28. 실무 프레임워크 연동

Spring Core, Spring Boot 기초

1. Spring Core 개요

1.1 Spring Framework란?

- **Java 객체 간의 결합을 느슨하게 유지하면서, 생산성과 테스트 가능성을 높이기 위한 애플리케이션 프레임워크**
- 주요 특징:
  - IoC(제어의 역전)
  - DI(의존성 주입)
  - AOP(관점 지향 프로그래밍)
  - 트랜잭션 관리
  - 테스트 지원
  - 모듈화된 구조

1.2 Spring Core의 주요 개념

개념	설명
IoC (Inversion of Control)	객체 생성과 관리 책임을 개발자가 아닌 컨테이너가 수행
DI (Dependency Injection)	객체가 스스로 의존 객체를 생성하지 않고 외부로부터 주입 받음
Bean	Spring이 관리하는 객체
ApplicationContext	Bean 생성 및 의존성 주입을 담당하는 Spring의 핵심 컨테이너

1.3 간단한 예제

```
@Component
public class UserService {
    public void hello() {
        System.out.println("Hello, Spring");
    }
}
```

```java
@Configuration
@ComponentScan
public class AppConfig {}

public class Main {
    public static void main(String[] args) {
        ApplicationContext ctx = new
AnnotationConfigApplicationContext(AppConfig.class);
        UserService us = ctx.getBean(UserService.class);
        us.hello();
    }
}
```

---

2. Spring Boot 개요

2.1 Spring Boot란?

- Spring Core의 강력한 기능을 **설정 없이 빠르게 사용할 수 있도록 만든 개발 플랫폼**
- 특징:
  - **자동 설정(Auto Configuration)**
  - 내장 서버(Tomcat, Jetty)
  - 실행 가능한 JAR로 패키징
  - RESTful API, JPA, Security 등 통합 지원

2.2 Spring Boot의 핵심 구성 요소

구성 요소	설명
Spring Initializr	웹 기반 프로젝트 생성기
`@SpringBootApplication`	주요 애노테이션 묶음 (`@Configuration`, `@EnableAutoConfiguration`, `@ComponentScan`)
의존성 관리	`spring-boot-starter-*`로 손쉬운 설정
내장 서버	Jetty, Undertow, Tomcat 내장 가능
실행 방법	`main()` → `SpringApplication.run(...)` 호출

2.3 최소 예제

```
@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

2.4 주요 Starter

Starter	설명
`spring-boot-starter-web`	MVC + Tomcat + Jackson 등
`spring-boot-starter-data-jpa`	JPA + Hibernate 설정
`spring-boot-starter-security`	Spring Security 설정
`spring-boot-starter-test`	JUnit, Mockito, AssertJ 포함 테스트 환경
`spring-boot-devtools`	핫 리로드 기능

2.5 Spring Boot 자동 설정 원리

- `@EnableAutoConfiguration`에 의해
  - `spring.factories` / `META-INF/spring/org.springframework.boot.autoconfigure.EnableAutoConfiguration` 탐색
  - 클래스패스에 존재하는 라이브러리를 감지하여 자동 설정

```
// 내부적으로 이런 설정을 자동 등록
@Bean
public DispatcherServlet dispatcherServlet() {
    return new DispatcherServlet();
}
```

---

3. Spring Core vs Spring Boot 비교

항목	Spring Core	Spring Boot
설정 방식	XML, Java Config	최소 설정 (Convention 기반)
서버 설정	외부 WAS 필요	내장 WAS 제공
의존성 구성	직접 설정	Starter로 자동 관리
실행 방식	수동 `main()` 또는 WAR 배포	`SpringApplication.run()`

4. 프로젝트 구조 예시

```
src/
├─ main/
|  ├─ java/com/example/
|  |  ├─ Application.java
|  |  ├─ controller/
|  |  ├─ service/
|  |  └─ repository/
|  └─ resources/
|     ├─ application.yml
|     └─ static/
```

5. 실무에서의 Spring Boot 활용 시나리오

목적	기능
RESTful API 개발	`@RestController`, `@GetMapping`
데이터 연동	Spring Data JPA, JDBC, MyBatis
보안	Spring Security
테스트	`@SpringBootTest`, `MockMvc`
API 문서화	Swagger, SpringDoc
배포	Docker + Spring Boot JAR 실행

6. 학습 추천 순서

1. IoC / DI → Bean 등록
2. ApplicationContext의 역할
3. Spring Boot 프로젝트 생성
4. REST API 개발 (`@RestController`)
5. 데이터베이스 연동 (JPA)
6. 예외 처리, Validation
7. Security, 인증/인가
8. 테스트, 로깅, 배포

Hibernate / JPA 연동

1. 전체 흐름 요약

```
@Entity 클래스 정의
    ↓
JpaRepository 인터페이스 작성
    ↓
@Service에서 @Transactional로 트랜잭션 처리
    ↓
컨트롤러 또는 서비스 테스트
```

---

2. Spring Boot 프로젝트 기본 구성

2.1 필수 의존성 (`build.gradle` or `pom.xml`)

```
// Gradle 예시
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    runtimeOnly 'com.h2database:h2' // 개발용 DB (MySQL 등으로 변경 가능)
}
```

2.2 `application.yml` 설정 예시

```
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password:
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        format_sql: true
```

---

3. 엔티티(Entity) 매핑

3.1 기본 엔티티

```
@Entity
public class Member {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;

    private String name;

    private int age;

    // 기본 생성자 필수
    protected Member() {}

    public Member(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

3.2 주요 매핑 애노테이션

애노테이션	설명
`@Entity`	JPA에서 관리할 객체
`@Id`	PK 지정
`@GeneratedValue`	자동 생성 전략
`@Column`	컬럼 속성 설정
`@Table(name = "...")`	테이블명 명시

---

4. Repository 구성

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    List<Member> findByName(String name);
}
```

계층	역할
`JpaRepository`	기본 CRUD 자동 생성
커스텀 메서드	이름 기반 쿼리 자동 해석 (`findByX`)
`@Query`	직접 JPQL 지정 가능

---

5. Service 계층에서 사용

```
@Service
@RequiredArgsConstructor
public class MemberService {

```

```
    private final MemberRepository memberRepository;

    @Transactional
    public void register(String name, int age) {
        memberRepository.save(new Member(name, age));
    }

    public List<Member> getByName(String name) {
        return memberRepository.findByName(name);
    }
}
```

- `@Transactional` 은 **쓰기 로직 필수**
- 조회용은 `@Transactional(readOnly = true)` 권장 (성능 최적화)

---

6. 연관 관계 매핑 예시

```
@Entity
public class Team {
    @Id @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(mappedBy = "team", cascade = CascadeType.ALL)
    private List<Member> members = new ArrayList<>();
}

@Entity
public class Member {
    ...
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "team_id")
    private Team team;
}
```

> ⚠️ 실무에서는 항상 `@ManyToOne(fetch = FetchType.LAZY)` 사용해야 함

---

7. 쿼리 전략

방식	예시
메서드 이름 기반	`findByAgeGreaterThan(int age)`
JPQL	`@Query("select m from Member m where m.name = :name")`
Native SQL	`@Query(value = "SELECT * FROM member", nativeQuery = true)`
Pageable 지원	`Page<Member> findByName(String name, Pageable pageable)`

8. 엔티티 생명주기 요약

상태	설명
비영속	`new` 상태 (JPA가 관리 안 함)
영속	`em.persist(entity)` 후 관리됨
준영속	`em.detach(entity)` 로 관리 중단
삭제	`em.remove(entity)`

9. 성능 최적화 포인트

항목	설명
Lazy Loading	`@ManyToOne(fetch = FetchType.LAZY)` 기본 적용
Fetch Join	`@Query("select m from Member m join fetch m.team")`
DTO 조회	엔티티 직접 노출 대신 DTO 변환
1차 캐시	같은 트랜잭션 내 동일 객체는 캐시로 반환됨
N+1 해결	`EntityGraph`, `fetch join`, `BatchSize` 조합 활용

10. DDL 자동 생성 전략

옵션	설명
`none`	자동 생성 안함 (실무 추천)
`create`	시작 시 테이블 생성
`update`	테이블 변경 자동 반영
`create-drop`	시작 시 생성, 종료 시 제거
`validate`	구조 일치 여부만 검사 (강력 추천)

11. 실전 구성 예시

```
domain/
├─ entity/
|  └─ Member.java
├─ repository/
|  └─ MemberRepository.java
├─ service/
|  └─ MemberService.java
└─ controller/
   └─ MemberController.java
```

---

12. 테스트 예시

```
@SpringBootTest
@Transactional
class MemberServiceTest {

    @Autowired
    MemberService memberService;

    @Test
    void testRegister() {
        memberService.register("Jane", 25);
        List<Member> found = memberService.getByName("Jane");
        assertThat(found).hasSize(1);
    }
}
```

RESTful API 구현

1. REST란 무엇인가?

- **Representational State Transfer**
- 자원을 URI로 표현하고, HTTP Method로 **의미 있는 행위(CRUD)**를 표현하는 아키텍처 스타일

메서드	의미
`GET /users`	사용자 목록 조회
`GET /users/1`	특정 사용자 조회
`POST /users`	사용자 등록
`PUT /users/1`	사용자 전체 수정
`PATCH /users/1`	사용자 일부 수정
`DELETE /users/1`	사용자 삭제

2. 프로젝트 구조 예시

```
src/
├─ controller/
|  └─ UserController.java
├─ dto/
|  └─ UserDto.java
├─ entity/
|  └─ User.java
├─ repository/
|  └─ UserRepository.java
├─ service/
|  └─ UserService.java
└─ exception/
   └─ GlobalExceptionHandler.java
```

---

3. REST 컨트롤러 기본 구성

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

    private final UserService userService;

    @GetMapping
    public List<UserDto> getUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public UserDto getUser(@PathVariable Long id) {
        return userService.getUserById(id);
    }

    @PostMapping
    public ResponseEntity<UserDto> createUser(@RequestBody UserDto dto) {
        UserDto saved = userService.createUser(dto);
        return ResponseEntity.status(HttpStatus.CREATED).body(saved);
    }

    @PutMapping("/{id}")
    public UserDto updateUser(@PathVariable Long id, @RequestBody UserDto dto) {
        return userService.updateUser(id, dto);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
    }
```

```
}
```

4. DTO 정의 예시

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class UserDto {
    private Long id;
    private String name;
    private String email;
}
```

5. Service 구현

```
@Service
@RequiredArgsConstructor
public class UserService {

    private final UserRepository userRepository;

    public List<UserDto> getAllUsers() {
        return userRepository.findAll().stream()
            .map(UserDto::fromEntity)
            .toList();
    }

    public UserDto getUserById(Long id) {
        return userRepository.findById(id)
            .map(UserDto::fromEntity)
            .orElseThrow(() -> new NotFoundException("User not found"));
    }

    public UserDto createUser(UserDto dto) {
        User entity = new User(dto.getName(), dto.getEmail());
        return UserDto.fromEntity(userRepository.save(entity));
    }

    public UserDto updateUser(Long id, UserDto dto) {
        User user = userRepository.findById(id)
            .orElseThrow(() -> new NotFoundException("User not found"));
        user.setName(dto.getName());
        user.setEmail(dto.getEmail());
        return UserDto.fromEntity(userRepository.save(user));
    }

    public void deleteUser(Long id) {
        if (!userRepository.existsById(id)) {
            throw new NotFoundException("User not found");
```

```
        }
        userRepository.deleteById(id);
    }
}
```

6. Entity 정의

```
@Entity
@Getter @Setter
@NoArgsConstructor
public class User {

    @Id @GeneratedValue
    private Long id;

    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }
}
```

7. DTO ↔ Entity 변환 예시

```
public class UserDto {
    ...

    public static UserDto fromEntity(User user) {
        return new UserDto(user.getId(), user.getName(), user.getEmail());
    }
}
```

8. 예외 처리 (GlobalExceptionHandler)

```
@RestControllerAdvice
public class GlobalExceptionHandler {

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

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<String> handleValidation(MethodArgumentNotValidException ex)
{
        return ResponseEntity.badRequest().body("Validation failed");
    }
}
```

---

9. 응답 구조 예시 (표준화)

```
public class ApiResponse<T> {
    private boolean success;
    private T data;
    private String message;

    public static <T> ApiResponse<T> ok(T data) {
        return new ApiResponse<>(true, data, null);
    }

    public static ApiResponse<?> error(String message) {
        return new ApiResponse<>(false, null, message);
    }
}
```

---

10. 테스트 예시 (`@WebMvcTest`)

```
@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    void testGetUser() throws Exception {
        UserDto dto = new UserDto(1L, "Kim", "kim@example.com");
        given(userService.getUserById(1L)).willReturn(dto);
```

```
14
15          mockMvc.perform(get("/api/users/1"))
16              .andExpect(status().isOk())
17              .andExpect(jsonPath("$.name").value("Kim"));
18      }
19  }
```

---

11. REST 설계 실무 팁

항목	권장 방식
URL 구조	`/api/resources/{id}`
동사 사용 금지	`POST /login`, `DELETE /users/1` ← ok / `POST /deleteUser` ← bad
상태 코드	200 OK, 201 Created, 404 Not Found 등 정확히 사용
입력 검증	`@Valid`, `@Validated`, `BindingResult`
예외 메시지	일관된 포맷으로 반환
응답 포맷	`ApiResponse<T>` 같은 공통 구조 사용

---

12. 향후 확장 주제

- `@Validated`를 통한 입력 검증
- Swagger 연동 (`springdoc-openapi`)
- REST Docs 통한 API 문서 자동화
- JWT 인증 기반 보호된 REST API
- Spring Security + OAuth2 적용

Jackson을 이용한 JSON 처리

1. Jackson이란?

- **자바 객체 ↔ JSON 간의 변환(직렬화/역직렬화)** 을 지원하는 고성능 라이브러리
- Spring Boot에서 **기본으로 통합됨**
- 핵심 모듈:
  - `jackson-databind`: JSON ↔ 객체 매핑
  - `jackson-core`: 파서/제너레이터
  - `jackson-annotations`: 애노테이션 기능 제공

---

2. 기본 예제 (Object ↔ JSON)

```
ObjectMapper objectMapper = new ObjectMapper();

User user = new User("Jane", "jane@example.com");
String json = objectMapper.writeValueAsString(user); // 객체 → JSON
User parsed = objectMapper.readValue(json, User.class); // JSON → 객체
```

```
{
  "name": "Jane",
  "email": "jane@example.com"
}
```

---

3. 주요 애노테이션

애노테이션	설명
`@JsonProperty("jsonFieldName")`	JSON 필드 이름 지정
`@JsonIgnore`	직렬화/역직렬화에서 제외
`@JsonInclude`	특정 조건일 때만 포함 (`NON_NULL`, `NON_EMPTY`)
`@JsonFormat`	날짜/시간 포맷 지정
`@JsonAlias`	여러 JSON 필드 이름 허용
`@JsonAnyGetter`, `@JsonAnySetter`	맵과 같은 동적 필드 지원
`@JsonCreator`, `@JsonValue`	커스텀 생성자/값 지정

예시: 애노테이션 사용

```
public class User {

    @JsonProperty("user_name")
    private String name;

    @JsonIgnore
    private String password;

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

---

4. Jackson과 Spring 연동 구조

Spring 사용 방식	내부 동작
`@RequestBody`	JSON → 객체 (역직렬화)
`@ResponseBody`	객체 → JSON (직렬화)
`RestTemplate`	자동 변환 (Jackson 사용)

```
@PostMapping("/user")
public ResponseEntity<?> saveUser(@RequestBody UserDto dto) {
    // JSON → 객체 자동 변환됨
    ...
    return ResponseEntity.ok(dto); // 객체 → JSON 응답
}
```

---

5. 날짜/시간 처리 (Java 8 `java.time` 지원)

설정 방법 1: `@JsonFormat`

```
@JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss")
private LocalDateTime createdAt;
```

설정 방법 2: `ObjectMapper` 전역 설정

```
objectMapper.registerModule(new JavaTimeModule());
objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
```

---

6. 컬렉션, 맵, 제네릭 처리

```
List<User> list = objectMapper.readValue(json, new TypeReference<List<User>>() {});
Map<String, User> map = objectMapper.readValue(json, new TypeReference<Map<String,
User>>() {});
```

---

7. 직렬화 컨트롤

기능	메서드
객체 → JSON 문자열	`writeValueAsString(obj)`
객체 → 파일 출력	`writeValue(new File("a.json"), obj)`
JSON → 객체	`readValue(json, Class)`
JSON → 컬렉션	`readValue(json, new TypeReference<List<T>>() {})`

8. 동적 JSON 처리 (`JsonNode`, Tree Model)

```
JsonNode node = objectMapper.readTree(json);
String name = node.get("name").asText();
```

> 구조가 고정되지 않은 JSON 파싱에 유용

---

9. 커스텀 직렬화/역직렬화

커스텀 직렬화기

```
public class MaskingSerializer extends JsonSerializer<String> {
    @Override
    public void serialize(String value, JsonGenerator gen, SerializerProvider
serializers) throws IOException {
        gen.writeString("****");
    }
}
```

```
public class User {
    @JsonSerialize(using = MaskingSerializer.class)
    private String ssn;
}
```

---

10. Spring Boot에서 전역 설정

```
@Configuration
public class JacksonConfig {
    @Bean
    public ObjectMapper objectMapper() {
        ObjectMapper om = new ObjectMapper();
        om.registerModule(new JavaTimeModule());
        om.setSerializationInclusion(JsonInclude.Include.NON_NULL);
        return om;
    }
}
```

또는 `application.yml`에서 날짜 포맷 설정:

```
spring:
  jackson:
    date-format: yyyy-MM-dd HH:mm:ss
    property-naming-strategy: SNAKE_CASE
```

---

11. 보안 실무 팁

항목	설명
민감 정보 제거	`@JsonIgnore`, DTO로 전환
JSON 파싱 오류 처리	`@ControllerAdvice`에서 `HttpMessageNotReadableException` 처리
무분별한 Jackson 사용 제한	구조 명확한 DTO 사용, JsonNode 남용 금지
성능 최적화	`@JsonView`, ViewModel 전략으로 데이터 필터링

---

12. Jackson 대체 가능성

대안	특징
Gson	Google 개발, 속도는 느리지만 유연
Moshi	Square 개발, Android 최적화
JSON-B	Java EE 공식 JSON 바인딩 API

Spring Boot는 기본적으로 **Jackson**에 최적화되어 있으므로 특별한 이유 없으면 그대로 사용하는 게 좋음.

Java + Docker 배포

1. 전체 흐름 요약

```
① 프로젝트 빌드 (.jar 생성)
      ↓
② Dockerfile 작성
      ↓
③ Docker 이미지 빌드
      ↓
④ Docker 컨테이너 실행
      ↓
⑤ 외부 포트 연결 / 환경변수 구성
      ↓
⑥ Docker Hub / 클라우드에 배포
```

---

2. 프로젝트 빌드 (JAR 생성)

Gradle

```
./gradlew build
```

생성 결과:

```
build/libs/myapp-0.0.1-SNAPSHOT.jar
```

Maven

```
./mvnw package
```

---

3. Dockerfile 작성

✅ 기본형

```
FROM openjdk:17-jdk-alpine
ARG JAR_FILE=build/libs/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

> `openjdk:17-jdk-alpine`은 경량 JDK 베이스 이미지
> `COPY`로 `.jar` 파일 복사
> `ENTRYPOINT`는 실행 명령어

---

4. `.dockerignore` 설정 (필수)

```
*.iml
*.class
.gradle
target/
build/
```

> Docker context에서 불필요한 파일 제외

---

5. 이미지 빌드

```
docker build -t myapp:latest .
```

> `myapp`이라는 이름으로 Docker 이미지 생성됨

---

6. 컨테이너 실행

```
docker run -d -p 8080:8080 --name myapp-container myapp
```

옵션	설명
`-d`	백그라운드 실행

옵션	설명
`-p 8080:8080`	로컬 포트:컨테이너 포트 연결
`--name`	컨테이너 이름 지정

---

7. 환경 변수 적용

```
ENV SPRING_PROFILES_ACTIVE=prod
ENV DB_URL=jdbc:mysql://...
```

또는 실행 시:

```
docker run -e SPRING_PROFILES_ACTIVE=dev ...
```

---

8. Docker Compose 활용 (선택)

```
docker-compose.yml
```

```
version: '3'
services:
  app:
    image: myapp:latest
    ports:
      - "8080:8080"
    environment:
      SPRING_PROFILES_ACTIVE: prod
```

```
docker-compose up -d
```

---

9. 경량화 최적화 (2단계 빌드)

```
# 1단계: Build
FROM gradle:8.2-jdk17 AS builder
COPY . /home/app
WORKDIR /home/app
RUN gradle build --no-daemon

# 2단계: Run
FROM openjdk:17-jdk-alpine
COPY --from=builder /home/app/build/libs/*.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

---

10. Docker Hub에 푸시 (선택)

```
docker tag myapp:latest username/myapp:1.0
docker push username/myapp:1.0
```

> 나중에 클라우드 서버에서 바로 `pull` 해서 실행 가능

---

11. AWS EC2, GCP 등 운영 배포 시

- EC2 + Docker 설치 후 `docker run` 으로 배포
- GitHub Actions / Jenkins와 연계한 CI/CD 자동화
- 도커 이미지 → ECS / Kubernetes로 확장 가능

---

12. 보안/운영 팁

항목	권장
JDK 경량 이미지 사용	`openjdk:17-jdk-alpine`
민감 정보	`ENV` 대신 `.env` 파일, Secrets Manager 활용
로그 관리	STDOUT + 로그 수집기(ELK, Loki 등)
헬스 체크	`/actuator/health` 활용 + `HEALTHCHECK` 명령어 설정
컨테이너 재시작 정책	`--restart unless-stopped` 설정

---

13. 실전 운영 예시 명령

```
docker build -t myapp:1.0 .
docker tag myapp:1.0 my-registry/myapp:1.0
docker push my-registry/myapp:1.0

# 원격 서버에서
docker pull my-registry/myapp:1.0
docker run -d -p 80:8080 my-registry/myapp:1.0
```

---

14. 전체 요약

단계	목적	명령
빌드	`.jar` 생성	`./gradlew build`
이미지 생성	`.jar → Docker 이미지`	`docker build`

단계	목적	명령
컨테이너 실행	포트 연결, 실행	`docker run -p ...`
경량화	멀티스테이지 빌드	`FROM builder → FROM runtime`
배포	서버, Hub, 클라우드	`push`, `pull`, `run`