22. 최신 기술 및 확장 주제

Kotlin + Spring Boot

1️⃣ Kotlin 소개

- **JetBrains** 에서 개발한 JVM 기반 프로그래밍 언어
- **간결성**, **안전성**, **함수형 프로그래밍 지원** → Java 대비 생산성 향상
- 100% **Java와 상호운용 가능**
- Spring Boot 공식적으로 Kotlin을 **1st-class citizen** 으로 지원

2️⃣ Kotlin + Spring Boot 사용 이유

장점

✅ Null-Safety → NullPointerException 방지
✅ 간결한 문법 → Boilerplate 코드 대폭 감소
✅ Data Class 지원 → DTO/VO 작성 매우 간편
✅ Extension Function 지원 → 기존 API 기능 확장 용이
✅ Coroutines 지원 → 비동기 처리 시 코드가 직관적
✅ Java 라이브러리와 100% 호환

단점

❌ Java 에 비해 컴파일 시간이 길어질 수 있음
❌ 일부 라이브러리/도구가 Kotlin 지원이 부족한 경우 있음 (점점 해결됨)

3️⃣ 프로젝트 구조 예시

```
src
└── main
    ├── kotlin
    │   └── com.example.app
    │       ├── Application.kt
    │       ├── domain
    │       ├── application
    │       ├── adapter
    │       ├── config
    ├── resources
        ├── application.yml
```

- 기존 **Java 기반 Spring Boot 구조와 동일** → 단지 Kotlin 언어 사용
- 주로 **Hexagonal Architecture / DDD 구조**와 매우 잘 어울림

4️⃣ Kotlin 문법 특징 (Spring Boot 활용시 유용한 것들)

4.1 Data Class

```
data class User(
    val id: Long,
    val name: String,
    val email: String
)
```

- equals, hashCode, toString 자동 구현 → DTO/VO 작성 매우 편리함

4.2 Null-Safety

```
val name: String = "Spring Boot"
val nullableName: String? = null
```

- 컴파일 타임에 Null 여부 체크 → NullPointerException 예방 가능

4.3 Extension Function

```
fun String.isEmail(): Boolean {
    return this.contains("@")
}
```

- 기존 클래스에 **메서드 추가** 가능 → API 확장성 높아짐

4.4 Default Parameter

```
fun greet(name: String = "World") {
    println("Hello, $name!")
}
```

- 메서드 오버로딩 필요성 감소

5️⃣ Spring Boot + Kotlin 주요 설정

5.1 build.gradle.kts 예시 (Kotlin DSL 사용)

```
plugins {
    id("org.springframework.boot") version "3.2.5"
    id("io.spring.dependency-management") version "1.1.4"
    kotlin("jvm") version "1.9.22"
    kotlin("plugin.spring") version "1.9.22"
    kotlin("plugin.jpa") version "1.9.22"
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web")
```

```
    implementation("org.springframework.boot:spring-boot-starter-data-jpa")
    implementation("org.springframework.boot:spring-boot-starter-validation")
    implementation("com.fasterxml.jackson.module:jackson-module-kotlin")
    implementation("org.jetbrains.kotlin:kotlin-reflect")
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")

    runtimeOnly("com.h2database:h2")

    testImplementation("org.springframework.boot:spring-boot-starter-test")
}
```

- `jackson-module-kotlin` → JSON 직렬화/역직렬화 지원 필수

6️⃣ Kotlin + Spring Boot 개발 스타일

6.1 Controller 예시

```
@RestController
@RequestMapping("/users")
class UserController(
    private val userService: UserService
) {

    @GetMapping("/{id}")
    fun getUser(@PathVariable id: Long): ResponseEntity<UserDto> {
        val user = userService.getUserById(id)
        return ResponseEntity.ok(user)
    }
}
```

6.2 Service 예시

```
@Service
@Transactional
class UserService(
    private val userRepository: UserRepository
) {

    fun getUserById(id: Long): UserDto {
        val user = userRepository.findById(id)
            .orElseThrow { EntityNotFoundException("User not found") }

        return UserDto(user.id, user.name, user.email)
    }
}
```

6.3 Repository 예시

```
interface UserRepository : JpaRepository<User, Long>
```

→ **Java 코드와 거의 동일**, 하지만 Kotlin 문법이 훨씬 간결하고 안전함

7️⃣ Kotlin + Spring Boot 활용시 추천 팁

✅ **Data Class 적극 활용 (DTO, VO, Response 객체 등)**
✅ **Constructor Injection** 사용 → 가독성/테스트 용이성 향상
✅ Jackson 설정 시 반드시 `jackson-module-kotlin` 추가
✅ Coroutines 를 사용할 경우 `spring-boot-starter-webflux` 활용 가능
✅ Bean Validation 시 Kotlin Null-Safety 와 충돌 없도록 주의 → `@field:NotBlank` 형태로 적용

8️⃣ Kotlin + Spring Boot 실전 적용시 패턴

상황	적용 팁
API 레이어	Data Class + ResponseEntity 사용
DDD Domain Layer	Entity/VO 에 Data Class 적극 활용 (주의: JPA Entity는 `open` 필요)
Event 처리	Coroutines + Event Handler 구성
서비스 내부	Extension Function 적극 활용
테스트	KotlinTest / Kotest 라이브러리 활용 추천

9️⃣ 결론

Kotlin + Spring Boot는:

✅ **생산성 높고**
✅ **코드 간결하고 명확하며**
✅ 기존 **Java 기반 라이브러리 활용 가능**
✅ DDD / Hexagonal / CQRS / Event Sourcing 과 **아주 잘 어울림**

그래서 많은 MSA 기반 프로젝트에서 **새로운 서비스부터 Kotlin 적용** → 점진적 마이그레이션 패턴도 많이 사용됨.

GraphQL 연동

1️⃣ GraphQL 기본 개념

정의

- Facebook에서 개발 → **API Query Language**
- 기존 REST API는 → **엔드포인트(URI)마다 고정된 리소스 제공**
- GraphQL은 → **하나의 Endpoint 에서 클라이언트가 원하는 데이터 구조로 요청 가능**

REST vs GraphQL 비교

측면	REST	GraphQL
Endpoint 수	기능별 다수	단일 Endpoint (`/graphql`)
데이터 제공 방식	고정된 Response 구조	클라이언트가 필요한 필드만 요청
Overfetching 문제	발생 (불필요한 필드 포함됨)	발생하지 않음 (정확히 요청한 데이터만 반환)
Underfetching 문제	발생 (추가 요청 필요)	발생하지 않음 (복합 구조로 한 번에 조회 가능)
API Evolution	Versioning 필요	Schema Evolution으로 대응 가능

---

2 GraphQL 주요 개념

개념	설명
Schema	GraphQL 서버가 제공하는 데이터 구조 정의
Query	데이터를 조회하는 요청
Mutation	데이터를 변경하는 요청 (CUD)
Subscription	실시간 데이터 스트리밍 (WebSocket 기반)
Resolver	실제 데이터를 조회/생성/변경하는 구현체
Type	GraphQL 타입 시스템 (ObjectType, EnumType, InputType 등)

---

3 Spring Boot + GraphQL 구성 흐름

```
Client → GraphQL Query (POST /graphql) → Spring Boot GraphQL Engine → Resolver 실행 → 데이터 반환
```

---

4 주요 라이브러리

라이브러리	설명
`spring-boot-starter-graphql`	Spring Boot 2.7+ 이후 공식 지원
`graphql-java`	Java 기반 GraphQL 구현체

라이브러리	설명
`graphql-kickstart`	Spring Boot 2.x 시절 많이 사용됨 (legacy)

현재는 `spring-boot-starter-graphql` 사용 권장

---

5 build.gradle.kts 예시

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-graphql")
    implementation("com.graphql-java:graphql-java-extended-scalars:21.0") // 필요 시 Scalar 확장 지원
    implementation("org.springframework.boot:spring-boot-starter-data-jpa")
    runtimeOnly("com.h2database:h2")

    testImplementation("org.springframework.boot:spring-boot-starter-test")
}
```

---

6 Schema 작성 예시

파일 위치: `src/main/resources/graphql/schema.graphqls`

```
type Query {
    user(id: ID!): User
    users: [User]
}

type Mutation {
    createUser(input: UserInput!): User
}

type User {
    id: ID!
    name: String!
    email: String!
}

input UserInput {
    name: String!
    email: String!
}
```

7️⃣ Resolver 구현 예시

7.1 Query Resolver

```
@GraphQlController
public class UserQueryResolver {

    private final UserService userService;

    public UserQueryResolver(UserService userService) {
        this.userService = userService;
    }

    @QueryMapping
    public UserDto user(@Argument Long id) {
        return userService.getUserById(id);
    }

    @QueryMapping
    public List<UserDto> users() {
        return userService.getAllUsers();
    }
}
```

7.2 Mutation Resolver

```
@GraphQlController
public class UserMutationResolver {

    private final UserService userService;

    public UserMutationResolver(UserService userService) {
        this.userService = userService;
    }

    @MutationMapping
    public UserDto createUser(@Argument UserInput input) {
        return userService.createUser(input);
    }
}
```

8️⃣ Client Query 예시

User 조회

```
query {
    user(id: 1) {
        id
        name
        email
```

```
    }
}
```

User 생성

```
mutation {
    createUser(input: {name: "Alice", email: "alice@example.com"}) {
        id
        name
        email
    }
}
```

9️⃣ GraphQL의 장점과 단점

장점

✅ 클라이언트가 원하는 데이터만 요청 → Overfetching/Underfetching 해소
✅ 단일 Endpoint → API 설계 단순화
✅ Schema 기반 API → 자동 문서화 가능
✅ 강력한 개발 도구 지원 (GraphQL Playground, Insomnia, Postman 등)
✅ API Evolution 대응이 REST 대비 유연

단점

❌ 초기 학습 비용 있음 (특히 Schema 설계)
❌ 복잡한 쿼리 → 성능 저하 가능 → N+1 문제 대응 필요
❌ 보안 설계 주의 필요 (쿼리 Depth 제한, Rate Limiting 등 필요)

1️⃣0️⃣ GraphQL + Spring Boot 실전 적용시 Best Practice

✅ Schema 중심 API 설계 → DDD 와 매우 잘 어울림
✅ Application Layer 를 통해 Resolver 구현 → Hexagonal Architecture 자연스럽게 적용 가능
✅ DataLoader / BatchLoader 활용 → N+1 문제 적극 대응
✅ Resolver 에 복잡한 비즈니스 로직 넣지 말 것 → Service Layer 로 위임
✅ Query와 Mutation 명확히 구분
✅ Subscription (실시간 기능) 은 WebSocket 지원으로 구성 가능 (GraphQL Subscriptions)

1️⃣1️⃣ 실전 구성 패턴 예시

```
com.example.app
├── domain
|   ├── model
|   ├── service
|   ├── event
├── application
|   ├── port.in
```

```
|   ├── port.out
|   ├── service
├── adapter
|   ├── in
|   |   ├── graphql (GraphQL Resolver)
|   ├── out
|   |   ├── persistence (JPA Adapter 등)
├── config
|   └── GraphQLConfig.kt
```

→ 기존 Hexagonal Architecture 구조에서 **GraphQL Adapter 가 IN Adapter 역할**로 자연스럽게 적용됨

결론

Spring Boot + GraphQL은:

- ✅ API 설계 유연성 향상 → 클라이언트 친화적 API 제공 가능
- ✅ DDD/Hexagonal Architecture 에 매우 잘 어울림 → IN Adapter 로 깔끔하게 구성 가능
- ✅ Schema 기반 설계로 API 문서 자동화 가능
- ✅ Query/Mutation 구분, Subscription 지원으로 강력한 API 플랫폼 구축 가능

WebSocket / STOMP

1️⃣ WebSocket 개요

기본 개념

- WebSocket은 **양방향(Full Duplex) 통신 프로토콜**
- HTTP는 **요청-응답 기반, 단방향 흐름**
- WebSocket은 한번 연결되면 **클라이언트-서버가 자유롭게 메시지 주고받음**

특징

- ✅ **항상 열린 연결 유지 (Persistent Connection)**
- ✅ **서버 Push 가능** → 클라이언트가 요청하지 않아도 서버가 데이터를 보냄
- ✅ **빠른 실시간 통신** 가능 (웹 기반 채팅, 실시간 알림, IoT 등 사용)

2️⃣ STOMP란?

기본 개념

- **Simple (or Streaming) Text Oriented Messaging Protocol**
- WebSocket 위에서 사용하는 **메시징 프로토콜** → 구조적 메시지 교환 지원
- Pub/Sub 구조 지원
- Spring에서는 WebSocket을 **STOMP 기반으로 추상화**해서 쉽게 사용 가능

WebSocket vs STOMP

구분	WebSocket	STOMP
레이어	Transport 프로토콜	Application 프로토콜
데이터 구조	자유로운 바이너리/텍스트	프레임 기반 구조화된 메시지
Pub/Sub 지원	직접 구현 필요	프로토콜에 내장 지원
브로커 연동	직접 구현	내장 브로커 or 외부 브로커 연계 가능

---

3 Spring Boot 기반 구성 흐름

```
Client <--> WebSocket (WS/ WSS) <--> Spring WebSocket Handler
                                          |
                                          v
                                    MessageBroker
                                          |
                                          v
                                Application @MessageMapping
```

---

4 주요 구성 요소 (Spring)

구성 요소	역할
`@EnableWebSocketMessageBroker`	WebSocket/STOMP 메시지 브로커 활성화
STOMP Client	웹 프론트엔드 (ex: SockJS, Stomp.js)
SimpleBroker	내장 메시지 브로커 (간단한 Pub/Sub 지원)
`@MessageMapping`	서버 측 메시지 수신 처리
`@SendTo`	브로커에 메시지 전송 (구독자에게 broadcast)

---

5 기본 설정 예시

5.1 build.gradle.kts

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-websocket")
}
```

5.2 WebSocketConfig

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic", "/queue"); // 구독 Prefix
        config.setApplicationDestinationPrefixes("/app"); // 서버로 보내는 Prefix
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws").withSockJS(); // WebSocket Endpoint (Fallback 지원)
    }
}
```

6️⃣ 서버 측 메시지 핸들링

6.1 수신 처리

```
@Controller
public class ChatController {

    @MessageMapping("/chat.sendMessage")
    @SendTo("/topic/public")
    public ChatMessage sendMessage(ChatMessage message) {
        return message;
    }
}
```

- 클라이언트 → `/app/chat.sendMessage` 로 송신
- 서버는 `/topic/public` 으로 broadcast

6.2 ChatMessage DTO 예시

```
public class ChatMessage {
    private String sender;
    private String content;
    private String type;
}
```

---

7 클라이언트 구성 (JS + Stomp.js + SockJS)

```
<script src="https://cdn.jsdelivr.net/npm/sockjs-client/dist/sockjs.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/stompjs/lib/stomp.min.js"></script>

<script>
    var socket = new SockJS('/ws');
    var stompClient = Stomp.over(socket);

    stompClient.connect({}, function (frame) {
        console.log('Connected: ' + frame);
        stompClient.subscribe('/topic/public', function (message) {
            console.log("Received: ", JSON.parse(message.body));
        });
    });

    function sendMessage() {
        stompClient.send("/app/chat.sendMessage", {}, JSON.stringify({
            sender: "Alice",
            content: "Hello World!",
            type: "CHAT"
        }));
    }
</script>
```

---

8 실전 적용 시 고려사항

✅ 클라이언트는 **STOMP 기반 Socket Client 사용 추천** → `SockJS + Stomp.js` 조합 많이 사용됨
✅ Spring에서는 기본 **SimpleBroker** → 대용량 시 **RabbitMQ / ActiveMQ / Redis 등 외부 브로커 연계 가능**
✅ **스케일 아웃 구성 시 브로커 사용 필수** (SimpleBroker는 단일 노드 한정)
✅ WebSocket 보안 → **WSS(TLS) 구성 필수** + **JWT 등으로 인증/인가 처리** 필요

---

9 WebSocket/STOMP 사용 시 주의사항

Nginx / Proxy 구성 시 WebSocket 지원 확인 필요

```
location /ws {
    proxy_pass http://your_backend_server;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
}
```

메시지 크기 제한

- 클라이언트/서버 간 **전송 메시지 크기 제한** 설정 가능
  → 대용량 전송 시 Chunk 처리 고려 필요

1️⃣0️⃣ 실전 적용 사례

✅ 실시간 채팅 서비스
✅ 실시간 알림 (Notification) 시스템
✅ 게임 서버 클라이언트 통신
✅ 주식 거래 시스템 (실시간 가격 Push)
✅ IoT 데이터 실시간 스트리밍

1️⃣1️⃣ 결론

기술	역할
WebSocket	Transport 레이어 (Full-Duplex 통신)
STOMP	Application 레이어 (메시지 프로토콜)
Spring Boot WebSocket	WebSocket/STOMP 지원 프레임워크
SimpleBroker / External Broker	Pub/Sub 메시지 브로커 구성 가능

Spring Boot + WebSocket + STOMP 구조는:

✅ DDD/Hexagonal Architecture 기반에서도 **IN Adapter로 자연스럽게 연동 가능**
✅ REST API 와 함께 **실시간 기능 보완적으로 사용**
✅ 테스트/구성/운영 모두 비교적 안정적이고 잘 지원됨

Reactive Programming (Spring WebFlux)

1️⃣ Reactive Programming 기본 개념

정의

- **데이터 흐름(Data Stream)과 변화(Propagation of Change)** 를 중심으로 설계하는 프로그래밍 패러다임
- 비동기 + 이벤트 기반 + 논블로킹 I/O 처리

주요 특징

✅ 비동기(Asynchronous) 처리
✅ 논블로킹(Non-blocking) → 적은 스레드로 많은 요청 처리 가능
✅ Backpressure 지원 → 과도한 데이터 흐름 시 압력 조절 가능
✅ Functional/Declarative 스타일 코드 → 선언적 스타일 작성 가능

2️⃣ 왜 사용하는가?

전통적 Servlet 기반 MVC (Blocking I/O)

```
Client → Thread → Blocked waiting for I/O → Response → Thread released
```

- 동시 요청이 많으면 Thread 수가 급증 → 성능/메모리 한계 발생 가능

Reactive / WebFlux 기반 (Non-blocking I/O)

```
Client → Event Loop → Callback 처리 → Response
```

- Thread 수 최소화 → 고효율 스케일 지원
- CPU 사용량/메모리 사용량 최적화
- WebSocket / Streaming / Microservices → 효과적

3️⃣ Reactive Streams 기본 구성 요소

구성 요소	설명
Publisher	데이터 제공자 (생성)
Subscriber	데이터 소비자
Subscription	Subscription 상태 관리
Processor	Publisher → Subscriber 역할을 모두 수행 (중간 처리 가능)
Backpressure	Publisher → Subscriber 에게 처리 속도 제어 기능 제공

대표 구현체

라이브러리	설명
Project Reactor (Spring WebFlux)	Spring 공식 채택, Flux/Mono 제공
RxJava	Netflix 주도 개발
Akka Streams	JVM 기반 Reactive Streams 구현

4️⃣ Project Reactor 핵심 타입

타입	설명
`Mono<T>`	0 ~ 1 개의 데이터 제공
`Flux<T>`	0 ~ N 개의 데이터 제공 (Stream)

예시

```
Mono<String> mono = Mono.just("Hello");
Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);
```

---

5 WebFlux 구성 흐름

```
Client → WebFlux Router/Handler → Reactive Service → Reactive Repository → DB
```

- 모든 Layer 가 **Reactive 타입(Mono/Flux)** 으로 처리 → End-to-End 논블로킹

---

6 build.gradle.kts 예시

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-webflux")
    implementation("org.springframework.boot:spring-boot-starter-data-r2dbc") //
Reactive DB
    runtimeOnly("io.r2dbc:r2dbc-h2") // H2 R2DBC 드라이버

    testImplementation("org.springframework.boot:spring-boot-starter-test") {
        exclude(group = "org.junit.vintage", module = "junit-vintage-engine")
    }
    testImplementation("io.projectreactor:reactor-test")
}
```

---

7 Controller 예시 (REST API)

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

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

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

    @GetMapping
    public Flux<UserDto> getAllUsers() {
        return userService.getAllUsers();
    }
```

```
}
```

8 Service 예시

```
@Service
public class UserService {

    private final UserRepository userRepository;

    public Mono<UserDto> getUserById(Long id) {
        return userRepository.findById(id)
                .map(user -> new UserDto(user.getId(), user.getName(),
user.getEmail()));
    }

    public Flux<UserDto> getAllUsers() {
        return userRepository.findAll()
                .map(user -> new UserDto(user.getId(), user.getName(),
user.getEmail()));
    }
}
```

9 Reactive Repository 예시

```
@Repository
public interface UserRepository extends ReactiveCrudRepository<User, Long> {
}
```

- R2DBC 기반 DB 사용 시 적용 가능
- ReactiveCrudRepository → Mono/Flux 기반 메서드 제공

10 WebFlux vs Spring MVC

측면	WebFlux	Spring MVC
프로그래밍 모델	Reactive (Mono/Flux)	Imperative (Blocking)
Servlet	사용 X	사용 (Servlet API 기반)
스레드 모델	Non-blocking, Event Loop	Blocking I/O Thread 사용
고성능 대량 요청 처리	우수	상대적으로 열위
개발 복잡성	높음	낮음 (전통적 방식 익숙함)
성능 튜닝 필요성	높음	낮음

1️⃣1️⃣ WebFlux 도입 시 유의사항

✅ 서비스 전체를 Reactive 로 구성해야 진정한 효과 → 중간에 Blocking API 사용 금지
✅ 외부 API 연동 시 WebClient 사용 (RestTemplate 사용 금지)
✅ JDBC (Blocking) DB 사용 시 WebFlux 이점 제한 → R2DBC 등 사용 권장
✅ Thread Context 관리 주의 (ex: SecurityContext 전파)
✅ Reactor Debug Mode 를 통해 Flow 분석 필수

1️⃣2️⃣ WebClient 예시 (Reactive HTTP Client)

```
WebClient webClient = WebClient.create();

Mono<UserDto> userMono = webClient.get()
        .uri("http://localhost:8080/users/1")
        .retrieve()
        .bodyToMono(UserDto.class);
```

- WebClient 자체가 **Reactive Client** → End-to-End 논블로킹 구성 가능

결론

Spring WebFlux + Project Reactor 조합은:

✅ 고성능 / 고동시 요청 처리에 매우 적합
✅ MSA / 실시간 스트리밍 / WebSocket / GraphQL 등에 잘 어울림
✅ DDD/Hexagonal Architecture에서도 **Reactive Adapter 계층으로 잘 적용 가능**
✅ REST API / WebSocket / RSocket / Server-Sent Events → 모두 지원 가능

적용 Best Practice

✅ **Layer 간 Mono/Flux 타입 일관성 유지**
✅ DDD Domain Layer는 필요시 Adapter 계층에서 Blocking <-> Reactive 변환 구간 명확히 정의
✅ WebClient 적극 활용 → 외부 API 연동 시 논블로킹 처리
✅ 단순 CRUD 서비스에 무리하게 도입할 필요는 없음 → 고성능/Reactive 요구가 있는 경우 추천
✅ DB Layer는 R2DBC or Reactive NoSQL(Mongo Reactive Driver 등) 사용 권장

Native Image (GraalVM) 빌드

1️⃣ Native Image란?

기본 개념

- **Java 애플리케이션 → OS Native 바이너리 (.exe, ELF, Mach-O 등)** 로 **Ahead-of-Time (AOT) Compile** 수행
- GraalVM Native Image Compiler 를 사용

전통적인 Java 실행

```
Java Source → Bytecode (.class) → JVM (JIT Compile) → Native Code 실행
```

Native Image 실행

```
Java Source → Bytecode → Native Image Compile → Native Binary → 직접 실행
```

주요 차이점

측면	JVM 실행	Native Image
부트 시간	수백 ms ~ 수 초	수 ms ~ 수십 ms
메모리 사용량	상대적으로 큼	매우 적음 (최대 50~70% 감소)
런타임 특성	JIT 최적화 있음	AOT Compile (고정 성능)
런타임 필요성	JVM 필요	없음 (Native 실행 파일)
동적 특성 (Reflection 등)	자유로움	제한적 (Static Analysis 필요)

2️⃣ 왜 사용하는가?

✅ **초고속 부팅 속도** → Serverless (AWS Lambda 등), Microservices 에 적합
✅ **낮은 메모리 사용량** → 비용 절감, 고밀도 배포 가능
✅ **런타임 없는 배포 가능** → Java 환경이 없는 OS에서도 실행 가능
✅ **경량 컨테이너 이미지 구축 용이** → Cloud Native 최적화

3️⃣ GraalVM Native Image 개요

주요 구성 요소

구성 요소	역할
GraalVM Compiler	Java Bytecode → Native Code 컴파일
Native Image Tool (`native-image`)	Native Binary 빌드 수행
Substrate VM	GraalVM Native Image 실행시 필요한 최소 런타임

4️⃣ Spring Boot Native 지원 현황

- Spring Boot 3.x → **GraalVM Native Image 공식 지원**
- Spring Boot 3.x + Spring Framework 6.x → AOT 기반 구성 최적화
- **spring-boot-starter-aot** 사용 가능
- GraalVM CE / EE / Oracle GraalVM 지원

5 구성 흐름

```
Java Source → Build → AOT Processing → Native Image Build → Native Executable

실행 흐름: OS 실행 → Native Binary 직접 실행 → 수 ms 내 서비스 가능
```

6 Build 예시 (Gradle 기반)

6.1 build.gradle.kts 구성 예시

```
plugins {
    id("org.springframework.boot") version "3.2.5"
    id("io.spring.dependency-management") version "1.1.4"
    kotlin("jvm") version "1.9.22"
    kotlin("plugin.spring") version "1.9.22"
    id("org.graalvm.buildtools.native") version "0.10.1"
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web")
    testImplementation("org.springframework.boot:spring-boot-starter-test")
}

graalvmNative {
    toolchainDetection.set(true) // GraalVM Toolchain 자동 감지
    binaries {
        named("main") {
            buildArgs.add("--no-fallback") // fallback 이미지 비활성화 (strict 모드)
            buildArgs.add("-H:+ReportExceptionStackTraces") // 예외 스택트레이스 출력
        }
    }
}
```

7 Build 및 실행 방법

1 Build

```
./gradlew nativeCompile
```

→ 결과: `build/native/nativeCompile/my-app` (OS Native Binary 생성)

2 실행

```
./build/native/nativeCompile/my-app
```

→ 수 ms 내에 서버 기동됨

8️⃣ 개발 시 주의사항

항목	주의 내용
Reflection	제한적 사용 → Spring AOT 자동 처리 지원
Dynamic Proxy	제한적 사용
JNI	지원 가능하나 복잡성 증가
Resource 접근	반드시 AOT 시점에 Resource 등록 필요
라이브러리 사용	Native Image Friendly 여부 확인 필요 (동적 ClassLoader 사용 여부 등)

→ Spring Boot 3.x + Spring AOT 플러그인 사용 시 대부분 자동 최적화 가능

9️⃣ GraalVM Native Image 성능 예시

항목	JVM 실행	Native Image
부트 속도	500 ms ~ 수 초	10 ~ 50 ms
메모리 사용량	200 ~ 500 MB	30 ~ 100 MB
Throughput (TPS)	JIT 효과로 JVM 우세	일정한 성능 (Peak 성능은 JVM 대비 약간 낮음)
Warm-up 필요성	Warm-up 필요	없음

1️⃣0️⃣ 도입 시 고려 사항

✅ **Cloud Native / Serverless** → 적극 추천
✅ **Microservices** → **빠른 스케일 아웃 필요** → 적극 추천
✅ **Legacy Monolith** → **전환 검토 필요 (Reflection 사용 여부 주의)**
✅ **고성능 연산 중심 서비스 (JVM JIT 최적화 의존 높음)** → 검토 필요 (성능 손실 가능성 있음)

결론

Spring Boot + GraalVM Native Image 조합은:

✅ **초고속 부팅, 낮은 메모리 사용량** → MSA/Serverless 환경에 최적화
✅ **Spring Boot 3.x 이후로 매우 강력하게 지원** → 실무 사용 가능 수준
✅ Gradle/Maven + `org.graalvm.buildtools.native` 플러그인으로 쉽게 통합 가능
✅ 실전 적용시 Reflection/Resource 관리에 주의 필요
✅ JVM 기반에서 충분히 검증한 후 단계적으로 Native 전환 적용 권장