Spring framework

Spring framework: 기업용 어플리케이션을 만드는데 사용 가능한 오픈소스 프레임워크

Spring framework runtime

1. Data Access/Integration: JDBC, ORM, OXM, JMS, Transactions
2. Web: WebSocket, Servlet, Web, Portlet
3. Core Container: Beans, Core, Context, SpEL

스프링 내에는 동일한 역할을 하는 다양한 기능이 있으며, 그 중에서 적합한 툴을 선택할 수 있어야 함.

Spring boot: 스프링 기반으로 자주 사용되는 설정으로 손쉽게 개발할 수 있게 해주는 상위 프레임워크

* 스프링(각종 도구가 있는 템플릿)보다 한층 더 편리한 프레임워크
* 웹 어플리케이션(톰캣 등) 서버 내장
* 자동 설정, 설정 표준화

Spring framework를 이루는 다양한 기술:

* Core (DI, IoC)
* APO (Aspect Oriented Programming)
* Validation, Data binding
* Resource
* SpEL
* Null-Safety

디자인 철학:

* 모든 기능에서 다양한 가능성(다양한 모듈)을 사용 가능, 심지어 외부 모듈을 활용 가능/단 너무 높은 자유도는 어떤 점에서는 스프링을 어렵게 하는 요소이다.
* 유연하게 계속 추가 개발을 하고 있는 프레임워크
* 이전 버전과의 강력한 호환성/단 너무 많은 레거시 때문에 코드의 복잡성이 높아짐
* API 디자인을 섬세하게 노력한다/스프링 코드 자체가 하나의 좋은 참고 소스
* 높은 코드 품질을 유지하려 함/

요약: 높은 자유도를 주고 계속 발전하는 고품질의 다양성이 있는 프로젝트, 그런데 너무 자유로워서 때론 어렵다

IoC (Inversion of Control) & DI (Dependency Injection):

* IoC나 DI는 레고와 같은 것이다.
* 스프링이 바닥판처럼 깔려 있고, 우리는 그 위에 조합(나의 어플리케이션)을 만들면 된다.
* 레고판위에 올려놓는 클래스들은 Bean, 판 밖에 있는 그냥 클래스이다 예를 들면 유틸리티 클래스(독립적인 클래스, 스프링에 관리를 받지 않는)

Bean:

* 자바에서 JavaBean:
* 데이터를 저장하기 위한 구조체로 자바 빈 규약이라는 것을 따르는 구조체
* Private 프로퍼티와 getter/setter로만 데이터를 접근한다.
* 스프링에서의 Bean
* 스프링 IoC 컨테이너에 의해 생성되고 관리되는 객체
* 자바 에서처럼 new Object();로 생성하지 않는다
* 각각의 Bean들끼리는 서로를 편리하게 의존(사용)할 수 있음

스프링 컨테이너 개요:

* ApplicationContext 인터페이스를 통해 제공되는 스프링 컨테이너는 Bean 객체의 생성 및 Bean들의 조립(상호 의존성 관리)을 담당한다.
* Bean의 등록:
* 과거에는 xml로 설정을 따로 관리하여 등록(불편)
* 현재는 annotation 기반으로 Bean 등록
* @Bean, @Controller, @Service
* Bean 등록 시 정보
* Class 경로
* Bean의 이름: 기본적으로는 원 Class 이름에서 첫 문자만 소문자로 변경 => accountService, userDao
* Scope: Bean을 생성하는 규칙
* Singleton: 컨테이너에 단일로 생성
* Prototype: 작업 시마다 Bean을 새로 생성하고 싶을 경우
* Request: http 요청마다 새롭게 Bean을 생성하고 싶은 경우
* Bean LifeCycle callback:
* Callback: 어떤 이벤트가 발생하는 경우 호출되는 메서드
* Lifcycle callback: Bean을 생성하고 초기화하고 파괴하는 등 특정 시점에 호출되도록 정의된 함수
* 주로 많이 사용되는 콜백: @PostConstruct(빈 생성 시점에 필요한 작업을 수행), @PreDestroy(빈 파괴 시점에 필요한 작업을 수행)

AOP: 관전 지향 프로그래밍 – Aspect Oriented Programming

* 특정한 함수 호출 전이나 후에 뭔가 공통적인 처리가 필요하다면 => AOP
* 로깅, 트랜잭션, 인증
* OOP로 처리하기에는 다소 까다로운 부분을 AOP라는 처리 방식을 도입하여 손쉽게 공통 기능을 추가/수정/삭제 할 수 있도록 함

AOP의 기본 개념:

* Aspect: 여러 클래스나 기능에 걸쳐서 있는 관심사, 그리고 그것들을 모듈화함. AOP 중에서 가장 많이 활용되는 부분은 @Transactional 트랜잭션 관리 기능
* Advice: 조언, AOP에서 실제로 적용하는 기능(로깅, 트랜잭션, 인증 등)을 뜻함
* Join point: 모듈화된 특정 기능이 실행될 수 있는 연결 포인트
* Pointcut: join point 중에서 해당 Aspect를 적용할 대상을 뽑을 조건식
* Target Object: Advice가 적용될 대상 오브젝트
* AOP Proxy: 대상 오브젝트에 Aspect를 적용하는 경우 Advice를 덧붙이기 위해 하는 작업을 AOP Proxy라고 함. 주로 CGLIB(Code Generation Library, 실행 중에 실시간으로 코드를 생성하는 라이브러리) 프록시를 사용하여 프록싱 처리를 한다.
* Weaving: Advice를 비즈니스 로직 코드에 삽입하는 것을 말함
* AspectJ 지원: AspectJ는 AOP를 제대로 사용하기 위해 꼭 필요한 라이브러리, 기본적으로 제공되는 Spring AOP로는 다양한 기법(pointcut 등)의 AOP를 사용할 수 없음

Aspect의 생성:

Ex):

Package org.xyz;

Import org.aspectj.lang.annotation.Aspect;

@Aspet

@Component // Component를 붙인 것은 해당 Aspect를 스프링의 Bean으로 등록해서 사용하기 위함

public class UsefulAspect{

}

Pointcut 선언

Package org.xyz;

Import org.aspectj.lang.annotation.Aspect;

@Aspet

@Component // Component를 붙인 것은 해당 Aspect를 스프링의 Bean으로 등록해서 사용하기 위함

public class UsefulAspect{

@Pointcut(“execution(\* transfer(..))”)

private void anyOldTransfer() {}

}

* 해당 Aspect의 Advice(실행할 액션)이 적용될 Join point를 찾기 위한 패턴 또는 조건 생성
* 포인트 컷 표현식이라고 부름

Validation:

* 유효성검증: 주로 사용자 또는 서버의 요청(http request) 내용에서 잘못된 내용이 있는지 확인하는 단계를 뜻함

Validation의 종류:

* 데이터 검증:
* 필수 데이터의 존재 유무
* 문자열의 길이나 숫자형 데이터의 경우 값의 범위
* Email, 신용카드 번호 등 특정 형식에 맞춘 데이터
* 비즈니스 검증
* 서비스에 정책에 따라 데이터를 확인하여 검증
* 예) 배달앱인 경우 배달 요청을 할 때 해당 주문건이 결제 완료 상태인지 확인 등
* 경우에 따라 외부 API를 호출하거나 DB의 데이터까지 조회하여 검증하는 경우도 존재

Spring의 Validation

* 스프링은 웹 레이어에 종속적이지 않은 방법으로 validation을 하려고 의도하고 있으며 주로 아래 두가지 방법을 활용하여 유효성검증 진행(둘다 데이터 검증에 가까움)
* Java Bean Validation
* JavaBean 기반으로 간편하게 개별 데이터를 검증

Ex):

Public class MemberCreationRequest {

@NotBlank(message=”이름을 입력해주세요.”)

@Size(max=64, message=”이름의 최대 길이는 64자 입니다.”)

….

}

위처럼 요청 dto에 어노테이션으로 명시 후 아래처럼 @Valid 어노테이션을 해당 @RequestBody에 달게 되면, Java Bean Validation을 수행한 후 문제가 없을때만 메서드 내부로 진입 된다.

@PostMapping(value = “/member”)

Public MemberCreationRTesponse createMember(

@Valid @RequestBodu final MemberCreationRequest memberCreationRequest){

// member createion logics here…

}

)

Validation 수행 시 주의사항 및 패턴

* 주의사항
* Validation이 너무 여러 군데에 흩어져있으면 테스트 및 유지보수성이 떨어짐
* 중복된 검증: 정책 변경 시에 모든 중복 코드를 수정해야 함
* 다른 검증: 여러 군데서 다른 정책을 따르는 검증이 수행될 수 있음
* 가능한 validation은 로직 초기에 수행 후 실패 시에는 exception을 던지는 편이 처리가 편리함
* 실무 활용 패턴
* 필자의 주 사용 패턴
* 요청 dto에서 Java Bean Validation으로 단순 데이터(유무, 범위, 형식 등)를 1차 검증
* 로직 초기에 2차로 비즈니스 검증 수행 후 실패 시에는 Custom Exception(ErrorCode, ErrorMessage를 입력)해서 예외를 던지도록 하고 예외처리하여 응답 생성
* Spring validator의 장단점:
* 장점: Java Bean Validation에 비해 조금 더 복잡한 검증이 가능
* 단점: Validation을 수행하는 코드를 찾기가 (상대적으로) 어렵다. 완전이 데이터만 검증하는 것이 아니기 때문에 일부 비즈니스적인 검증이 들어가는 경우가 있음 => 이 경우 비즈니스 검증 로직이 여러 군데로 흩어지기 때문에 잘못된 검증(중복 검증, 다른정책을 따르는 검증)을 수행할 가능성이 높아짐

Data Binding: 사용자나 외부 서버의 요청 데이터를 특정 도메인 객체에 저장해서 우리 프로그램에 Request에 담아주는 것을 뜻한다.