙을 선언**하는 핵심 도구다.

---

1. @RequestMapping (기본형)

```
@RequestMapping("/path")
```

- 모든 HTTP 메서드(GET, POST 등)를 처리하는 **범용 매핑 애노테이션**
- 주로 클래스 레벨에서 공통 경로 설정 시 사용

```
@RequestMapping("/api/members")
@RestController
public class MemberController {

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String hello() {
        return "Hello";
    }
}
```

---

2. 단축형 애노테이션

Spring 4.3부터는 HTTP 메서드별로 다음과 같은 **축약 애노테이션**을 제공한다.

HTTP 메서드	애노테이션	설명
GET	`@GetMapping`	리소스 조회
POST	`@PostMapping`	리소스 생성
PUT	`@PutMapping`	리소스 전체 수정
PATCH	`@PatchMapping`	리소스 일부 수정
DELETE	`@DeleteMapping`	리소스 삭제

예시

```java
@GetMapping("/members")
public List<Member> findAll() { ... }

@PostMapping("/members")
public Member create(@RequestBody Member member) { ... }

@PutMapping("/members/{id}")
public Member update(@PathVariable Long id, @RequestBody Member member) { ... }

@DeleteMapping("/members/{id}")
public void delete(@PathVariable Long id) { ... }
```

---

3. 클래스 vs 메서드 레벨 @RequestMapping

클래스 수준 매핑 (공통 prefix)

```java
@RequestMapping("/api/members")
@RestController
public class MemberController {

    @GetMapping("/{id}")
    public Member get(@PathVariable Long id) {
        ...
    }
}
```

→ `/api/members/1` 요청 매핑됨

---

4. 주요 속성

속성	설명	예시
`value`	경로(Path) 지정	`@GetMapping("/hello")`
`method`	HTTP 메서드	`@RequestMapping(value="/", method=GET)`
`consumes`	요청 Content-Type 제한	`@PostMapping(consumes = "application/json")`
`produces`	응답 Content-Type 설정	`@GetMapping(produces = "application/json")`
`params`	특정 쿼리 파라미터 조건	`@GetMapping(params = "type=admin")`
`headers`	특정 헤더 존재 시에만 매핑	`@GetMapping(headers = "X-API-VERSION=1")`

---

5. 복수 경로 매핑

```
@GetMapping({"/hello", "/hi"})
public String hello() {
    return "Hello or Hi";
}
```

---

6. @RequestMapping vs 단축형 차이점

항목	@RequestMapping	단축형 (`@GetMapping` 등)
범용성	모든 HTTP 메서드 처리 가능	특정 HTTP 메서드만
명시성	method 옵션으로 제어	애노테이션 자체로 명확
권장도	클래스 레벨에서 prefix 지정 시 유용	메서드 레벨에서는 단축형 사용 권장

---

7. 응답 제어 관련 확장

```
@GetMapping("/members")
public ResponseEntity<List<Member>> getAll() {
    List<Member> members = memberService.findAll();
    return ResponseEntity
        .status(HttpStatus.OK)
        .body(members);
}
```

→ `ResponseEntity`를 이용하면 상태 코드, 헤더, 바디를 모두 제어 가능

---

8. 잘못된 매핑 예시 및 주의사항

- **중복된 경로와 메서드 조합은 에러 발생**
- **동일한 경로에 여러 메서드 존재 시 method 속성 명시 필수**
- RESTful 설계에서는 **의미 있는 URI + HTTP 메서드 조합**이 중요

결론

Spring의 `@RequestMapping` 계열 애노테이션은 HTTP 요청을 어떤 Controller 메서드에 매핑할지 선언적으로 명시하는 가장 기본적이면서도 중요한 기능이다.
실무에서는 단축형 애노테이션을 메서드에 사용하고, 클래스에는 `@RequestMapping("/prefix")`로 공통 경로를 설정하는 방식이 가장 널리 사용된다.

요청 파라미터: `@RequestParam`, `@PathVariable`, `@ModelAttribute`

Spring MVC에서 **요청 파라미터를 컨트롤러 메서드의 인자로 주입하는 방식**은 상황에 따라 다음과 같이 다양하게 제공된다:

- `@RequestParam`: 쿼리 스트링 또는 폼 필드 값
- `@PathVariable`: URL 경로의 일부
- `@ModelAttribute`: 폼 데이터나 요청 파라미터를 객체로 바인딩

각 방식은 용도와 데이터 구조에 따라 선택되어야 하며, 아래에 비교와 예제를 중심으로 상세히 정리한다.

---

1. `@RequestParam`

HTTP 요청 파라미터(쿼리 스트링, 폼 필드)를 개별 값으로 추출

1.1 사용 대상

- `?name=kim&age=30` 같은 **쿼리 파라미터**
- `<form>` 전송 시 각 입력 필드

1.2 기본 사용 예시

```
@GetMapping("/hello")
public String hello(@RequestParam String name, @RequestParam int age) {
    return "name = " + name + ", age = " + age;
}
```

→ `/hello?name=kim&age=20` 요청 시: `"name = kim, age = 20"`

1.3 옵션

속성	설명
`value`	파라미터 이름 지정
`required`	필수 여부 (기본값 true)
`defaultValue`	값이 없을 때 기본값

1.4 예시 (기본값, 선택 파라미터)

```
@GetMapping("/search")
public String search(@RequestParam(required = false, defaultValue = "0") int page) {
    return "page = " + page;
}
```

---

2. `@PathVariable`

URL 경로 자체에 포함된 값을 변수로 추출

2.1 사용 대상

- RESTful API 설계에서 자주 사용
- 예: `/users/1001`의 `1001` → `@PathVariable Long id`

2.2 기본 사용 예시

```
@GetMapping("/users/{id}")
public String getUser(@PathVariable("id") Long userId) {
    return "User ID = " + userId;
}
```

→ `/users/123` 요청 시: `"User ID = 123"`

2.3 변수명 일치 시 생략 가능

```
@GetMapping("/items/{itemId}")
public String getItem(@PathVariable Long itemId) {
    return "Item = " + itemId;
}
```

---

3. `@ModelAttribute`

요청 파라미터를 객체에 자동으로 바인딩하여 주입

3.1 사용 대상

- 폼 제출 데이터 (`POST form`)
- 복수의 파라미터 → 하나의 객체로 받을 때

3.2 예시: DTO 객체에 바인딩

```
public class MemberForm {
    private String name;
    private int age;
    // Getter/Setter 필수
}

@PostMapping("/join")
public String join(@ModelAttribute MemberForm form) {
    return "name = " + form.getName() + ", age = " + form.getAge();
}
```

→ `name=kim&age=20` 요청 본문이 자동으로 `MemberForm` 객체에 매핑됨

3.3 생략 가능 여부

- `@ModelAttribute`는 생략 가능 → 단, 객체 타입일 때만 생략되고, `@RequestBody`와 혼동되지 않아야 함

---

4. 세 방식 비교

구분	`@RequestParam`	`@PathVariable`	`@ModelAttribute`
용도	쿼리 파라미터, 폼 필드	URL 경로 변수	여러 파라미터 → 객체
대상 데이터	`?key=value`	`/resource/{id}`	`name=kim&age=20`
타입	기본형, String 등	단일 값	객체
필드 개수	적음	1~2개	많을 때 유리
생략 가능	파라미터명 일치 시 일부 가능	변수명 일치 시 가능	객체일 경우 가능
바인딩 방식	파라미터 → 변수	경로 → 변수	파라미터 → 객체 필드

---

5. 사용 예시 통합

```
@GetMapping("/products/{id}")
public String getProduct(
    @PathVariable Long id,
    @RequestParam(defaultValue = "1") int page,
    @ModelAttribute SearchForm form
) {
    return "id = " + id + ", page = " + page + ", keyword = " + form.getKeyword();
}
```

---

6. 오해 방지

- `@RequestParam`과 `@ModelAttribute`는 모두 쿼리 스트링에서 값을 가져오지만, **목표가 객체냐 단일 값이냐**에 따라 사용이 갈림
- `@PathVariable`은 반드시 경로 변수와 **URL 템플릿이 일치**해야 함

결론

목적	사용 애노테이션
단일 값 파라미터 (폼, 쿼리)	`@RequestParam`
RESTful 경로 변수	`@PathVariable`
다수 파라미터를 객체로 수신	`@ModelAttribute`

다음으로는 `@RequestBody`, `BindingResult`, `Validation`, `DTO 변환 전략`, `Content-Type: application/json` 대응 전략 등으로 이어갈 수 있다. 계속 진행하려면 `"계속"`이라고 입력하면 된다.

응답 처리: `ResponseEntity`, JSON 반환

Spring Boot에서의 **응답 처리**는 HTTP 요청에 대해 **적절한 데이터(JSON 등)**를 반환하거나,
상태 코드와 헤더를 제어하여 클라이언트에게 필요한 정보를 제공하는 데 초점을 맞춘다.
특히 REST API 개발에서는 다음 두 가지 방식이 널리 사용된다:

1. 객체를 직접 반환 → **JSON 자동 직렬화**
2. `ResponseEntity<T>` → **상태 코드, 헤더, 바디 전부 직접 제어**

---

1. 객체 반환 → JSON 자동 변환

Spring Boot는 `@RestController` 또는 `@ResponseBody`가 붙은 컨트롤러에서
객체를 반환하면 내부적으로 Jackson 등을 통해 **자동으로 JSON으로 변환**한다.

```
@RestController
public class UserController {

    @GetMapping("/user")
    public User getUser() {
        return new User("kim", 30);
    }
}
```

결과 응답 (Content-Type: application/json):

```
{
  "name": "kim",
  "age": 30
}
```

---

2. `ResponseEntity<T>` 사용 – 더 유연한 응답

`ResponseEntity<T>`는 HTTP 응답 전체를 제어할 수 있게 해주는 Spring의 유틸리티 클래스다.

2.1 예: 200 OK + JSON 응답

```
@GetMapping("/user")
public ResponseEntity<User> getUser() {
    User user = new User("kim", 30);
    return ResponseEntity.ok(user); // 상태 코드 200 자동 지정
}
```

2.2 예: 201 Created + 메시지

```
@PostMapping("/user")
public ResponseEntity<String> createUser(@RequestBody UserDto dto) {
    userService.save(dto);
    return ResponseEntity.status(HttpStatus.CREATED).body("사용자 생성됨");
}
```

2.3 예: 커스텀 헤더 + 바디

```
@GetMapping("/download")
public ResponseEntity<byte[]> downloadFile() {
    byte[] fileData = ...;
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);

headers.setContentDisposition(ContentDisposition.attachment().filename("report.pdf").bui
ld());

    return new ResponseEntity<>(fileData, headers, HttpStatus.OK);
}
```

---

3. 상태 코드 제어 요약

상황	메서드	상태 코드
성공 응답	`ResponseEntity.ok(body)`	200 OK
리소스 생성	`ResponseEntity.status(HttpStatus.CREATED)`	201 Created
잘못된 요청	`ResponseEntity.badRequest()`	400 Bad Request
인증 오류	`ResponseEntity.status(HttpStatus.UNAUTHORIZED)`	401 Unauthorized
권한 없음	`ResponseEntity.status(HttpStatus.FORBIDDEN)`	403 Forbidden
리소스 없음	`ResponseEntity.notFound()`	404 Not Found

---

4. 응답 JSON 커스터마이징 (응답 DTO 사용)

DTO 설계:

```
@Getter
@AllArgsConstructor
public class ApiResponse<T> {
    private int code;
    private String message;
    private T data;
}
```

컨트롤러에서 사용:

```
@GetMapping("/user")
public ResponseEntity<ApiResponse<User>> getUser() {
    User user = new User("kim", 30);
    ApiResponse<User> response = new ApiResponse<>(200, "조회 성공", user);
    return ResponseEntity.ok(response);
}
```

응답 JSON:

```
{
  "code": 200,
  "message": "조회 성공",
  "data": {
    "name": "kim",
    "age": 30
  }
}
```

---

5. 정리: 선택 기준

목적	방법
단순 JSON 응답	객체 반환
상태 코드와 헤더 제어	`ResponseEntity<T>` 사용
예외 상황 처리	`ResponseEntity` 또는 `@ControllerAdvice` 활용
REST API의 응답 구조 일관화	응답 DTO (`ApiResponse<T>`) 설계

결론

- `@RestController`는 객체를 JSON으로 자동 직렬화하여 응답할 수 있게 해주며,
- `ResponseEntity<T>`는 상태 코드, 헤더, 바디를 **세밀하게 제어할 수 있는 방법**이다.
- **실무에서는 두 방식을 조합해 사용하는 것이 일반적**이며, 응답 형식을 일관성 있게 유지하기 위해 **응답 DTO를 설계**하는 것이 중요하다.