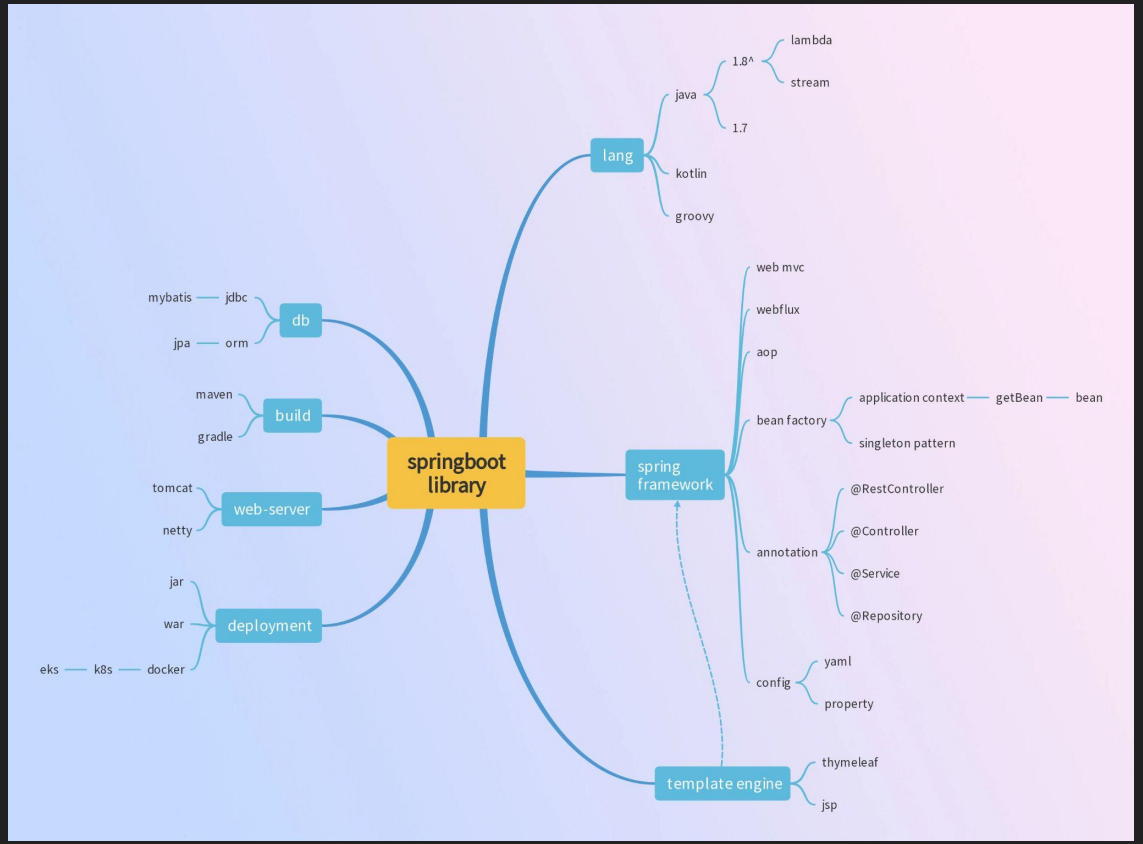
Springboot Architecture

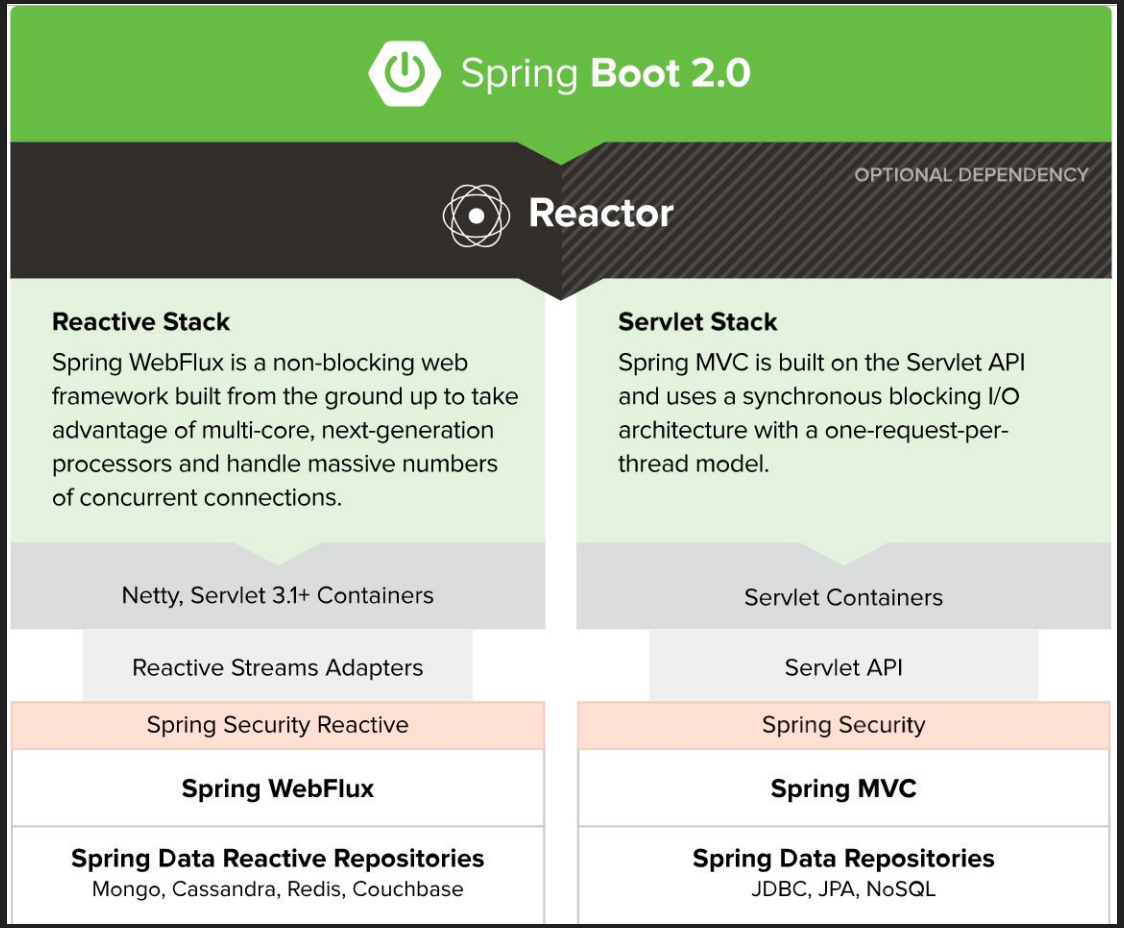
22.09.18 일요일 오후 2시 을지로입구역

What? Springboot

- 프레임워크인가?
- 라이브러리인가?
- 툴킷인가?

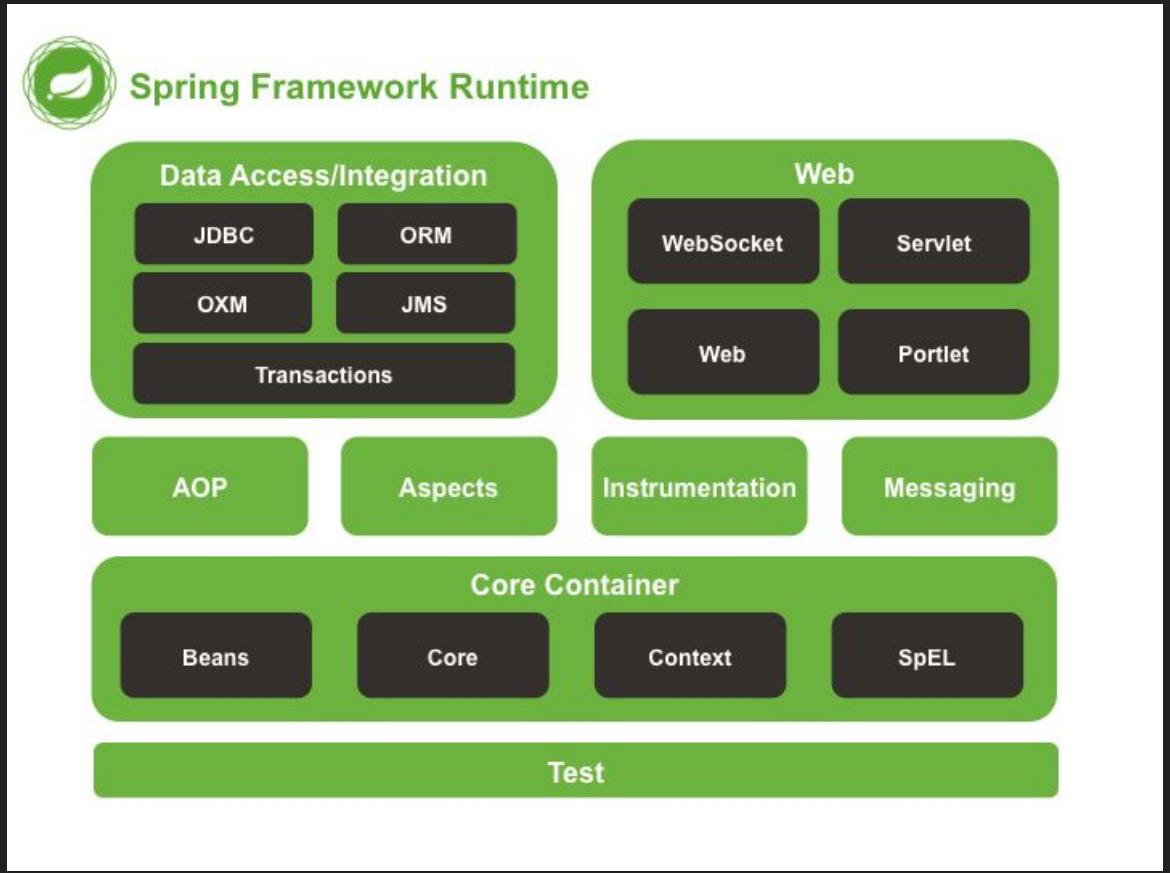


구성

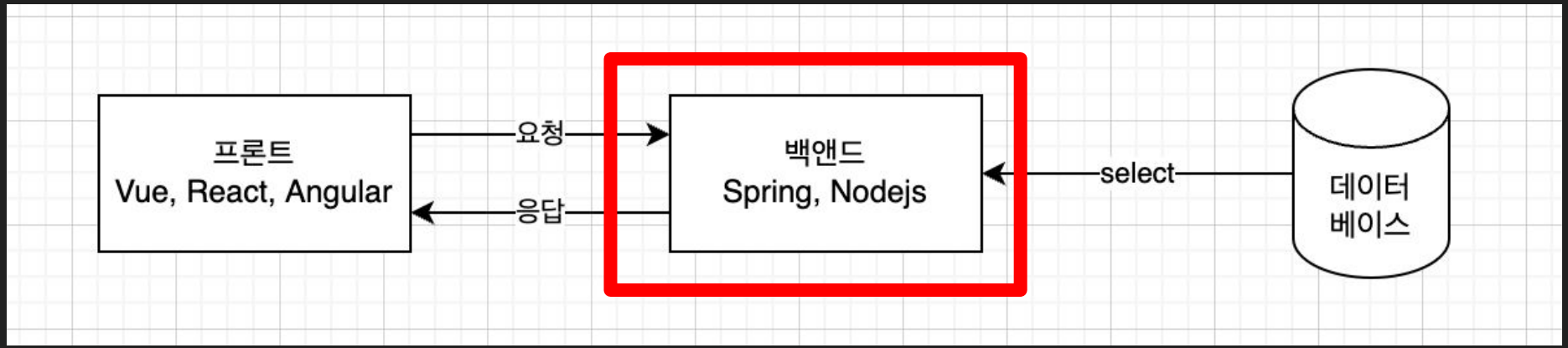

Spring Boot 2.0
Reactor
OPTIONAL DEPENDENCY
Reactive Stack
Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.
Servlet Stack
Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.
Netty, Servlet 3.1+ Containers
Servlet Containers
Reactive Streams Adapters
Servlet API
Spring Security Reactive
Spring Security
Spring WebFlux
Spring MVC
Spring Data Reactive Repositories
Mongo, Cassandra, Redis, Couchbase
Spring Data Repositories
JDBC, JPA, NoSQL

레이어

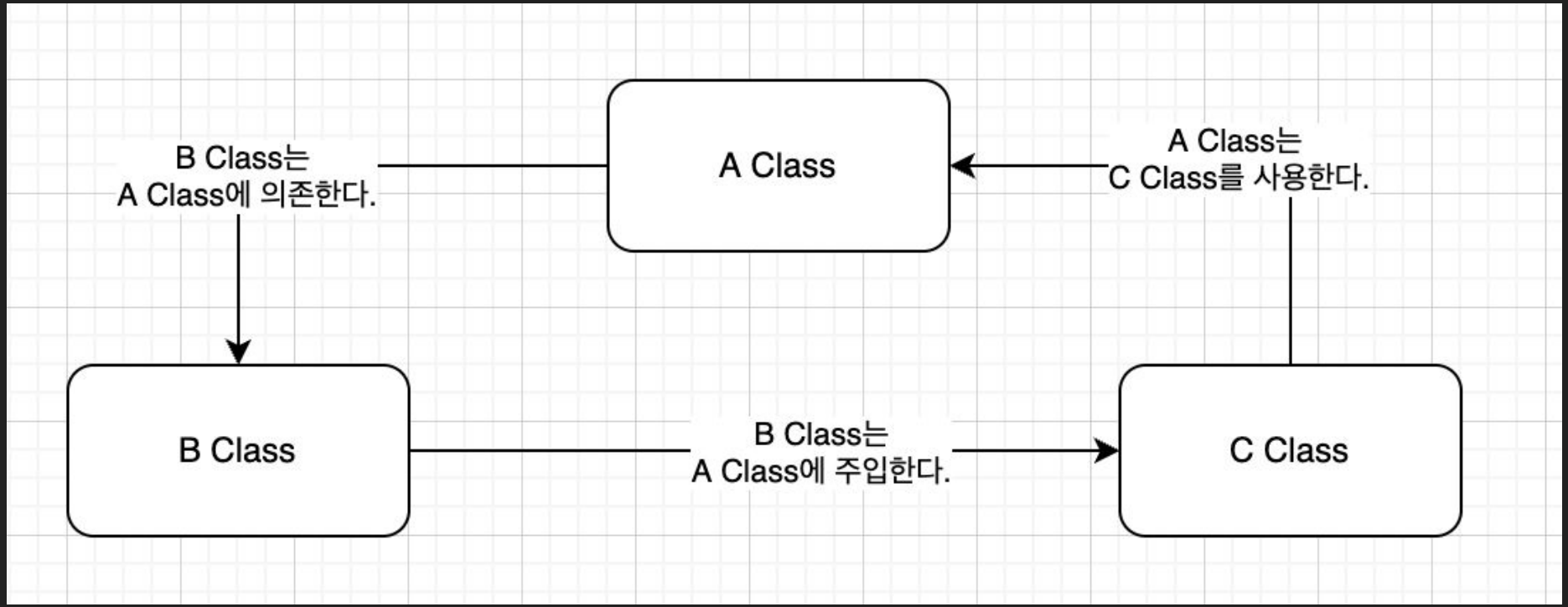
- View engine
- Java(1.8^, 1.7), Kotlin



Springboot Stack Position



의존성 관계


A Class
B Class는
A Class에 의존한다.
A Class는
C Class를 사용한다.
B Class
B Class는
A Class에 주입한다.
C Class

의존성 관계를 코드로 풀면?

```
Class A {
  private B b = new B();
  private print() {
    b.print();
  }
}

Class B {
  print() {
    logger.log('print called');
  }
}
```

의존성 주입 - 생성자 주입

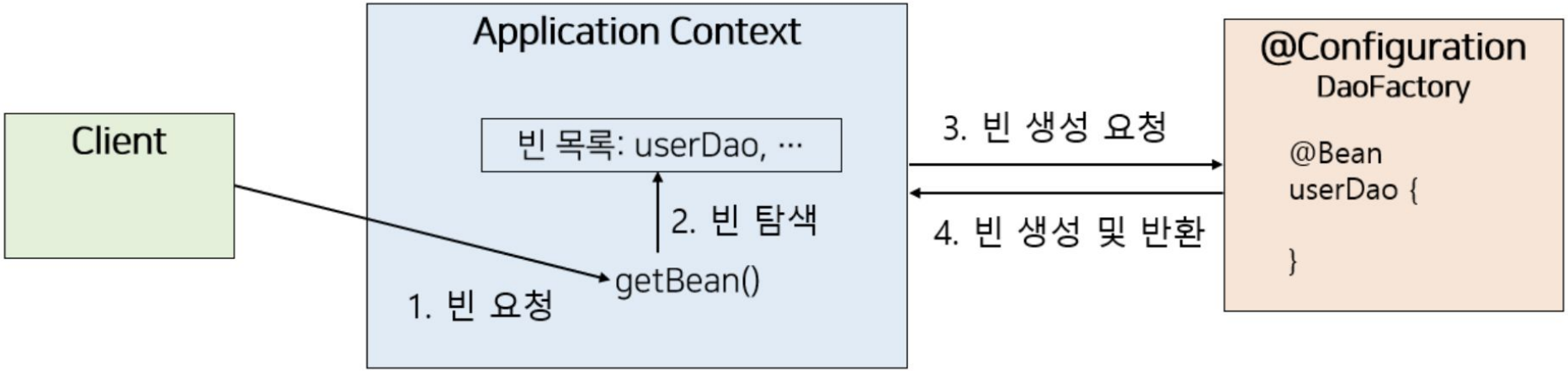
```
Class A {
  private B b;
  A(B b) {
    this.b = b;
  }
  private print() {
    b.print();
  }
}

Class B {
  print() {
    logger.log('print called');
  }
}
```

용어 정리

- Application Context
- IoC
- Di
- Bean
- Bean Factory
- OOP
- SOLID
- Design Pattern
- Class
- Instance

빈 요청시 처리과정


Application Context
Client
빈 목록: userDao, …
2. 빈 탐색
getBean()
1. 빈 요청
3. 빈 생성 요청
4. 빈 생성 및 반환
@Configuration
DaoFactory
@Bean
userDao {
}

어플리케이션 컨텍스트의 장점

- 클라이언트는 @Configuration이 붙은 구체적인 펙토리 클래스를 알 필요 없다.
- 애플리케이션 컨텍스트는 종합 IoC를 제공해 준다.
- 애플리케이션 컨텍스트는 다양한 검색 방법을 제공해 준다.

스프링이란 그래서 뭘까요?

좋은 객체지향 개발을 쉽게 할 수 있도록 도와주는 도구.

OOP란 무엇인가?

- 상속
- 캡슐화
- 다형성
- 추상화

각각의 객체는 **메세지**를 주고받고 데이터를 처리 할 수 있고, **협력** 해야 한다.

더불어 이는 프로그램을 마치 컴퓨터 부품을 갈아 끼우듯 **유연**하고 **변경**에 용이 하게 만들어 주기 때문에 대규모 소프웨어 개발에서 많이 이용되고 있다.

다형성은 역할과 구현으로 표현 할 수 있다.

- 자동차가 역할이라면 구현체는 K3, 소나타, 테슬라 모델3라는 것이 있다.
- 로미오 줄리엣이라는 역할이 있다면, 구현체는 장동건, 김태희 라는 것이 있을 수 있다.

이렇게 역할과 구현으로 나누면 세상은 단순해 지고, 유연하고 변경이 용이해 진다.

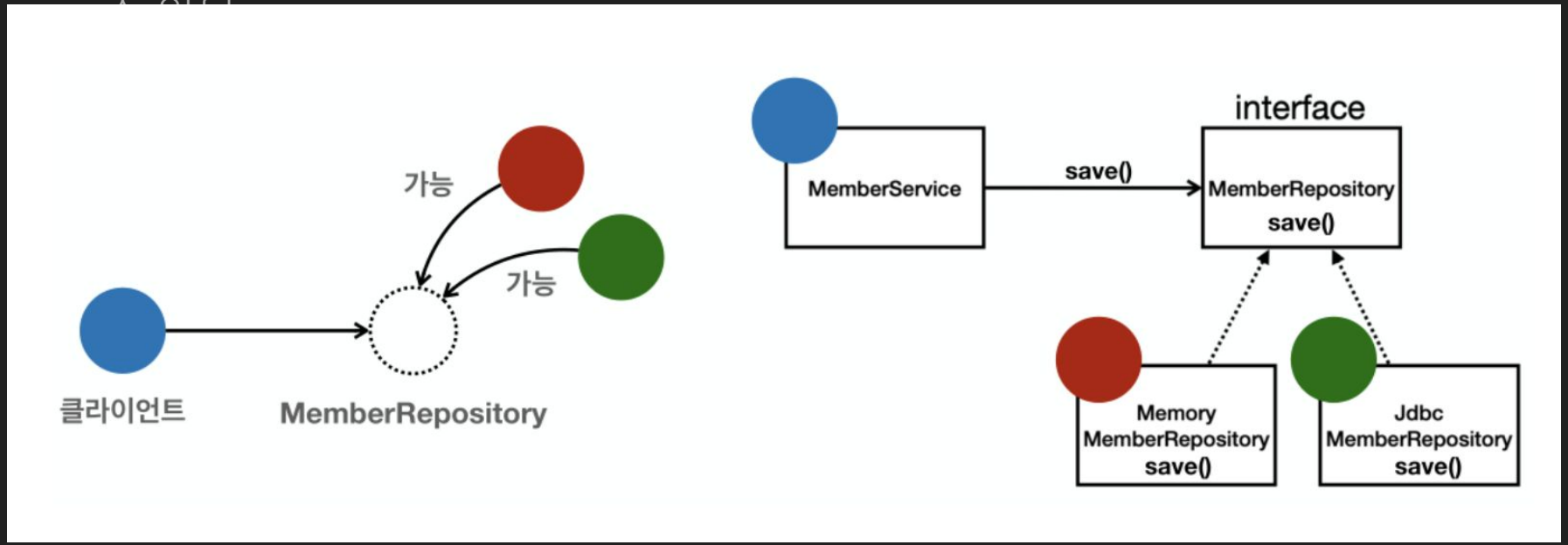
- 클라이언트는 대상의 역할만 알면 된다.(인터페이스)
- 클라이언트는 구현 대상의 내부구조를 몰라도 된다.
- 클라이언트는 구현 대상의 내부구조가 변경되어도 영향을 받지 않는다.
- 클라이언트는 구현 대상 자체를 바꿔도 영향을 받지 않는다.

자바 언어에 대입하면 역할은 **인터페이스**이고, 구현은 **클래스**가 된다.

즉 객체설계시 역할을 먼저 만들고 클래스를 만들어야 한다.

오버라이딩

- 실제 동작은 **오버라이딩** 된 메서드에 **구현**을 하는 것
- 클래스는 **인터페이스**를 구현한 것이므로, **유연**하게 필요에 따라 객체를 **변경**할



방금 그림을 코드로 보면,

```
public class MemberService {
//  private MemberRepository memberRepository = new MemoryMemberRepository();
    private MemberRepository memberRepository = new JdbcMemberRepository();
}
```

즉, 다형성의 본질은 인터페이스를 구현한 객체 인스턴스를 **실행시점**에 **유연**하게 **변경**할 수 있다.

이를 이해 하려면, **협력**이라는 **객체 사이**에 관계에서 시작해야 한다.

클라이언트를 변경하지 않고, 서버의 구현 기능을 유연하게 변경 가능 해야 한다.

좋은 객체지향 설계의 원칙 - SOLID

- SRP : 단일 책임의 원칙
- **OCP : 개방/패쇄의 원칙**
- LSP : 리스코프 치환의 원칙
- ISP : 인터페이스 분리의 원칙
- **DIP : 의존관계 역전의 원칙**