10. 웹 계층 (Spring MVC) 심화

DispatcherServlet 구조

`DispatcherServlet`은 Spring MVC의 핵심 구성 요소로,
HTTP 요청을 중앙에서 받아서 처리 흐름을 분기하고, 응답을 작성하는 프론트 컨트롤러(Front Controller)이다.

다음은 `DispatcherServlet`의 **역할, 동작 구조, 내부 구성, 처리 순서, 주요 컴포넌트, 커스터마이징 전략, 실무 활용법**까지 포함한
가장 깊이 있고 체계적인 설명이다.

📘 1. DispatcherServlet이란?

- Spring MVC에서 **HTTP 요청을 받아 Controller로 전달하고, 결과를 View로 렌더링**하는 **프론트 컨트롤러**
- `web.xml` 또는 Spring Boot에서는 `DispatcherServlet`이 **서블릿 경로 `/`에 등록**되어 모든 요청을 가로챔

> 📌 핵심: Spring MVC의 중심 축이자 **모든 요청 진입점**

🧱 2. DispatcherServlet 기본 구조도

```
[Client HTTP Request]
          ↓
   DispatcherServlet
          ↓
  ┌──────────────────────────────────┐
  | HandlerMapping                   | → 어떤 컨트롤러가 처리할지 결정
  | HandlerAdapter                   | → 실제 컨트롤러 실행
  | HandlerInterceptor(s)            | → 사전/후 처리
  | ArgumentResolvers                | → @RequestParam, @ModelAttribute 해석
  | ReturnValueHandlers              | → @ResponseBody 등 처리
  | ViewResolver                     | → 뷰 이름 → View 객체
  | View                             | → 최종 HTML 또는 JSON 렌더링
  └──────────────────────────────────┘
          ↓
[Client HTTP Response]
```

🔄 3. DispatcherServlet 요청 처리 흐름

```
1. 요청 수신 → DispatcherServlet
2. HandlerMapping → 요청 URL과 매핑되는 컨트롤러 탐색
3. HandlerAdapter → 해당 컨트롤러를 실행
4. ArgumentResolver → 메서드 매개변수 바인딩
5. Interceptor (preHandle)
6. Controller → 비즈니스 로직 수행
7. ReturnValueHandler → 반환값 처리
8. ViewResolver → View 이름을 View 객체로 변환
9. View 렌더링 (HTML or JSON 등)
10. Interceptor (postHandle, afterCompletion)
11. 최종 응답 반환
```

⚙️ 4. 주요 내부 컴포넌트

컴포넌트	설명
`HandlerMapping`	URL → 컨트롤러 찾기 (`@RequestMapping`)
`HandlerAdapter`	컨트롤러 메서드 실행 어댑터
`HandlerInterceptor`	전/후처리 필터 (로그인 검증 등)
`ArgumentResolver`	`@RequestParam`, `@PathVariable` 등 해석
`ReturnValueHandler`	`@ResponseBody`, `ModelAndView` 처리
`ViewResolver`	View 이름 → View 객체 변환
`View`	최종 렌더링 (JSP, Thymeleaf, JSON 등)

🧪 5. 실전 흐름 예제

요청

```
GET /user/1
```

컨트롤러

```
@GetMapping("/user/{id}")
public String getUser(@PathVariable Long id, Model model) {
    model.addAttribute("user", userService.getUser(id));
    return "user/detail"; // 뷰 이름
}
```

처리 과정 요약

단계	내용
1. DispatcherServlet	요청 수신
2. HandlerMapping	`/user/{id}` → 컨트롤러 탐색
3. HandlerAdapter	컨트롤러 메서드 호출
4. ArgumentResolver	`@PathVariable id` 추출
5. ReturnValueHandler	뷰 이름 `"user/detail"` 해석
6. ViewResolver	`user/detail` → `/templates/user/detail.html`
7. View	Thymeleaf 렌더링
8. 응답 반환	HTML 완성 →

🧰 6. Spring Boot에서 DispatcherServlet 자동 설정

```
@Bean
public DispatcherServlet dispatcherServlet() {
    DispatcherServlet servlet = new DispatcherServlet();
    servlet.setThrowExceptionIfNoHandlerFound(true);
    return servlet;
}
```

→ Spring Boot에서는 `/` 경로에 자동 등록됨

⚠️ 7. 실무 주의사항

항목	설명
404 발생	`HandlerMapping`이 없거나 경로 불일치
View 이름 해석 안 됨	`ViewResolver` 설정 누락
JSON 반환 오류	`@ResponseBody` 없거나 `HttpMessageConverter` 미설정
Interceptor 적용 안 됨	`addPathPatterns()` 미설정
커스터마이징 필요	`WebMvcConfigurer`에서 구성 요소 주입 가능

🔧 8. DispatcherServlet 확장 전략

방법	설명
`HandlerInterceptor` 등록	로그인 인증, 로깅 등
`WebMvcConfigurer` 구현	MVC 설정 전체 확장
`Custom ArgumentResolver`	`@LoginUser` 등 사용자 정의 파라미터 처리
`Custom ReturnValueHandler`	표준 응답 래핑 처리
`ExceptionHandler` / `@ControllerAdvice`	전역 예외 처리 구성

✅ 9. 요약

항목	설명
정의	Spring MVC의 요청 진입점이자 분기 처리기
주요 기능	요청 → 컨트롤러 → 뷰로의 흐름 제어
핵심 컴포넌트	HandlerMapping, ViewResolver 등
자동 설정	Spring Boot에서 자동 등록
실무 확장 포인트	WebMvcConfigurer, Interceptor, ExceptionHandler 등

요청 매핑

- ## `@RequestMapping`, `@GetMapping`, `@PostMapping`

  다음은 Spring MVC에서 사용되는 핵심 HTTP 매핑 어노테이션인
  `@RequestMapping`, `@GetMapping`, `@PostMapping`의
  역할, 차이점, 속성, 동작 원리, 실무 사용 전략, 중복 매핑 처리법까지 포함한 가장 깊이 있는 설명이다.

  ### 📘 1. 개요 요약

어노테이션	설명
`@RequestMapping`	모든 HTTP 메서드(GET, POST 등)를 처리하는 **범용 매핑 어노테이션**
`@GetMapping`	`@RequestMapping(method = GET)` 의 축약형
`@PostMapping`	`@RequestMapping(method = POST)` 의 축약형

  → Spring 4.3 이후부터는 **특정 HTTP 메서드용 축약 어노테이션(@GetMapping 등)**을 사용하는 것이 **표준**이며, 가독성과 유지보수성이 더 좋음

🧱 2. `@RequestMapping`

✅ 역할

- **URL 패턴과 컨트롤러 메서드를 연결하는 전반적 라우팅 어노테이션**
- 다양한 속성 조합으로 매우 유연한 요청 처리 가능

✅ 예제

```
@RequestMapping(value = "/users", method = RequestMethod.GET)
public String getUsers() { ... }
```

✅ 주요 속성

속성	설명
`value` / `path`	URL 패턴 지정
`method`	HTTP 메서드 (`GET`, `POST` 등)
`params`	특정 요청 파라미터 유무로 매핑 제한
`headers`	특정 헤더 조건으로만 허용
`consumes`	요청 Content-Type 제한
`produces`	응답 Content-Type 지정

✅ 다양한 활용 예

```
@RequestMapping(path = "/test", method = RequestMethod.GET)
@RequestMapping(value = "/submit", method = {RequestMethod.POST, RequestMethod.PUT})
@RequestMapping(value = "/json", consumes = "application/json", produces =
"application/json")
@RequestMapping(value = "/download", headers = "X-File=true")
@RequestMapping(value = "/submit", params = "mode=fast")
```

🌐 3. `@GetMapping`, `@PostMapping`

Spring 4.3부터 도입된 **HTTP 메서드별 전용 매핑 어노테이션**

어노테이션	내부 변환
`@GetMapping`	`@RequestMapping(method = GET)`
`@PostMapping`	`@RequestMapping(method = POST)`
`@PutMapping`	PUT

어노테이션	내부 변환
`@DeleteMapping`	DELETE
`@PatchMapping`	PATCH

✅ 사용 예제

```
@GetMapping("/users")
public List<User> list() { ... }

@PostMapping("/users")
public void create(@RequestBody User user) { ... }
```

→ 가독성이 뛰어나고 REST API 설계에 매우 적합

🧩 4. 클래스 레벨 + 메서드 레벨 매핑

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) { ... }

    @PostMapping
    public void create(@RequestBody User user) { ... }
}
```

- `클래스 레벨 @RequestMapping("/api/users")` 는 **공통 prefix**
- 메서드 레벨은 **상대 경로**

⚠️ 5. 중복 매핑 처리

Spring은 하나의 요청 URL에 대해 **단 하나의 컨트롤러 메서드만 매핑되어야 함**

```
@GetMapping("/users")
public void a() {}

@RequestMapping("/users")
public void b() {}  // ❌ 애플리케이션 시작 시 충돌 → 예외 발생
```

→ URL + HTTP Method 조합이 겹치지 않도록 설계해야 함

🧠 6. 실무 전략

전략	설명
REST API 개발 시	`@GetMapping`, `@PostMapping` 등 **축약형 사용 권장**
HTML 렌더링 포함 시	`@RequestMapping`으로 확장성 확보 가능
필터링 조건이 많을 경우	`params`, `headers`, `consumes` 활용
JSON 응답 API일 경우	`@ResponseBody` 또는 `@RestController` 함께 사용
동적 PathVariable 매핑 시	`@GetMapping("/users/{id}")` 형태로 명확하게 표현

📌 7. 요약 비교표

항목	`@RequestMapping`	`@GetMapping` / `@PostMapping`
HTTP 메서드 지정	명시적 (`method = GET`)	축약형 전용 어노테이션
가독성	복잡한 매핑에 적합	REST API 작성에 적합
유연성	✅ 조건 기반 필터링 가능	제한적
실무 사용 권장	공통 prefix, 복합 조건 필요할 때	✅ 대부분의 컨트롤러 메서드

요청 파라미터 처리

- `@RequestParam`, `@PathVariable`, `@ModelAttribute`, `@RequestBody`

다음은 Spring MVC에서 클라이언트 요청 데이터를 **컨트롤러 메서드의 파라미터로 바인딩할 때 사용하는** 핵심 어노테이션인
`@RequestParam`, `@PathVariable`, `@ModelAttribute`, `@RequestBody`에 대한
개념, 동작 원리, 차이점, 사용 예제, 바인딩 방식, JSON/Form 연동, 실무 전략까지 포함한 가장 깊이 있는 설명이다.

📘 1. 개념 요약

어노테이션	용도
`@RequestParam`	**쿼리 파라미터**, `form-urlencoded` 등 단일 값 바인딩
`@PathVariable`	**URL 경로의 일부분**을 변수로 추출
`@ModelAttribute`	**폼 데이터 → 객체로 바인딩** (HTML Form에서 사용)
`@RequestBody`	**HTTP Body(JSON/XML 등)**를 객체로 역직렬화

🧩 2. 각각의 동작 방식과 예제

✅ 1. `@RequestParam`

설명:

- 요청 파라미터(Query String, Form field 등)를 단일 변수에 바인딩

요청 예:

```
GET /users?name=kim&age=30
```

컨트롤러:

```
@GetMapping("/users")
public String getUser(@RequestParam String name, @RequestParam int age) {
    ...
}
```

속성:

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

✅ 2. `@PathVariable`

설명:

- URL 경로 일부를 **동적으로 변수화**하여 매핑

요청 예:

```
GET /users/42
```

컨트롤러:

```
@GetMapping("/users/{id}")
public String getUser(@PathVariable("id") Long userId) {
    ...
}
```

→ `@PathVariable`은 `{}`로 감싼 URL 경로의 변수를 메서드 인자로 전달

✅ 3. `@ModelAttribute`

설명:

- HTML `<form>`에서 전송된 **요청 파라미터들을 Java 객체로 바인딩**
- 내부적으로 `@RequestParam`을 사용해서 각 필드를 바인딩
- **객체에 세터 메서드가 필수**

요청 예:

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

name=kim&age=30
```

DTO:

```
public class UserForm {
    private String name;
    private int age;
    // getter/setter 필수
}
```

컨트롤러:

```
@PostMapping("/register")
public String register(@ModelAttribute UserForm form) {
    ...
}
```

→ 생략 가능: `@ModelAttribute`는 생략해도 자동으로 적용됨

✅ 4. `@RequestBody`

설명:

- 요청의 **HTTP 본문(Body)**을 **JSON → Java 객체**로 역직렬화
- Jackson, Gson 등 HTTP Message Converter가 작동

요청 예:

```
POST /api/users
Content-Type: application/json

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

DTO:

```
public class UserDto {
    private String name;
    private int age;
    // getter/setter 필수
}
```

컨트롤러:

```
@PostMapping("/api/users")
public String create(@RequestBody UserDto dto) {
    ...
}
```

→ **객체로 바인딩 + @Valid와 함께 검증 가능**

⚠️ 3. 차이점 비교표

항목	`@RequestParam`	`@PathVariable`	`@ModelAttribute`	`@RequestBody`
사용 위치	쿼리 파라미터, Form field	URL 경로	Form field (`application/x-www-form-urlencoded`)	HTTP Body (JSON 등)
요청 방식	GET/POST 모두 가능	주로 REST GET/DELETE 등	HTML Form 전송에 특화	REST API (POST/PUT)
복수 필드 지원	❌ (개별 필드마다 지정)	❌	✅ 필드 자동 바인딩	✅ JSON 구조 매핑
MessageConverter 작동	❌	❌	❌	✅ (Jackson 등 동작)
DTO 클래스 필요	선택	선택	✅	✅
어노테이션 생략 가능	일부만 (`@RequestParam`)	❌	✅ 가능	❌

🧠 4. 실무 전략

상황	추천 어노테이션
단순 GET 쿼리 파라미터	`@RequestParam`
RESTful URL 경로 변수	`@PathVariable`
HTML Form 전송 (회원가입 등)	`@ModelAttribute`
JSON 기반 REST API 요청	`@RequestBody`

📌 5. 요약

어노테이션	목적
`@RequestParam`	쿼리 스트링 또는 폼 필드 단일 값 추출
`@PathVariable`	URL 경로 값 추출
`@ModelAttribute`	폼 필드 → 객체 자동 바인딩 (setter 필수)
`@RequestBody`	HTTP Body(JSON) → 객체 역직렬화 (MessageConverter 필요)

응답 처리

- ## `@ResponseBody`

`@ResponseBody`는 Spring MVC에서 **컨트롤러 메서드의 반환값을 HTTP 응답 바디에 직접 출력**하도록 지정하는 어노테이션이다.
기본적으로 Spring MVC는 반환값을 **뷰 이름으로 해석**하지만, `@ResponseBody`를 사용하면
뷰를 렌더링하지 않고 반환값을 JSON, XML 등으로 직렬화하여 바로 클라이언트에게 응답한다.

다음은 `@ResponseBody`의 **개념, 동작 원리, 직렬화 방식, 예제, MessageConverter 연동, RestController와의 관계, 실무 전략**까지 포함한 가장 깊이 있는 설명이다.

📘 1. @ResponseBody란?

✅ 정의

`@ResponseBody`는 **컨트롤러의 반환값을 HTTP 응답 본문에 직접 작성**하도록 지시하는 어노테이션이다.

> 📌 ViewResolver를 거치지 않고, **HttpMessageConverter를 통해 JSON 등으로 변환된 결과를 응답**으로 전송

🧱 2. 동작 방식

```
@GetMapping("/hello")
@ResponseBody
public String hello() {
    return "hello";  // → 뷰가 아니라 → HTTP Body "hello"
}
```

기본 MVC 흐름 vs @ResponseBody

기본 View 반환	`@ResponseBody` 반환
ViewResolver 작동	❌ 작동하지 않음
View 이름 해석	❌ X

기본 View 반환	`@ResponseBody` 반환
템플릿 엔진(JSP 등)	❌ 렌더링 없음
HttpMessageConverter	❌ 없음 → ✅ 작동함

🔧 3. JSON 응답 예제

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

응답:

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

→ 내부적으로 `MappingJackson2HttpMessageConverter`가 작동하여
Java 객체를 JSON 문자열로 변환하고, `Content-Type: application/json`으로 응답

🔁 4. @RestController와의 관계

```
@RestController
public class ApiController {
    @GetMapping("/user")
    public User getUser() { ... }
}
```

- `@RestController`는 **`@Controller + @ResponseBody`의 조합**
- 모든 메서드에 자동으로 `@ResponseBody`가 적용됨
- **REST API 개발 시 `@RestController` 사용을 권장**

🧪 5. 다양한 타입 반환

반환 타입	결과
`String`	텍스트 응답
`Object`	JSON/XML 응답 (MessageConverter에 따라 다름)
`List<T>`	JSON 배열

반환 타입	결과
`Map<String, ?>`	JSON 객체
`void` + `@ResponseBody`	204 No Content 응답

⚙️ 6. 내부 동작: HttpMessageConverter

Spring은 `@ResponseBody`가 적용된 메서드에서 반환된 객체를
다음 순서로 처리한다:

1. 컨트롤러 메서드 실행 → 객체 반환
2. `HandlerMethodReturnValueHandler` → `@ResponseBody` 확인
3. 등록된 `HttpMessageConverter` 중 적절한 것 선택
4. Java 객체를 JSON 등으로 직렬화 → HTTP 응답 바디에 기록

📦 대표 MessageConverter들

Converter 클래스	역할
`StringHttpMessageConverter`	`text/plain`
`MappingJackson2HttpMessageConverter`	`application/json`
`MappingJackson2XmlHttpMessageConverter`	`application/xml`

⚠️ 7. 주의사항

항목	설명
반환 타입이 문자열일 경우	`"hello"` → 그대로 응답됨 (`View 이름 아님`)
객체 반환 시 Jackson 필요	`jackson-databind`가 클래스패스에 있어야 함
ResponseBody 생략 시 뷰 해석됨	→ `@RestController` 또는 `@ResponseBody` 명시 필수
문자열이 JSON으로 직렬화되지 않음	→ 객체 반환해야 JSON 가능 (`new User(...)`)

✅ 8. 실무 전략

상황	전략
REST API 컨트롤러	`@RestController` 사용 (자동 JSON)
HTML 뷰 렌더링 컨트롤러	`@Controller + return "view"` 사용

상황	전략
단일 JSON 응답	`@ResponseBody` + 객체 반환
문자열 텍스트 응답 (`text/plain`)	`@ResponseBody` + `String` 반환 또는 `ResponseEntity<String>` 사용

📌 9. 요약

항목	설명
정의	반환값을 View가 아닌 HTTP Body로 직접 전송
자동 변환	JSON/XML 등은 HttpMessageConverter가 처리
RestController와의 관계	`@RestController` = `@Controller + @ResponseBody`
실무 활용	REST API 응답, Ajax 응답, JSON 통신 등에서 필수

• JSON 직렬화

JSON 직렬화(JSON Serialization)는 자바 객체(Java Object)를 **JSON 문자열(String)**로 변환하는 과정이다.
Spring에서는 이 작업을 대부분 자동으로 수행하며, 주로 **HTTP 응답(`@ResponseBody`)**, **파일 저장**, **API 통신**에서 사용된다.

다음은 **JSON 직렬화의 개념, Jackson 동작 방식, 주요 애너테이션, 직렬화 제어, Spring 연동 구조, 실무 튜닝 전략**까지 포함한 가장 깊이 있는 설명이다.

📘 1. JSON 직렬화란?

- **Java 객체 → JSON 문자열로 변환하는 과정**
- 예를 들어 다음 객체는:

```
public class User {
    private String name = "kim";
    private int age = 30;
}
```

→ JSON 직렬화 결과:

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

🔁 2. JSON 역직렬화란?

- **JSON 문자열 → Java 객체로 변환하는 과정**

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

→ `User` 객체로 역변환됨

→ 역직렬화 시 **기본 생성자 + 세터 or 필드 접근** 필요

🔧 3. Jackson: Spring 기본 JSON 직렬화 엔진

항목	설명
기본 엔진	Jackson (`com.fasterxml.jackson`)
주요 클래스	`ObjectMapper`, `JsonGenerator`, `JsonParser`
스프링 통합 방식	`MappingJackson2HttpMessageConverter` 자동 등록
Maven 의존성	Spring Boot 스타터에 자동 포함됨 (`spring-boot-starter-web`)

🧪 4. 예제: 객체 → JSON

```
ObjectMapper mapper = new ObjectMapper();

User user = new User("kim", 30);
String json = mapper.writeValueAsString(user);
System.out.println(json);
```

→ 결과:

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

⚙️ 5. JSON 직렬화 조건

조건	설명
기본 생성자 존재	Jackson이 내부적으로 객체를 생성해야 함
getter 또는 public field	필드 접근하여 JSON 필드 구성
순환 참조 없음	무한 루프 방지 필요 (`@JsonIgnore`, `@JsonManagedReference` 등)
필드 타입 직렬화 가능	`Date`, `enum` 등도 커스터마이징 필요할 수 있음

🧩 6. 주요 Jackson 애너테이션

애너테이션	설명
`@JsonProperty`	필드 이름 지정 (`snake_case` 등 가능)
`@JsonIgnore`	해당 필드 직렬화 제외
`@JsonInclude(Include.NON_NULL)`	null 제외
`@JsonFormat(pattern = "...")`	날짜/시간 형식 지정
`@JsonUnwrapped`	중첩 객체의 필드를 평면화
`@JsonManagedReference` / `@JsonBackReference`	양방향 참조 방지

예제:

```
public class User {
    @JsonProperty("user_name")
    private String name;

    @JsonIgnore
    private String password;

    @JsonInclude(JsonInclude.Include.NON_NULL)
    private String nickname;

    @JsonFormat(pattern = "yyyy-MM-dd")
    private LocalDate birthDate;
}
```

🧱 7. Spring Boot에서 자동 직렬화 흐름

```
컨트롤러 메서드 → @ResponseBody or @RestController →
MappingJackson2HttpMessageConverter →
ObjectMapper → JSON 문자열 생성 →
HTTP 응답 바디에 작성
```

예:

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

→ 결과:

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

🧰 8. 커스터마이징 ObjectMapper

전역 설정 (Spring Boot 2.x 이후)

```
@Configuration
public class JacksonConfig {
    @Bean
    public ObjectMapper objectMapper() {
        return JsonMapper.builder()
            .propertyNamingStrategy(PropertyNamingStrategies.SNAKE_CASE)
            .serializationInclusion(JsonInclude.Include.NON_NULL)
            .build();
    }
}
```

application.yml

```
spring:
  jackson:
    property-naming-strategy: SNAKE_CASE
    serialization:
      indent_output: true
      fail-on-empty-beans: false
```

🧠 9. 실무 전략

전략	설명
DTO 전용 직렬화 클래스 사용	Entity 직접 반환 X (지연 로딩, 순환 참조 문제 방지)
null 필드 제외	`@JsonInclude` 사용으로 경량화
날짜 포맷 명시	`@JsonFormat` 또는 글로벌 설정
순환 참조 주의	양방향 관계 시 `@JsonIgnore`, DTO 분리
Enum 변환 지정	`@JsonValue`, `@JsonCreator` 사용

📌 10. 요약

항목	설명
정의	Java 객체 → JSON 문자열 변환
엔진	Jackson (`ObjectMapper`)

항목	설명
스프링 연동	`@ResponseBody`, `@RequestBody`, `HttpMessageConverter`
주요 애너테이션	`@JsonProperty`, `@JsonIgnore`, `@JsonFormat` 등
실무 전략	DTO 사용, null 제거, 순환참조 회피, 날짜 포맷 지정 등

View Resolver

- ## Thymeleaf, JSP

`Thymeleaf`와 `JSP`는 모두 **서버사이드 템플릿 엔진(Server-Side Template Engine)**으로,
Java 웹 애플리케이션에서 **HTML 페이지를 동적으로 생성**하는 데 사용된다.
하지만 이 두 기술은 **철학, 구조, 장단점, 현대 웹과의 호환성** 측면에서 큰 차이를 가진다.

다음은 **Thymeleaf vs JSP**에 대한 **개념, 동작 원리, 문법 비교, 장단점, Spring Boot 연동 전략, 실무 추천**까지 포함한 가장 깊이 있는 설명이다.

📘 1. 정의 비교

항목	Thymeleaf	JSP (Java Server Pages)
정의	HTML 기반의 현대적 템플릿 엔진	자바 코드가 삽입된 서버측 HTML 문서
특징	XML/HTML 문법 완전 준수, 뷰-디자인 분리	HTML 내부에 Java 코드 (`<% %>`) 가능
대표 목적	Spring MVC와 HTML 뷰 렌더링	전통적인 Servlet/JSP 구조 기반 렌더링

⚙️ 2. 동작 구조 비교

✅ JSP 동작 구조

```
요청 → DispatcherServlet → InternalResourceViewResolver
     → *.jsp 컴파일 → Servlet → HTML 생성 → 응답
```

✅ Thymeleaf 동작 구조

```
요청 → DispatcherServlet → ThymeleafViewResolver
     → HTML + 템플릿 엔진 해석 → HTML 생성 → 응답
```

🧩 3. 문법 비교

✅ 변수 출력

엔진	문법
Thymeleaf	`${user.name}` → `<span th:text="${user.name}"></span>`
JSP	`${user.name}` → `${user.name}` or `<%= user.getName() %>`

✅ 조건문

Thymeleaf:

```
<div th:if="${user != null}">Hello</div>
```

JSP:

```
<c:if test="${user != null}">
    Hello
</c:if>
```

✅ 반복문

Thymeleaf:

```
<tr th:each="item : ${items}">
  <td th:text="${item.name}"></td>
</tr>
```

JSP:

```
<c:forEach var="item" items="${items}">
  <td>${item.name}</td>
</c:forEach>
```

🔧 4. Spring Boot 연동 비교

✅ Thymeleaf 설정 (Spring Boot)

```
implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
```

- 기본 경로: `src/main/resources/templates/`
- 기본 확장자: `.html`

✅ JSP 설정 (Spring Boot, 권장 X)

```
implementation 'org.apache.tomcat.embed:tomcat-embed-jasper'
```

- 기본 경로: `/WEB-INF/views/`
- 별도 설정 필요: JSP는 Spring Boot 기본 뷰 리졸버 대상이 아님

🎾 5. 장단점 비교

항목	Thymeleaf	JSP
HTML 호환성	✅ HTML5 완전 호환 (디자이너 협업 용이)	❌ HTML로 바로 실행 불가능
디자인/뷰 분리	✅ 가능	❌ Java 코드 혼재 가능
유지보수	✅ 쉽고 명확함	❌ 복잡한 뷰에서는 혼란 가능
IDE 미리보기	✅ HTML 자체로 열림	❌ JSP는 컴파일 필요
REST/SPA 연계	✅ Thymeleaf는 fragment, layout, ajax 등 지원	❌ 구조적으로 한계 있음
속도/성능	조금 느림 (HTML 파싱)	빠름 (서블릿 변환)
최신 표준 지원	✅ Spring 공식 지원	⚠️ Spring Boot에서 비권장

🚫 6. Spring Boot에서 JSP가 비권장되는 이유

- Spring Boot는 내장 톰캣 구조상 JSP를 기본 지원하지 않음
- WAR 패키징이 필요하며, `src/main/webapp/WEB-INF/` 경로도 강제됨
- **JSP는 서블릿 종속적이며, Spring MVC의 구조적 이점과 충돌이 많음**

> 📌 실무에서는 **Thymeleaf 또는 외부 프론트엔드(React, Vue 등) + REST API** 조합이 표준

✅ 7. 실무 추천 전략

상황	추천
Spring Boot 기반 웹 서비스	✅ **Thymeleaf** 사용
레거시 프로젝트 유지보수	JSP 사용 가능 (가급적 개선 필요)
디자이너와의 협업 필요	✅ Thymeleaf (정적 HTML 작업 가능)
REST API + SPA 개발	✅ JSON API + Vue/React로 분리
복잡한 뷰 레이아웃	✅ Thymeleaf Fragment/Layout 사용

📌 8. 요약

항목	Thymeleaf	JSP
HTML 정합성	✅ HTML5 완전 지원	❌ 스크립틀릿 혼합으로 깨짐 가능
MVC 구조	✅ 분리 철저	❌ 뷰에 로직 포함 우려
Spring Boot 지원	✅ 기본 내장	❌ 별도 설정 필요 + 비권장
유지보수	✅ 생산성 높음	❌ 복잡도 ↑

유효성 검사

- ## `@Valid`, `@Validated`

`@Valid`와 `@Validated`는 Spring에서 **입력값 검증(Validation)**을 위해 사용하는 어노테이션이다.
둘 다 Java Bean Validation(JSR-380)의 기반 위에서 작동하며, 주로 **Controller의 요청 DTO 검증**, **Service 계층의 메서드 파라미터 검증** 등에 사용된다.

다음은 `@Valid`와 `@Validated`의 **공통점, 차이점, 적용 위치, 내부 동작, 그룹 검증, 실무 전략, Bean Validation 연동 구조**까지 포함한 가장 깊이 있는 설명이다.

📘 1. 공통점 요약

항목	`@Valid` / `@Validated` 공통
기반 기술	Bean Validation(JSR-303/JSR-380)
대상	DTO 필드, 메서드 파라미터, 메서드 리턴 값 등
작동 방식	`javax.validation.Validator`를 통해 자동 검증
스프링 통합	Spring MVC에서 AOP 기반 자동 검증 지원 (`BindingResult`)

📜 2. `@Valid`

✅ 정의

- JSR-380(Bean Validation)의 표준 어노테이션
- `javax.validation.Valid` 패키지
- **기본 검증만 가능**, 그룹 지정 불가능

✅ 사용 위치

- 필드, 메서드 인자, 생성자, 리턴 값

✅ 예시

```
@PostMapping("/users")
public String createUser(@Valid @RequestBody UserDto dto, BindingResult result) {
    if (result.hasErrors()) {
        return "error";
    }
    ...
}
```

📋 3. `@Validated`

✅ 정의

- Spring 전용 어노테이션 (`org.springframework.validation.annotation.Validated`)
- `@Valid` 기능 + **검증 그룹(grouping)** 기능 제공

✅ 특징

- 그룹 기반 유효성 검사 가능 (`@Validated(Group1.class)` 등)
- 일반적으로 **서비스 계층, 메서드 레벨 검증**에 자주 사용됨

✅ 예시

```
@Validated(Group1.class)
@PostMapping("/users")
public String create(@RequestBody @Validated(Group1.class) UserDto dto) {
    ...
}
```

🔍 4. 차이점 요약

항목	`@Valid`	`@Validated`
패키지	`javax.validation.Valid`	`org.springframework.validation.annotation.Validated`
표준 여부	✅ JSR 표준	❌ Spring 전용
검증 그룹 지원	❌ 불가능	✅ 그룹 분류 가능
사용 위치	DTO, 필드, 메서드 인자 등	주로 Controller, Service 메서드 인자 등
일반 컨트롤러 검증	✅ 가능	✅ 가능
메서드/클래스 레벨 검증	❌ 거의 안 씀	✅ 많이 사용

🧪 5. DTO 검증 예제

```
public class UserDto {
    @NotBlank
    private String name;

    @Min(18)
    private int age;
}
```

```
@PostMapping("/user")
public ResponseEntity<?> createUser(@Valid @RequestBody UserDto dto,
                                    BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return ResponseEntity.badRequest().body(bindingResult.getAllErrors());
    }
    ...
}
```

🎯 6. 검증 그룹 예제 (`@Validated` 전용)

DTO

```
public class UserDto {

    @NotBlank(groups = Create.class)
    private String name;

    @Min(value = 18, groups = Update.class)
    private int age;

    public interface Create {}
    public interface Update {}
}
```

컨트롤러

```
@PostMapping("/user")
public void create(@Validated(UserDto.Create.class) @RequestBody UserDto dto) {
    ...
}
```

🧰 7. 서비스 계층에서 `@Validated` 사용

```
@Service
@Validated
public class UserService {

    public void save(@Valid UserDto dto) {
        // 내부에서도 검증됨
    }
}
```

→ Service 계층에서도 Spring AOP가 붙어 `MethodValidationPostProcessor`**가 작동**하여 검증 수행

⚠️ 8. 주의사항

항목	설명
`@Valid`는 그룹 검증 불가	→ `@Validated`로 대체해야 함
`@Valid` 생략 시 자동 검증 작동 안 함	MVC 자동 검증은 어노테이션 기반
`BindingResult` 없으면 예외 발생	`MethodArgumentNotValidException` 발생
서비스 메서드에는 `@Validated` 클래스 레벨 필요	메서드 파라미터 검증을 위해서

📌 9. 요약

항목	`@Valid`	`@Validated`
표준 여부	✅ JSR-380 표준	❌ Spring 전용
그룹 검증	❌ 불가능	✅ 가능 (`groups = ...`)
DTO 검증	✅ 자주 사용	✅ (그룹 필요할 때만)
서비스 검증	❌ 거의 안 씀	✅ 클래스 레벨에 붙이면 메서드도 검증
추천 전략	일반 검증: `@Valid` / 그룹·서비스 검증: `@Validated`	

- `BindingResult`

`BindingResult`는 Spring MVC에서 **입력값 검증(@Valid, @Validated)** 결과를 저장하는 객체로,
검증 에러가 발생해도 예외를 던지지 않고 컨트롤러에서 직접 처리할 수 있게 해주는 매우 중요한 인터페이스이다.

다음은 `BindingResult`의 **개념, 동작 원리, 사용 조건, 예제, 검증 흐름, 에러 처리 방법, 실무 전략**까지 포함한 가장 깊이 있는 설명이다.

📘 1. `BindingResult`란?

✅ 정의

- `org.springframework.validation.BindingResult`는
  `@Valid` 또는 `@Validated`로 수행된 **검증 결과(성공 또는 실패)를 보관하는 객체**
- **입력값 검증이 실패하더라도 예외를 발생시키지 않고**
  컨트롤러 내부에서 자유롭게 결과를 다룰 수 있게 해줌

🔄 2. 동작 구조

```
@PostMapping("/user")
public String saveUser(@Valid UserDto dto, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        // 검증 실패 시 수행 로직
    }
    // 검증 통과 시 수행 로직
}
```

> ✅ **`@Valid`와 `BindingResult`는 항상 연속해서 붙어야 동작**
> → 파라미터 순서: `@Valid` → `BindingResult`

🧪 3. 기본 사용 예

DTO:

```
public class UserDto {
    @NotBlank
    private String name;

    @Min(18)
    private int age;
}
```

컨트롤러:

```
@PostMapping("/users")
public String createUser(@Valid @RequestBody UserDto dto, BindingResult result) {
    if (result.hasErrors()) {
        for (FieldError error : result.getFieldErrors()) {
            System.out.println("에러 필드: " + error.getField());
            System.out.println("메시지: " + error.getDefaultMessage());
        }
        return "error";
    }
    return "ok";
}
```

⚠️ 4. 예외 vs BindingResult

조건	동작
`@Valid`만 있을 때	검증 실패 시 → 예외 발생 (`MethodArgumentNotValidException`)
`@Valid` + `BindingResult`	검증 실패 시 → 예외 없음, `bindingResult.hasErrors() == true`

→ 즉, `BindingResult`가 있으면 컨트롤러에서 **직접 에러 처리**할 수 있고
없으면 **Spring이 예외를 던짐** → 예외 처리기(`@ExceptionHandler`)로 넘김

📋 5. 유용한 메서드

메서드	설명
`hasErrors()`	에러가 하나라도 있으면 true
`hasFieldErrors("name")`	특정 필드에 에러가 있는지 확인
`getAllErrors()`	전체 에러 목록 (`ObjectError` 리스트)
`getFieldErrors()`	필드 검증 실패 목록 (`FieldError` 리스트)
`getGlobalErrors()`	전체 객체에 대한 에러
`getFieldValue("name")`	바인딩된 필드의 실제 값
`rejectValue("name", "error.code")`	수동으로 에러 등록 가능

🧠 6. 실무 패턴: 에러 메시지 출력

```
for (FieldError fieldError : result.getFieldErrors()) {
    String field = fieldError.getField();
    String message = fieldError.getDefaultMessage();
    model.addAttribute(field + "Error", message);
}
```

🔍 7. `@ModelAttribute`와도 호환

```
@PostMapping("/register")
public String register(@ModelAttribute @Valid UserForm form,
                       BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return "registerForm";
    }
    ...
}
```

→ `@RequestParam` 또는 `@ModelAttribute` 기반 폼 바인딩에서도 사용 가능

📜 8. API 응답용 전략 (JSON)

```
@PostMapping("/api/users")
public ResponseEntity<?> save(@Valid @RequestBody UserDto dto, BindingResult
result) {
    if (result.hasErrors()) {
        List<String> messages = result.getFieldErrors().stream()
            .map(FieldError::getDefaultMessage)
            .collect(Collectors.toList());
        return ResponseEntity.badRequest().body(messages);
    }
    return ResponseEntity.ok("success");
}
```

⚠️ 9. 주의사항

항목	설명
순서 중요 (`@Valid`, `BindingResult`)	반드시 `@Valid` 다음 파라미터에 위치해야 함
`BindingResult` 없으면 예외 발생	→ `@ControllerAdvice`로 처리 필요
`ModelAttribute` 기반일 때만 동작	`@RequestBody` 시에도 가능 (단, `BindingResult` 필요)
바인딩 오류 = HTTP 400 방지	BindingResult 사용 시 → 정상 200 응답 가능

📌 10. 요약

항목	설명
정의	입력값 검증 결과를 저장하는 객체
동작 조건	`@Valid` 또는 `@Validated` 바로 다음 위치

항목	설명
예외 vs 수동 처리	없음 → 컨트롤러 내에서 에러 직접 처리 가능
주요 메서드	`hasErrors()`, `getFieldErrors()`, `rejectValue()` 등
실무 전략	폼 검증, REST API 에러 응답, 커스텀 메시지 출력 등에서 필수 사용

예외 처리

- ## `@ExceptionHandler`, `@ControllerAdvice`

`@ExceptionHandler`와 `@ControllerAdvice`는 Spring MVC에서 **예외를 전역적으로, 또는 개별적으로 처리**할 수 있도록 도와주는 강력한 기능이다.
컨트롤러 코드 안에서 try-catch 없이도, 예외 상황을 **우아하게 처리하고 HTTP 응답까지 커스터마이징**할 수 있게 해준다.

다음은 두 어노테이션의 **개념, 역할 차이, 동작 원리, 사용 예제, 우선순위, REST API 처리 전략, 실무 확장 방식**까지 포함한 가장 깊이 있는 설명이다.

📘 1. 기본 정의

어노테이션	설명
`@ExceptionHandler`	특정 컨트롤러 클래스 내에서 **해당 예외를 직접 처리**
`@ControllerAdvice`	**전역 예외 처리기 역할**, 모든 컨트롤러에서 발생한 예외 처리 가능

🧱 2. `@ExceptionHandler`

✅ 정의

- 컨트롤러 클래스 내부에서 **특정 예외 타입을 처리하는 메서드에 부착**

✅ 예제

```
@Controller
public class UserController {

    @GetMapping("/user/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.findById(id); // 예외 발생 가능
    }

    @ExceptionHandler(UserNotFoundException.class)
    public String handleUserNotFound(UserNotFoundException ex, Model model) {
        model.addAttribute("message", ex.getMessage());
        return "error/user-not-found";
    }
```

```
}
```

- 위 예외는 **UserController 내에서만 작동**
- **뷰 이름 반환, 상태 코드 지정 불가**

🧭 3. `@ControllerAdvice`

✅ 정의

- 하나의 클래스에서 **모든 컨트롤러의 예외를 전역적으로 처리**하도록 구성
- `@RestControllerAdvice`는 JSON 응답에 특화된 버전

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<List<String>>
handleValidation(MethodArgumentNotValidException ex) {
        List<String> errors = ex.getBindingResult().getFieldErrors()
            .stream().map(FieldError::getDefaultMessage).toList();
        return ResponseEntity.badRequest().body(errors);
    }
}
```

🔄 4. 동작 흐름 요약

```
Controller 실행 중 예외 발생
  ↓
ExceptionResolver 찾음
  ↓
@ExceptionHandler 메서드 탐색
  ↓
→ @Controller 클래스 내부 우선
→ 없으면 @ControllerAdvice 내부 탐색
  ↓
응답 반환 (뷰 or JSON)
```

⚙️ 5. HTTP 상태 코드 제어

방법 1: `ResponseEntity`

```
@ExceptionHandler(NotFoundException.class)
public ResponseEntity<String> handle(NotFoundException ex) {
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
}
```

방법 2: `@ResponseStatus`

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class NotFoundException extends RuntimeException {
}
```

→ 예외 자체에 상태 코드를 부여

📦 6. REST API용 전역 예외 처리기

```
@RestControllerAdvice
public class ApiExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ApiError> handleAll(Exception ex) {
        return ResponseEntity.internalServerError()
            .body(new ApiError("서버 내부 오류", ex.getMessage()));
    }
}
```

- `@RestControllerAdvice` = `@ControllerAdvice + @ResponseBody`
- JSON 형식의 오류 응답에 적합

📄 7. 실무 패턴: 커스텀 에러 응답 객체

```
public class ErrorResponse {
    private String error;
    private String message;
    private LocalDateTime timestamp;
}
```

```
@ExceptionHandler(CustomException.class)
public ResponseEntity<ErrorResponse> handle(CustomException e) {
    return ResponseEntity.badRequest().body(
        new ErrorResponse("BAD_REQUEST", e.getMessage(), LocalDateTime.now()));
}
```

🔧 8. 우선순위 및 주의사항

항목	설명
컨트롤러 내 `@ExceptionHandler` 우선	그 클래스 내에서만 작동, 전역보다 먼저 실행
`@ControllerAdvice`는 전역 적용	모든 컨트롤러 대상
`@ExceptionHandler(Exception.class)`	모든 예외 잡음 → 마지막에 위치
`BindingResult`와 충돌 주의	검증 실패 시 `@ExceptionHandler`보다 `BindingResult`가 먼저 처리됨

📌 9. 요약

항목	`@ExceptionHandler`	`@ControllerAdvice` / `@RestControllerAdvice`
적용 범위	해당 컨트롤러 클래스	모든 컨트롤러 (전역)
목적	특정 예외 처리	글로벌 예외 관리
응답 형식	뷰 또는 ResponseEntity	JSON or ResponseEntity
상태 코드 제어	`ResponseEntity`로 제어	동일
REST API 추천 형태	❌ 비추천	✅ `@RestControllerAdvice` 사용

파일 업로드 처리

Spring MVC에서의 **파일 업로드 처리**는 `MultipartFile`을 활용하여 **클라이언트가 보낸 파일 데이터를 서버 측에서 수신, 저장, 검증**하는 과정을 포함한다.
Spring Boot를 사용하면 설정 없이도 **multipart/form-data** 요청을 자동 처리할 수 있으며,
작은 파일부터 대용량 파일까지 확장성 있게 처리할 수 있다.

다음은 **파일 업로드의 개념, 요청 구조, MultipartFile 사용법, 단일/다중 업로드, 저장 전략, 예외 처리, 보안 고려사항**까지 포함한
가장 깊이 있는 설명이다.

📘 1. 개념 요약

항목	설명
업로드 방식	HTML form (`enctype="multipart/form-data"`)
서버 수신 방식	Spring의 `MultipartFile` 인터페이스 사용
처리 엔진	`CommonsMultipartResolver` (과거) → Spring Boot에서는 자동 내장
의존성	`spring-boot-starter-web`만 있으면 기본 처리 가능

🔧 2. application 설정

```
spring:
  servlet:
    multipart:
      max-file-size: 10MB
      max-request-size: 20MB
      enabled: true
      location: /tmp
```

→ 최대 업로드 크기 제한 가능 / 디스크 임시 저장소 지정

🧪 3. 단일 파일 업로드 처리 예제

HTML Form

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

컨트롤러

```
@PostMapping("/upload")
public String handleUpload(@RequestParam("file") MultipartFile file) throws IOException
{
    if (file.isEmpty()) {
        return "파일이 비어 있습니다.";
    }

    // 원본 파일명
    String originalFilename = file.getOriginalFilename();

    // 저장 경로 설정
    Path savePath = Paths.get("uploads", originalFilename);

    // 파일 저장
    Files.copy(file.getInputStream(), savePath, StandardCopyOption.REPLACE_EXISTING);

    return "업로드 성공";
}
```

📦 4. 다중 파일 업로드 처리

HTML Form

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

컨트롤러

```
@PostMapping("/upload/multi")
public String handleMultiUpload(@RequestParam("files") List<MultipartFile> files)
throws IOException {
    for (MultipartFile file : files) {
        if (!file.isEmpty()) {
            Path path = Paths.get("uploads", file.getOriginalFilename());
            Files.copy(file.getInputStream(), path,
StandardCopyOption.REPLACE_EXISTING);
        }
    }
    return "여러 파일 업로드 성공";
}
```

📁 5. 저장 위치 전략

방식	설명
로컬 디스크 저장	`Files.copy()` 또는 `transferTo(File)` 활용
DB BLOB 저장	`byte[]`로 추출 → DB에 저장 (비효율적이므로 제한적 사용)
클라우드 연동	AWS S3, Google Cloud Storage, FTP 등과 연동
날짜/UUID 분리	`2025/05/23/{uuid}.ext` 구조 추천

🛠️ 6. 파일 정보 추출 API

메서드	설명
`getOriginalFilename()`	클라이언트가 업로드한 파일명
`getSize()`	파일 크기 (byte)
`getContentType()`	MIME 타입 (`image/png`, `application/pdf` 등)
`getInputStream()`	스트림 추출
`transferTo(File)`	직접 저장

🚫 7. 파일 업로드 보안 주의사항

항목	설명
파일 확장자 검사	`.exe`, `.sh`, `.jsp` 등 금지
MIME 타입 검사	`image/*`, `application/pdf` 등 제한
저장 경로 검증	외부로 노출된 경로에 직접 저장 ❌
경로 탐색 방지	`../` 같은 상대 경로 사용 금지
UUID 파일명 사용	원본 파일명 사용 시 충돌 및 보안 문제 가능
업로드 후 스캔	이미지 검증 등 백신 연동 가능 (대규모 시스템)

✅ 8. 실무 전략

상황	전략
일반 업로드 (이미지, 문서 등)	MultipartFile + 로컬 저장
REST API 업로드	JSON + base64 또는 multipart/form-data
클라우드 저장소 사용	AWS S3 SDK 연동, File → InputStream 변환
썸네일, 리사이징 필요	업로드 후 `Thumbnailator`, `ImageIO` 등 후처리
게시글 첨부파일 등	DB에 `uuid`, `originalFilename`, `path` 저장하고 파일은 로컬 저장

📌 9. 요약

항목	설명
업로드 처리 기본	`MultipartFile` 사용
요청 형식	`multipart/form-data`
저장 방식	로컬 디스크 or 클라우드
실무 전략	UUID + 날짜 디렉토리 분리, 확장자 제한 등 보안 처리 필수
Spring Boot 설정	`spring.servlet.multipart.*` 자동 설정 지원

국제화(i18n)

Spring의 국제화(i18n, Internationalization)는 애플리케이션에서 다국어 메시지를 지원하기 위한 기능이다.
사용자의 **Locale(언어/국가 설정)**에 따라 **메시지 번역**, **날짜/숫자 포맷**, **뷰 처리 방식** 등을 자동으로 조정할 수 있게 해준다.

다음은 Spring 국제화(i18n)의 **개념, 메시지 리소스 설정, `MessageSource` 사용법, LocaleResolver 전략, 다국어 컨트롤러 처리, 실무 구조**까지 포함한 가장 깊이 있는 설명이다.

📘 1. 국제화(i18n)란?

- **국제화(i18n)**: 다국어를 지원하도록 시스템을 설계하는 과정 (i + 18글자 + n)
- **지역화(l10n)**: 특정 언어/국가에 맞춰 콘텐츠를 실제로 번역 및 적용하는 것

🧱 2. 메시지 프로퍼티 파일 구조

파일명	대상 언어/지역
`messages.properties`	기본 (영어 또는 설정 없음)
`messages_ko.properties`	한국어
`messages_en.properties`	영어
`messages_ja.properties`	일본어

예: `messages_ko.properties`

```
greeting=안녕하세요
username.label=사용자 이름
```

예: `messages_en.properties`

```
greeting=Hello
username.label=Username
```

🔧 3. Spring Boot 국제화 설정

```yaml
spring:
  messages:
    basename: messages   # messages_ko.properties, messages_en.properties 등
    encoding: UTF-8
```

- 기본 위치: `src/main/resources/messages_*.properties`
- 다중 파일: `basename: messages,errors,labels`

🧩 4. Locale 설정 전략

Spring에서는 사용자의 언어 정보를 **LocaleResolver**를 통해 결정함.

✅ 1. `AcceptHeaderLocaleResolver` (기본값)

- 브라우저의 `Accept-Language` 헤더에 따라 자동 설정

✅ 2. `SessionLocaleResolver`

- 세션에 Locale 저장

✅ 3. `CookieLocaleResolver`

- 쿠키로 사용자 Locale 기억

✅ 등록 방법 (Java Config)

```
@Bean
public LocaleResolver localeResolver() {
    SessionLocaleResolver resolver = new SessionLocaleResolver();
    resolver.setDefaultLocale(Locale.KOREA);
    return resolver;
}
```

🌐 5. Locale 변경 방법

✅ 웹에서 언어 변경

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor interceptor = new LocaleChangeInterceptor();
    interceptor.setParamName("lang"); // ?lang=ko
    return interceptor;
}
```

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(localeChangeInterceptor());
}
```

→ `/main?lang=ko`처럼 요청하면 언어 변경 가능

🧪 6. 메시지 출력 방식

✅ Thymeleaf

```
<p th:text="#{greeting}">기본 인사</p>
```

✅ JSP

```
<spring:message code="greeting"/>
```

✅ Java 코드 (Service, Controller)

```
@Autowired
private MessageSource messageSource;

public String getMessage(Locale locale) {
    return messageSource.getMessage("greeting", null, locale);
}
```

🧠 7. 메시지 포맷팅

✅ 메시지 파일

```
welcome=안녕하세요, {0}님! 오늘은 {1}입니다.
```

✅ 코드에서 사용

```
messageSource.getMessage("welcome", new Object[]{"정석", "금요일"}, locale)
→ "안녕하세요, 정석님! 오늘은 금요일입니다."
```

🛠️ 8. 실무 전략

항목	설명
다국어 키 관리	`messages.properties`로 통합 관리 (prefix 기준 그룹화)
영어 → 한국어 → 중국어 순 적용	LocaleResolver + fallback 전략 사용
텍스트 외에도 **날짜/숫자 포맷** 지원	`@DateTimeFormat`, `NumberFormat`과 연계 가능
사용자 언어 변경 지원	`LocaleChangeInterceptor` + 쿼리 파라미터
Spring Security와 연계 시 주의	인증 전 Locale 설정은 Cookie/Session 기반 권장

✅ 9. 요약

항목	설명
메시지 리소스 파일	`messages_{locale}.properties`
메시지 출력 방법	Thymeleaf: `#{code}` / Java: `MessageSource`
Locale 결정 방식	기본: `AcceptHeader`, 커스텀: Session/Cookie
언어 변경 방법	쿼리스트링 (`?lang=ko`) + Interceptor
실무 전략	prefix 그룹화, 다국어 키 네이밍 규칙화, JSON 동적 메시지 대응 가능