< 의존성 관리 >

<artifactId>spring-boot-starter-dependencies<artifactId>
- <artifactId>spring-boot-starter-parent</artifactId> 상위 프로젝트
- 가장 상위 프로젝트
- <dependencyManagement> 영역에서 의존성 관리(버젼 관리)
- 장점
    - 우리가 직접관리해야할 의존성이 줄어든다

스프링 부트 관리 2가지 방법
1) 자신의 프로젝트에 <parent>로 spring-boot-starter-parent를 선언하여 설정
2) <dependencyManagement> 엘리먼트를 사용하여 dependency 주입
    - 의존성 관리외 다른 여러 설정 적용 X(자바설정, 인코딩설정)

<자동설정>

@SpringBootApplication
= @SpringBootConfiguration + @ComponentScan + @EnableAutoConfiguration

스프링부트에서 Bean을 등록하는 단계
1단계 : @ComponentScan
- @Componet를 가진 클래스들을 스캔해서 빈으로 등록
- @Configuration / @Repository / @Service / @Controller / @RestController
- 하위 패키지 까지 모두 찾는다

2단계 :@EnableAutoConfiguration
- spring-boot-autoconfigure > META-INF > spring.factories 안에 org.springframework.boot.autoconfigure.EnableAutoConfiguration 라는 키값 아래 AutoConfiguration들이 정의되어있다 (설정 파일들)
- 정의되있는 설정들은 조건에 따라 설정이 된다

< 자동 설정 만들기>

- Xxx-Spring-Boot-Autoconfigure 모듈 : 자동 설정
- Xxx-Spring-Boot-Starter 모듈: 필요한 의존성 정의
- 그냥 하나로 만들고 싶을 때는? = Xxx-Spring-Boot-Starter

구현 방법

1) 의존성 추가

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure-processor</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.0.3.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2) @Configuration 파일 작성
3) src/main/resource/META-INF에 spring.factories 파일 만들기
4) spring.factories 안에 자동 설정 파일 추가

@ConditionalOnMissingBean

- 덮어쓰기 방지

빈 재정의 수고 덜기

1) 의존성 추가

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

2) @ConfigurationProperties(“AAAA”)
3) @EnableConfigurationProperetiers(AAAA.class)
4) 프로퍼티 키값 자동 완성

< 내장 웹 서버>

- 스프링부트는 서버가 아님
    - 톰캣 객체 생성
    - 포트 설정
    - 톰캣에 컨텍스트 추가
    - 서블릿만들기
    - 톰캣에 서블릿 추가
    - 컨텍스트에 서블릿 맵핑
    - 톰캣 실행 및 대기

- 이 모든 과정 보다 상세하고 유연하게 설정하고 실행해주는 것이 자동설정
    - ServletWebServerFactoryAutoConfiguration(서블릿 웹 서버 생성)
        - TomcatServletWebServerFactoryCustomizer(서버 커스터마이징)
    - DispatcherServletAutoConfiguration
        - 서블릿 만들고 등록
    - 2개가 떨어져 등록돼 있음
        - 서블릿컨테이너들은 다 달라질 수 있지만 서블릿은 달리지지 않음

다른 서블릿 컨테이너로 변경
- <spring-boot-starter-web>에서 <spring-boot-starter-tomcat>을 exclusion하고 원하는 서버를 의존성 추가

웹서버 사용하지 않기
- 프로퍼티에 spring.main.web-application-type=none

포트
- 원하는 포트 설정 server.port = A
- 랜덤 포트 server.port=0
- ApplicationListner <ServletWebServerInitializedEvent>

<jar>
- 독립적으로 실행 가능
- mvn package를 하면 실행 가능한 JAR파일 하나가 생성 됨
- spring-maven-plugin이 해주는일 (패키징)
- 과거 “uber” jar 를 사용
    - 모든 클래스(의존성 및 애플리케이션)를 하나로 압축하는 방법
    - 무엇이 어디에서 온건지 알 수 없음
- 스프링 부트의 전략
    - 내장 JAR : 기본적으로 자바에는 내장JAR를 로딩하는 표준방법 X
    - 애플리케이션 클래스와 라이브러리 위치 구분
    - org.springframework.boot.loader.jar.JarFile을 사용해서 내장 JAR를 읽는다
    - org.springframework.boot.loader.Launcher를 사용해서 실행한다

<springApplication>
- 기본 로그 레벨은 INFO
- FailureAnalyzer
    - 오류 출력을 이쁘게
- 배너
    - 리소스 패키지 아래에 banner.txt | gif | jpg | png 파일을 생성
    - classpath 또는 spring.banner.location
    - ${spring-boot.version} 등의 변수를 사용할 수 있음.
    - Banner 클래스 구현하고 SpringApplication.setBanner()로 설정 가능.
    - SpringApplicationBuilder로 빌더 패턴 사용 가능
    - 배너 끄는 방법
        - app.setBannerMode(Banner.mode.OFF)
- ApplicationEvent 등록
    - ApplicationContext를 만들기 전에 사용하는 리스너는 빈으로 등록할 수 없다
        - SpringApplication.addListners()

- WebApplicationType 설정
    - SERVLET / REACTIVE / NONE
- 애플리케이션 아규먼트 사용하기
    - ApplicationArguments를 빈으로 등록해 주니까 가져다 쓰면 됨
    - -- 옵션
- 애플리케이션 실행한 뒤 뭔가 실행하고 싶을 때
    - ApplicationRunner (추천) 또는 CommandLineRunner
    - 순서 지정 가능 @Order
        - 숫자가 낮을 수록 우선순위가 높음

< 외부설정>
● properties
● YAML
● 환경 변수
● 커맨드 라인 아규먼트

프로퍼티 우선순위
1. 유저 홈 디렉토리에 있는 spring-boot-dev-tools.properties
2. 테스트에 있는 @TestPropertySource
3. @SpringBootTest 애노테이션의 properties 애트리뷰트
4. 커맨드 라인 아규먼트
5. SPRING_APPLICATION_JSON (환경 변수 또는 시스템 프로티) 에 들어있는 프로퍼티
6. ServletConfig 파라미터
7. ServletContext 파라미터
8. java:comp/env JNDI 애트리뷰트
9. System.getProperties() 자바 시스템 프로퍼티
10. OS 환경 변수
11. RandomValuePropertySource
12. JAR 밖에 있는 특정 프로파일용 application properties
13. JAR 안에 있는 특정 프로파일용 application properties
14. JAR 밖에 있는 application properties
15. JAR 안에 있는 application properties
16. @PropertySource
17. 기본 프로퍼티 (SpringApplication.setDefaultProperties)

application.properties 우선 순위
1. file:./config/
2. file:./
3. classpath:/config/
4. classpath:/

프로퍼티 랜던값
● ${random.*}

플레이스 홀더
● name = YoungSung
● fullName = ${name} Ko

#타입-세이프 프로퍼티 @ConfigurationProperties(“AAA”)

- 여러 프로퍼티를 묶어서 읽어올 수 있음
- 빈으로 등록해서 다른 빈에 주입할 수 있음
    - @EnableConfigurationProperties
    - @Component
    - @Bean
- 융통성 있는 바인딩
    - context-path (케밥)
    - context_path (언드스코어)
    - contextPath (캐멀)
    - CONTEXTPATH
- 프로퍼티 타입 컨버전
    - @DurationUnit
- 프로퍼티 값 검증
    - @Validated
    - JSR-303 (@NotNull, ...)
- 메타 정보 생성
- @Value
    - SpEL 을 사용할 수 있지만 위에 있는 기능들은 전부 사용 X

<프로파일>

- 프로퍼티에 활성화할 프로퍼티 추가
    - spring.profiles.active=aaa
- 프로파일 추가
    - spring.profiles.include
- 프로파일용 프로퍼티
    - application-{profile}.properties

<스프링 부터 2.1>

의존성 변경

- 스프링 프레임워크 5.0 -> 5.1
    - 로거 설정 개선 spring-jcl
    - 컴포넌트 인덱스
        - 컴포넌트 스캐닝 성능 개선이 가능
    - 함수형 프로그래밍 스타일 지원
    - 코틀린 지원
    - 리액티브 프로글매이 모델 지원
    - JUnit5
- JUnit4->5
    - Jupiter
    - extension모델
    - 람다 지원
- 톰캣 8.5 -> 9
    - BIO 커넥터 -> NIO 커넥터

        - 블로킹 IO / 넌블로킹 IO
    - HTTP/2지원
    - 웹소켓2
    - 서블릿4.0/JSP2.4
- 하이버네이트 5.2->5.3
    - JPA2.2 지원
    - JAVA 8 Date/Time API

<로거>
- Logging Facade
    - SLF4J (현재 거의 이거 씀)
    - JCL
- 로거
    - JUL
    - Log4j2 (main)
    - Log4j
    - Logback (main)

문제
1) 기존에 이미 다른 로깅 퍼사드나 로걸르 사용중인 프로젝트
    - SLF4J로 통하는 다리(Bridge)를 놓는다
        - ex) JCL-over-SLFJ4
              Log4j-to-SLF4J
2) SLF4J가 사용할 로거는 어떻게 정하는가
    - Binder 사용
        - ex) logback-classic

Spring-jcl
- JCL-over-SLF4J 대체제
- 클래스패스에 Log4J 2가 있다면 JCL을 사용한 코드가 Log4j 2를 사용
- 클래스패스에 SLF4J가 있다면 JCL을 사용한 코드가 SLF4j를 사용한다
- Log4J 2 를 사용할 때는 별다른 브릿지나 바인더가 필요없다
- SLF4J를 사용할 때에도 JCL을 굳이 exclusion하거나 JCL용 브릿지를 추가할 필요 없다

설정하기
- <spring-boot-starter>(<spring-boot-starter-logging>을 exclusion) 의존성 추가
- <spring-boot-starter-log4j2> 의존성 추가

<빈 오버라이딩>
1) 애플리케이션에 정의한 빈 등록
2) 자동설정이 제공하는 빈 등록
- 이때 1번에서 정의한 빈을 2번과정에 등록하는 빈이 오버라이딩 할 수도 있었는데 2.1 이후로는 허용하지 않는다 -> 오류 발생
- 프로퍼티를 변경해서 빈 오버라이딩을 허용 할 수도 있다
    - spring.main.allow-bean-definition-overriding=true
- 오버라이딩이 일어나지 않도록 자동설정 제공하는 쪽에 @Condition* 애노테이션 활용

< 자동설정 지원>
스프링 부트 2.1부터 지원하는 자동설정

1) 태스크 실행
    - @EnableAsync 사용 시 자동설정(TaskExecutionAutoConfiguration) 적용
    - spring.task.execution 프로퍼티로 제공
    - TaskExecutorBuilder 제공
2) 태스크 스케줄링
    - @EnableScheduling 사용 시 자동설정(TaskSchedulingAutoConfiguration)
    - 주기적으로 오퍼레이션 실행
    - spring.task.scheduling 프로퍼티 제공
    - TaskSchedulerBuilder 제공
3) 스프링 데이터 JDBC
    - spring-boot-starter-data-jdbc 의존성 추가시 지원
4) 기타
    - 카프카 스트림 지원
    - JMS ConnectionFactory 지원
    - 엘라스틱 서치 REST 클라이언트 지원 등

<프로퍼티 변경>

스프링 데이터 JPA 부트스트랩 모드 지원
- 애플리케이션 구동 시간을 줄이기 위해 스프링 데이터 JPA 리파지토리 생성을 지연시키는 설정
- spring.data.jpa.repositories.bootstrap-mode= ?
    1) DEFERRED : 애플리케이션 구동 이후에 리파지토리 인스턴스를 만들어 주입
    2) LAZY : 구동 이후에도 만들지 않다가 처음 사용하는 시점에 만들어 주입

프로퍼티 마이그레이션
- spring-boot-properties-migrator 의존성 추가
- 프로퍼티를 마이그레이션 하지 않더라도 기존 프로퍼티로 애플리케이션 구동이 가능하며 프로퍼티가 어떻게 바뀌었는지 알려주는 툴

<JUnit>

- 2.1부터 지원하지만 기본적으로 들어오는건 2.2부터
- @SpringBootTest 위에 @ExtensdWith 메타 어노테이션 -> @Runwith() 생략
- 모든 API가 Jupiter 에 포함

<DataSize>
- org.springframework.util.unit.DataSize
    - 스프링 부트가 아니라 스프링 프레임워크가 5.1부터 지원하는 타입
    - 지원하는 타입 : B ,KB, MB, GB, TB
- 스프링 부트는 컨버터를 지원
    - StringToDataSizeConverter
    - NumberToDataSizeconverter
- application.properties에서 데이터 사이즈를 손쉽게 바인딩 받기 가능

<로그 그룹>

- 같은 로그 레벨을 적용할 패키지 묶음을 만들 수 있는 기능으로 여러 패키지의 로그 레벨을 손쉽게 변경할 수 있다
- 로그 그룹 정희나는 방법
    - logging.group.{그룹이름}={패키지},{패키지},{패키지}...
    - logging.level.{그룹이름}={로그레벨}
- 스프링 부트가 미리 정의해둔 로그 그룹
    - web = 스프링 웹 MVC 관련 패키지 그룹
    - sql = 스프링 JDBC 와 하이버네이트 SQL을 묶어둔 로그 그룹

<Actuator>

- spring-boot-starter-actuator 의존성 추가
- /info와 /health 엔드포인트가 스프링 시큐리티를 추가하더라도 기본적으로 '공개' 하도록 변경됨
    - 스프링부트 2.0.* 에서는 스프링 시큐리티를 추가하면 모든 엔드포인트가 인증을 거쳐야 했음
        - 스프링 시큐리티 설정 추가해서 컨트롤 가능
- /info 엔드포인트에 정보 추가하는 방법
    1) Info 키값에 들어있는 모든 프로퍼티
    2) git.properties에 들어있는 프로퍼티
    3) META-INF/build-info.properties에 들어있는 프로퍼티

[스프링 부트 2.2]

주요 변경 내용

- 자바 13 지원
- 의존성 변경
- 초기화 지연을 통해 애플리케이션 구동 시간 줄이는 기능 추가
- Immutable @ConfigurationProperties
- Actuator, health 그룹 지원

스프링 HATEOAS API 변경

- 참조

< 의존성 / 프로퍼티 변경>

의존성 변경

- 스프링 프레임워크 5.2
    - 스프링 웹플럭스 기반 코틀린 코루틴 지원
    - Rsocket 지원
    - R2DBC
    - JUnit Jupiter 5.7 지원
    - @Configuration 에 proxyBeanMethods 속성 추가
- 스프링 시큐리티 5.2
    - Oauth 2.0 클라이언트 기능 추가
    - Oauth 2.0 리소스 서버 기능 추가
    - Oauth 2.0 인증서버 기능은 별도의 커뮤니티 프로젝트로 분리
- 스프링 데이터 Moore

        - 선언적인 리액티브 트랜잭션 지원(@Transactional)
        - 리액티브 QueryDSL
        - 성능향상
- Junit5

프로퍼티 변경
- logging.file -> logging.file.name
- logging.path -> logging.file.path

< 성능 개선 >

스프링 부트 애플리케이션 구동을 빠르게
- Configuration(proxyBeanMethods-false)적용
- spring.main.lazy-initialization=true
- spring.data.jpa.repositories.bootstrap-mode=lazy
- spring.jmx.enabled=false(이미 기본적용)
- 단점
    - 요청 처리 시간이 늦어질 수 있다
        - 아직초기화 하지 않은 빈을 만들기 때문
    - 애플리케이션 구동시 발생해야할 에러가 동작중 발생 할 수 있다

< @ConfigurationProperties 개선 >

@ConfigurationPropertiesScan
- SpringBootApplication 에 추가로 선언하면 편함

Immutable @ConfigurationProperties 지원
- setter 가 아닌 생성자를 사용해서 프로퍼티를 바인딩할 수 있는 기능을 지원
    - @ConfigurationPropertiesScan 또는 @EnableConfigurationProperties를 통해 빈으로 만드는 경우에만 동작 그외 다른 방법으로 빈 등록 시 적용x
    - 생성자가 여러개일 경우 바인딩에 사용할 생성자에 @ConstructorBinding 추가
    - 롬복과 같이 사용할 경우
        - @Getter @AllArgsConstructor 추가

<Actuator, health 엔드포인트>

health 엔드포인트
- 프로퍼티에 management.endpoint.health.show-components/details=?
    - never
    - always : 항상
    - when_authorized : 인증된 사용자만
- 애플리케이션의 상태 점검 용도로 사용 가능
- HealthContributor를 사용해서 커스텀한 Health 체크 추가 가능
- 스프링부트가 기본으로 다양한 구현체
- 그룹 기능 추가
    - management.endpoint.health.group.**

[스프링부트 2.3]

주요변경내용

- 자바14 지원
- 의존성과 프로퍼티 변경
- 효율적인 컨테이너 이미지 생성 방법 제공
- Liveness 와 Readiness
- Graceful과 Shutdown 지원
- Actuator, configprops 엔드포인트

<의존성 및 프로퍼티 변경>

의존성 변경

- Spring-Boot-Starter-Web에서 Validation 모듈을 가져오지 않게 변경
- Spring Data Neumann
    - 코틀린 코루틴 지원
    - MongoDB, Cassandra, Couchbase SDK, QueryDSL, Elasticsearch 버전 업그레이드
    - Spring data R2DBC 추가
- Jackson 2.11
    - Date와 Calendar 기본 포맷 중 timezone 표현하는 방법이 표준에 맞도록 변경
    - 필드 이름 없이 Array 만들지 못하도록 변경
- Spring Security 5.3
    - 문서개선
    - OAuth 2.0 클라이언트와 리소스 서버 관련 기능 개선
- JUnit Jupiter 5.6
- Mockito 3.3

프로퍼티 변경

- Period 지원
    - PeriodToStringConverter
    - StringToPeriodConverter
- spring.data.jpa.repositories.bootstrap-mode 기본으로 deferred 모드

##
** 도커부분은 나중에 **

<Graceful 셧다운>

- 애플리케이션 서버 종료시 새로운 요청은 막고 기존에 처리중이던 요청은 완전히 처리 후에 서버 종료를 지원하는 기능

```
server.shutdown=graceful
```

- 서블릿 기반 MVC 와 리액티브 스트림 기반 웹 플럭스 모두 지원
- 톰캣/제티/네티는 새요청을 네트워크 단에서 받지 않고 기존 요청을 계속 진행하지만 언더토우는 Service Unavailable(503) 응답
- 기존 요청 처리에 타임아웃 설정 가능
    - spring.lifecycle.timeout-per-shutdown-phase=20s

<Liveness / Readiness>

Liveness

- 애플리케이션이 살아있는가
- 상태가 비정상이고 복구하지 못한다면 보통 애플리케이션은 재기동 한다
- LivenessState.CORRECT
- LivenessState.BROKEN

#Readiness

- 요청 받을 준비가 되있는가
- 준비가 될때까지 해당 서버로 요청을 보내지 않고 다른 서버로 보냄
- ReadinessState.ACCEPTING_TRAFFIC
- ReadinessState.REFUSING_TRAFFIC

애플리케이션 내부에서 상태 정보 조회

```
@Autowired ApplicationAvailability availability
LivenessState livenessState = availability.getLivenessState();
Readiness readiness = availability.getReadinessState();
```

- 애플리케이션 밖에서 조회할땐 Actuator helath 사용

쿠버네티스 연동

- 쿠버네티스의 Liveness probe
    - 특정횟수(기본값 3)이상 Liveness 상태가 안좋으면 앱 재시작
- Readiness
    - Readiness 상태가 안 좋은 경우 해당 pod로 요청 보내지 안음

[2.4]

주요 변경사항

- 스프링 5.3
- 자바 15
- 의존성과 프로퍼티
- 설정파일 처리 방식 변경

<의존성과 프로퍼티>

- 스프링부트 5.3
  - LTS버전 5.3.x
  - GraalVM 개선
  - R2DBC를 지원하는 spring-r2dbc 모듈 지원
  - queryForStream 제공
- 스프링데이터 2020.0
  - 버저닝을 캘린더 기반으로 바꿈
  - RxJava 3 지원
- 스프링배치4.3
  - 성능향상

프로퍼티

- spring-profiles -> spring.config.activate.on-profile

<설정파일 처리 방식 변경>

#기존 application.properties 와 applcation.yaml의 문제들

1) application.properties는 여러 문서를 표현 할 수 없다
2) spring.profiles(설정 적용할 프로파일) 이름이 모호
3) 설정 읽어들이는 순서 복잡

해결

1) #--- 를 통해 문서구분
2) spring.profiles 대신 보다 직관적인 spring.config.activate.on-profile 사용
   - spring.config.activate.on-profile : 현재 설정을 적용할 프로파일
   - spring.profiles.active 사용할 프로파일
   - spring.profiles.include 추가로 사용할 프로파일
3) 특정 프로파일을 사용하거나추가하는 설정과 현재설정을 적용하는 프로파일을 같이 사용할 수 없다

설정 파일 추가

- spring.config.import
- spring.config.activate.on-profile과 같이 사용 가능(프로파일추가 x 설정파일 추가 o)
- 추가하는 설정파일을 제일 아래 있는 문서로 취급 -> 기존의 다른 설정 덮어씀

<Configuration Tree>

- spring.config.import 값으로는 여러 접두어를 지원하는데 아무런 접두어를 사용하지 않으면 일반적인 파일이나 디렉토리로 인식
- configtree : 접두어를 사용하면 Configuration Tree 스타일의 볼륨 기반 설정 트리를 지정할 수 있다.
- optional : 접두어를 사용하면 해당 디렉토리 또는 파일이 존재하지 않아도 에러가 발생하지 않는다

- spring.config.import=configtree:config

<클라우드 플랫폼 기반 설정>

- 특정 클라우드 플랫폼에 배포했을 때 설정 파일 사용하기
- spring.config.activate.on-clout-platform의 값으로 CloudPlatform을 사용할 수 있다

- 특정 프로파일이 아니라 특정한 클라우드 플랫폼에 배포했을 때 설정 파일을 사용하도록 설정할 수있다

  # 지원하는 클라우드 플랫폼
  - Kubernates
  - Cloud Foundary
  - Heroku
  - SAP
  - NONE

<프로파일 그룹>
- 프로파일을 세밀하게 만든 경우, 특정 프로파일 하나에 다른 여러 프로파일을 그룹으로 묶어 동시에 사용 가능
- spring.profiles.group.local=localService,localControll
  - local이라는 프로파일 사용할 때 localService와localControll 프로파일도 사용