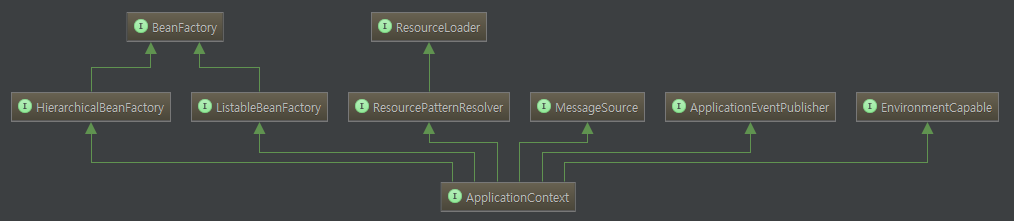
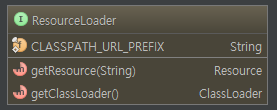
**ApplicationContext 전체 구조**

****

**1) ResourceLoader**

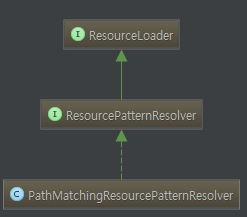
****

Resource 인스턴스를 로딩할 수 있는 객체

**Resource 종류**

|  |  |
| --- | --- |
| **Resource 명** | **설 명** |
| UrlResource | URL로 얻어와야 하는 리소스 (java.net.URL) |
| ClassPathResource | 클래스패스(resource)에서 얻어와야 하는 리소스 |
| FileSystemResource | file:/ (java.io.File) |
| ServletContextResource | 웹 어플리케이션의 루트 경로 |
| InputStreamResource | InputStream에 대한 리소스 (파일, 버퍼, 데이터) |
| ByteArrayResource | 바이트 배열에 대한 리소스 |

**2) ResourcePatternResolver extends ResourceLoader**

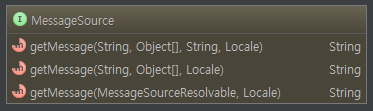


**얜 뭐임?**

앤트스타일로 읽어 오는 애임 ㅇㅇ

ex) "/WEB-INF/\*-context.xml")

**3) MessageSource**



다국어 처리 할 때 사용한다. (로케일 사용)

ex)

**message\_ko\_kr.properties**

안녕하세요.

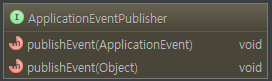
**message\_en\_us.properties**

Hello

**message\_ja\_jp.properties**

こんにちわ

**4) ApplicationEventPublisher**



이벤트 지원

|  |  |
| --- | --- |
| **이벤트 종류** | **용도** |
| ApplicationEventPublisher | 이벤트 발신자 |
| ApplicationListener | 이벤트 수신자 |
| ApplicationEvent | 이벤트 |

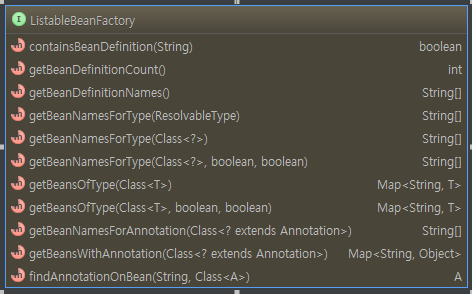
**5) EnvironmentCapable**



Profile에 대한 선택, property에 대한 통합접근 제공

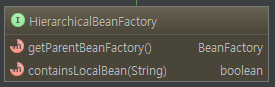
GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();  
ctx.getEnvironment().setActiveProfiles("dev");  
ctx.load("classpath:/\*-config.xml");  
ctx.refresh();

**6) ListableBeanFactory (열거하다) extends BeanFactory**



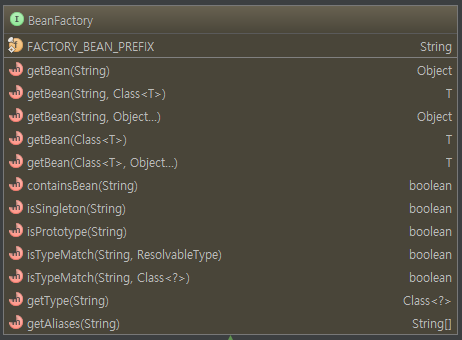
BeanFactory의 서브인터페이스 (기능 확장)

**7) HierarchicalBeanFactory (계층적) extends BeanFactory**



BeanFactory의 서브인터페이스 (기능 확장)

**8) BeanFactory (lazyloading)**



Bean의 생성과 소멸 담당

Bean 생성 시 필요한 속성 설정

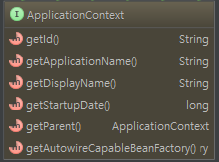
Bean의 LifeCycle에 관련된 메소드 호출

다수의 BeanFactory 인터페이스 구현 클래스를 제공

BeanFactory factory = new XmlBeanFactory(new FileInputStream("bean.xml"));  
MyBean myBean = (Mybean) factory.getBean("myBean");

getBean이 호출되면, 팩토리는 의존성 주입을 이용해 빈을 인스턴스화 한다.

**9) ApplicationContext (preloading)**



BeanFactory의 모든 기능 제공

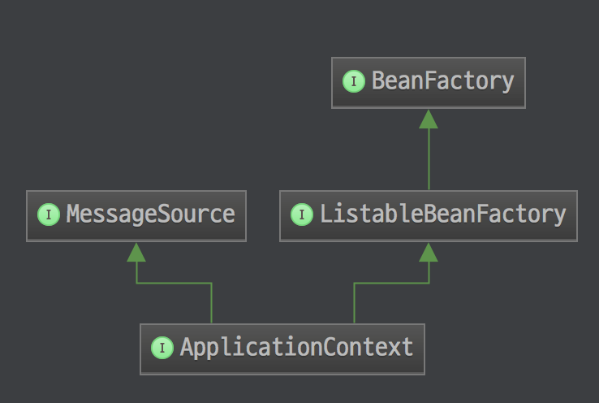
ResourceBundle 파일을 이용한 국제화(I18N) 지원

다양한 Resource 로딩 방법 제공

이벤트 핸들링

**스프링의 히스토리**

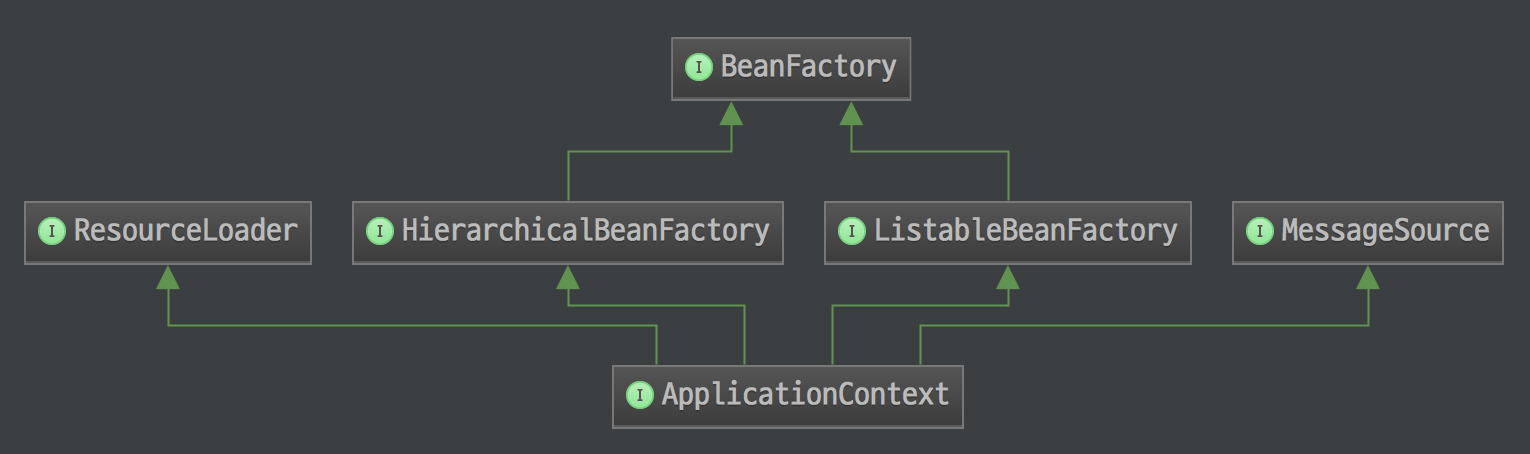
**ver 0.9**



BeanFactory의 Sub Interface인 **HierarchicalBeanFactory** 가 없었고, ApplicationContext의

초기 기능은 **MessageSource**(국제화 뿐이였다.)

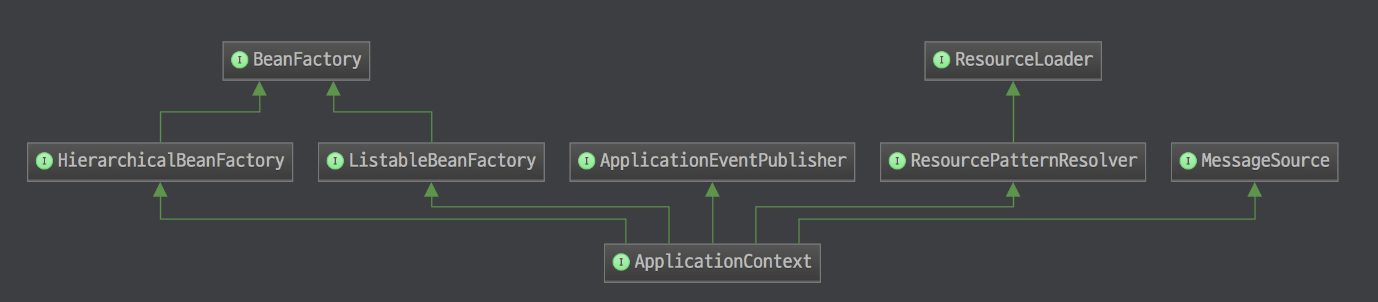
**ver 1.0**



BeanFactory의 Sub Interface인 **HierarchicalBeanFactory**가 추가되었고,

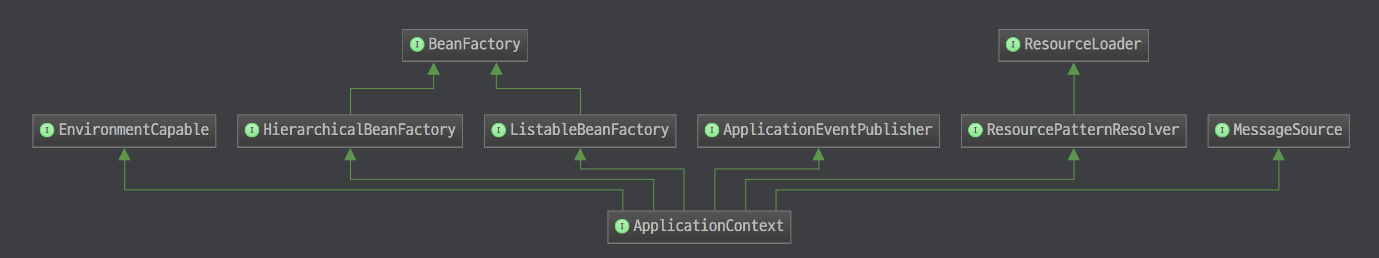
**ResourceLoader**가 추가되었다.

**ver 2.0**



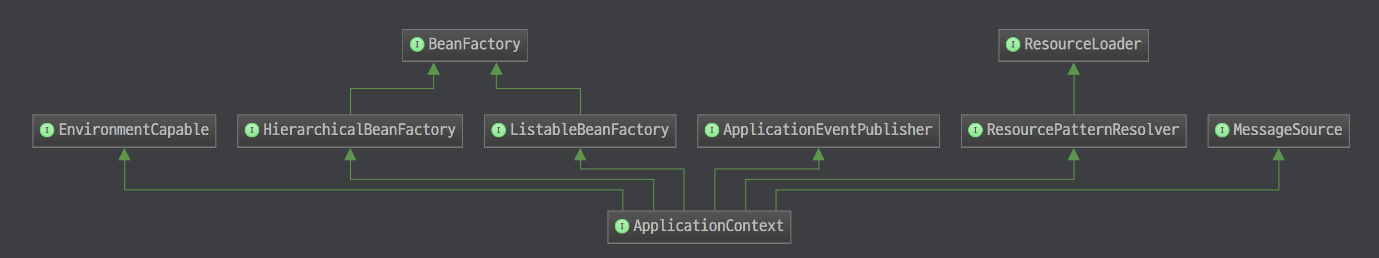
**ResourcePatternResolver**(앤트처럼 리소스 불러오는애) 가 추가되었고, **ApplicationEventPublisher**가 추가되었다.

**ver 3.0**



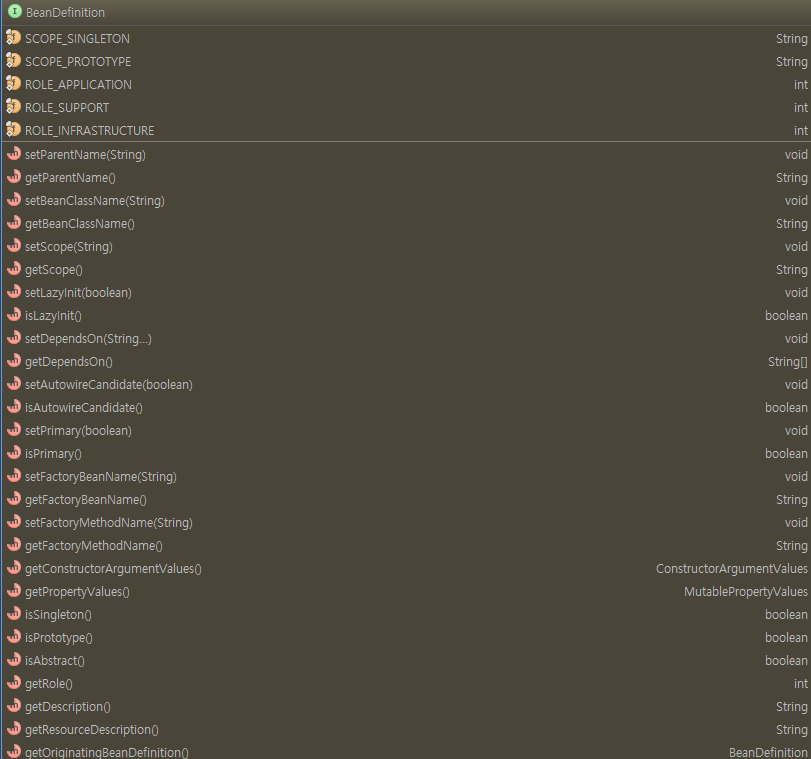
**EnvironmentCapable**이 추가되었다.

**ver 4.0 (동)**



**BeanDefinition과 BeanDefinitionReader**

**BeanDefinition**

****

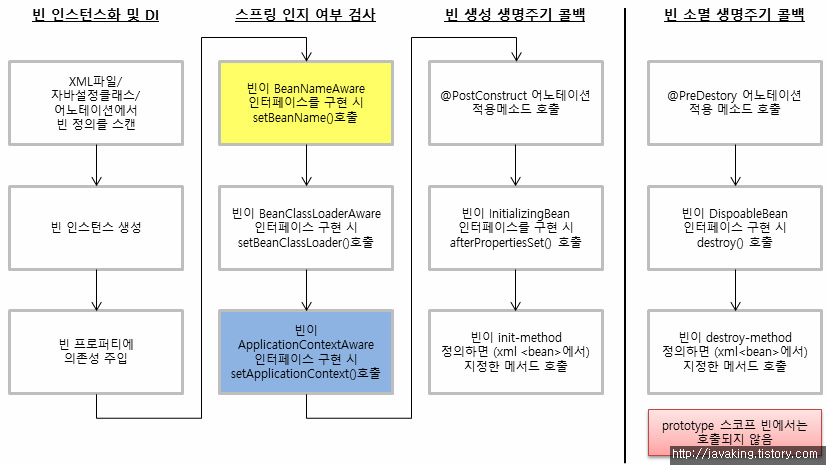


**BeanDefinition**

Spring의 빈 설정은 XML과 JavaConfig(Annotation)으로 구성되어 잇다.

이 말은 사실 BeanDefinition으로 추상화되어 있는 빈설정을 xml, JavaConfig로 표현하고 있다고 말할 수 있다.

**Bean LifeCycle**



**빈 인스턴스화 및 DI**

**1. XML 파일/ 자바설정클래스/ 어노테이션에서 빈 정의를 스캔**

XML 방식

<context:component-scan base-package="com.tmoncorp"/>

JavaConfig 방식

@Configuration  
@EnableWebMvc  
@ComponentScan("com.sjb.chapter5.web")  
public class WebConfig extends WebMvcConfigurerAdapter {

...

}

**2. 빈 인스턴스 생성**

**3. 빈 프로퍼티에 의존성 주입**

<bean id="testService" class="com.ljw.Service">  
 <constructor-arg>  
 <value> 10 </value>  
 </constructor-arg>  
</bean>  
  
<bean id="testService" class="com.ljw.Service">  
 <constructor-arg value="10"/>  
</bean>

**스프링 인지 여부 검사**

**1. 빈이 BeanNameAware 인터페이스를 구현 시, setBeanName() 호출**

<!—applicationContext.xml ->

<bean id="beanName" name="simjunbo" class="com.sjb.core.BeanName" />

// Java  
public class BeanName implements BeanNameAware {  
 @Override  
 public void setBeanName(String name) {  
 System.*out*.println("bean name is : " + name);  
 }  
}

스프링에서 관리되는 bean 내부에서 id나 name이 무엇으로 지정되어 있는지 확인이 필요한 경우. 로그용..트래킹 (bean 생성과, property 주입을 완료 후, init method를 수행하기 전 호출)

**2. 빈이 BeanClassLoaderAware 인터페이스 구현 시, setBeanClassLoader() 호출**

<!—applicationContext.xml ->

<bean id="loader" class="com.sjb.core.Loader"/>

// Java  
public class Loader implements BeanClassLoaderAware {  
 @Override  
 public void setBeanClassLoader(ClassLoader classLoader) {  
 System.*out*.println("parent class loader : "+classLoader.getParent());  
 }  
}

클래스들을 로드하는 클래스로더

**3. BeanFactoryAware의 setBeanFactory() 호출**

<!—applicationContext.xml ->

<bean id="beanFactoryAware" class="com.sjb.core.BeanFactoryAwareExample"/>

// Java  
public class Loader implements BeanClassLoaderAware {  
 private BeanFactory beanFactory;  
  
 @Override  
 public void setBeanFactory(BeanFactory beanFactory) throws BeansException {  
 this.beanFactory = beanFactory;  
 }  
  
 public AccountRepository getAccountRepository() {  
 return beanFactory.getBean(AccountRepository.class);  
 }  
}

어떤 클래스에서 bean을 꺼낼(look up) 수 있는factory가 필요하다면 사용.

한마디로 bean 정보 꺼내올 때 사용.

**4. 빈이 ApplicationContextAware 인터페이스 구현 시 setApplicationContext() 호출**

<!—applicationContext.xml ->

<bean id="applicationContextAware" class="com.sjb.core.ApplicationContextAwareExample"/>  
<bean id="Sample" class="com.sjb.core.Sample" />

// Java  
public class ApplicationContextAwareExample implements ApplicationContextAware {  
 @Override  
 public void setApplicationContext(ApplicationContext context) throws BeansException {  
 Sample sample = (Sample) context.getBean("Sample");  
 sample.test();  
 }  
}

빈이 실행되는 환경인 ApplicationContext 인스턴스에 접근할 수 있다.

이것도 bean 정보가 필요 할 때 사용하는듯.. 다른점은 BeanFactory와 ApplicationContext 차이?

**빈 생성 생명주기 콜백**

**1. BeanPostProcessor의 사전 초기화**

<!—applicationContext.xml ->

<bean id="beanPostProcessor" class="com.sjb.core.BeanPostProcessorExample"/>

// Java  
public class BeanPostProcessorExample implements BeanPostProcessor {  
  
 @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 if("sample".equals(beanName)){  
 Sample sample = (Sample) bean;  
 sample.test2();  
 }  
 return bean;  
 }  
  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 return bean;  
 }  
}

ApplicationContext이 관리하는 모든 빈에 대해서 Initializing 전/후 시점으로 수행할 프로세스 등록

**2. @PostConstruct 어노테이션 적용 메소드 호출 (JSR-250)**

@PostConstruct를 이용하면 Bean 객체를 생성한 이후에 초기화를 수행할 메소드를 지정할 수 있다.

<!—applicationContext.xml ->

<bean id="customerService" class="com.sjb.core.CustomerService">  
 <property name="message" value="i'm Groot"/>  
</bean>

@PostConstruct  
public class CustomerService {  
 String message;  
  
 public void setMessage(String message) {  
 this.message = message;  
 }  
  
 @PostConstruct  
 public void initIt() throws Exception {  
 System.*out*.println("@PostConstruct : " + message);  
 }  
  
 @PreDestroy  
 public void cleanUp() throws Exception {  
 System.*out*.println("@PreDestroy");  
 }  
}

**3. InitializingBean 인터페이스를 구현 시 afterPropertiesSet() 호출**

<!—applicationContext.xml ->

<bean id="PreDestroy" class="com.sjb.core.PreDestroyExample"/>

// Java

public class InitializingBeanExample implements InitializingBean {  
  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 System.*out*.println("InitializingBean");  
 }  
}

**4. init-method**

<bean id="InitMethod" class="com.sjb.core.InitMethodExample" init-method="init"/>  
public class InitMethodExample {  
 public void init() {  
 System.*out*.println("init-method");  
 }  
}

**빈 소멸 생명주기 콜백**

**1. @PreDestory 어노테이션 적용 메소드 호출**

<!—applicationContext.xml ->

<bean id="PreDestroy" class="com.sjb.core.PreDestroyExample"/>

// Java  
public class PreDestroyExample {  
 @PreDestroy  
 public void cleanUp() throws Exception {  
 System.*out*.println("@PreDestroy");  
 }  
}

@PreDestory는 Bean 객체의 라이프 사이클이 종료되기 직전에 사용한 자원의 해제할 필요가 있을 경우등에 사용.

**2. 빈이 DisposableBean 인터페이스 구현 시 destroy() 호출**

<!—applicationContext.xml ->

<bean id="disposableBean" class="com.sjb.core.DisposableBeanExample"/>

// Java

public class DisposableBeanExample implements DisposableBean {  
  
 @Override  
 public void destroy() throws Exception {  
 System.*out*.println("destroy");  
 }  
}

**Bean과 Component 차이**

@Bean의 경우 개발자가 컨트롤이 불가능한 외부 라이브러리들을 Bean으로 등록하고 싶을 때 사용 (ex ObjectMapper)

@Component는 개발자가 컨트롤 가능한 Class들 (Controller, Service, Repository)