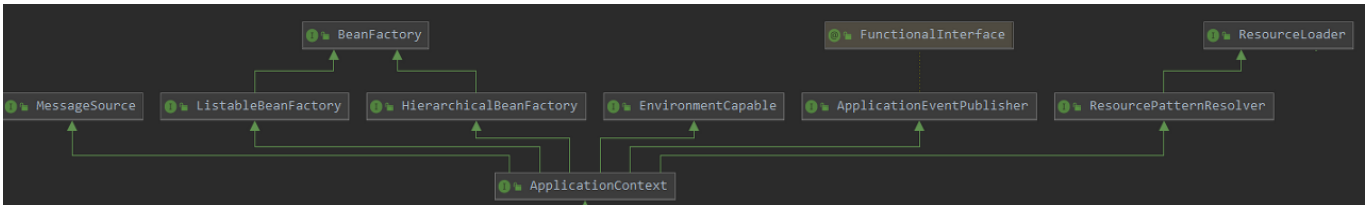
소스코드

다양한 종류의 애플리케이션 컨텍스트

- 웹 애플리케이션이 아닌 경우
  - 애플리케이션 컨텍스트 : AnnotationConfigApplicationContext
  - 웹 서버 x
- 서블릿 기반의 웹 애플리케이션
  - 애플리케이션 컨텍스트 : AnnotationConfigServletWebServerApplicationContext
  - 웹 서버: TOMCAT
- 리액티브 웹 애플리케이션인 경우
  - 애플리케이션 컨텍스트: AnnotationConfigReactiveWebServerApplicationContext
  - 웹서버: Reactor Netty
- 애플리케이션 컨텍스트
  - 관련 클래스들은 spring-context 프로젝트에 존재, spring-core, spring-aop, spring-bean을 추가하면 같이 불러와짐
  - AnnotationConfigServletWebServerApplicationContext나 AnnotationConfigReactiveWebServerApplicationContext는 Springboot에 추가된 클래스이므로, spring-boot-starter-web 또는 spring-boot-starter-webflux 같은 spring-boot 관련 의존성을 추가해야함.

DI 컨테이너와 애플리케이션 컨텍스트

- 애플리케이션 컨텍스트 : 애플리케이션을 실행하기 위한 환경
- DI컨테이너라 불리는 이유는 BeanFactory 인터페이스를 부모로 상속받았기 때문



BeanFactory

- 애플리케이션 컨텍스트의 최최상위 인터페이스, 1개의 빈을 찾기위한 메소드를 가짐

```
public interface BeanFactory{
    String FACTORY_BEAN_PREFIX="&"
}
Object getBean(String name) throws BeansException;

<T> T getBean(String name, Class<T> requiredType) throws BeansException;

Obejct getBean(String name, Object...args) throws BeansException;

<T> T getBean(Class<T> requiredType)throws BeansException;
```

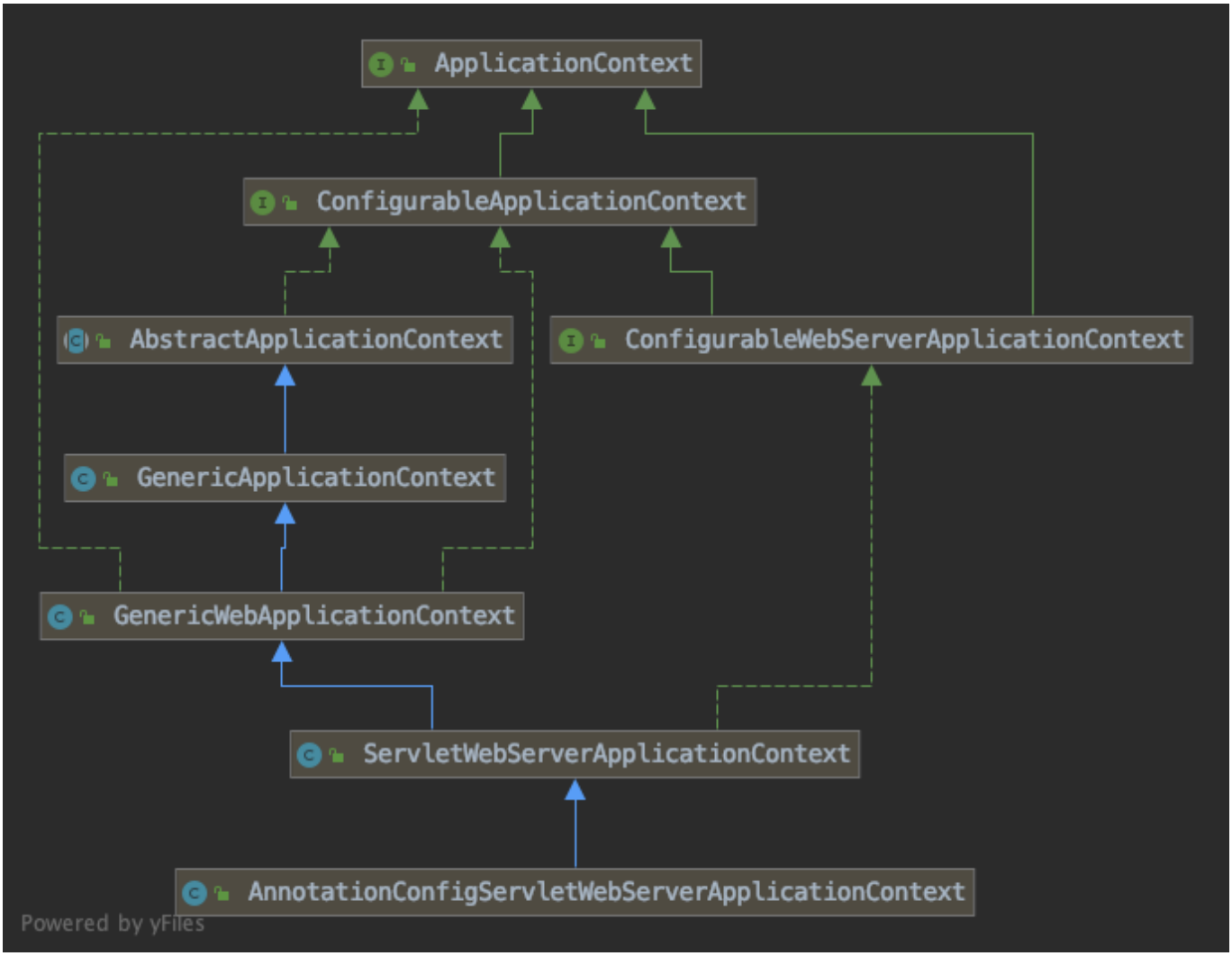
```
<T> T getBean(Class<T> requiredType, Object...args) throws BeansException;

<T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);

<T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType);

boolean containsBean(String name);

boolean isSingleton(String name) throws NoSuchBeanDefinitionException;

boolean isPrototype(String name) throws NoSuchBeanDefinitionException;

boolean isTypeMatch(String name, ResolvableType typeToMatch) throws
NoSuchBeanDefinitionException;

boolean isTypeMatch(String name, Class<?> typeToMatch) throws
NoSuchBeanDefinitionException;

@Nullable
Class<?> getType(String name) throws NoSuchBeanDefinitionException;

@Nullable
Class<?> getType(String name, boolean allowFactoryBeanInit) throws
NoSuchBeanDefinitionException;

String[] getAliases(String name);
```

- 동일한 타입의 빈이 여러개 존재할때에 List로 빈을 찾아 주입해줌.
    - ListableBeanFactory
- 여러 BeanFactory들 간의 계층 관계를 설정해줌
    - HierarchicalBeanFactory
- 하나의 빈을 처리해줌
    - BeanFactory
- @AutoWire처리를 위한 빈 팩토리
    - AutowireCapableBeanFactory
    - 얘는 상속받지 않았는데 사용할수 있는 이유는 컨텍스트 내에서 처리되기 때문
- 때문에 컨텍스트는 단일빈 외에도 다양하게 처리가 가능하다.

```
public interface ApplicationContext extends EnvironmentCapable,
ListableBeanFactory,
        HierarchicalBeanFactory, MessageSource, ApplicationEventPublisher,
ResourcePatternResolver {

    @Nullable
    String getId();

    String getApplicationName();

    String getDisplayName();
```

```
    long getStartupDate();

    @Nullable
    ApplicationContext getParent();

    AutowireCapableBeanFactory getAutowireCapableBeanFactory() throws
IllegalStateException;
}
```

- 다만 스프링의 빈들이 진짜로 애플리케이션 컨텍스트에서 관리되는 것은 아님
- 일반적으로 스프링 부트가 만들어내는 3가지 애플리케이션 컨텍스트 모두 GenericApplicationContext라는 애플리케이션 컨텍스트를 부모로 가지고있음



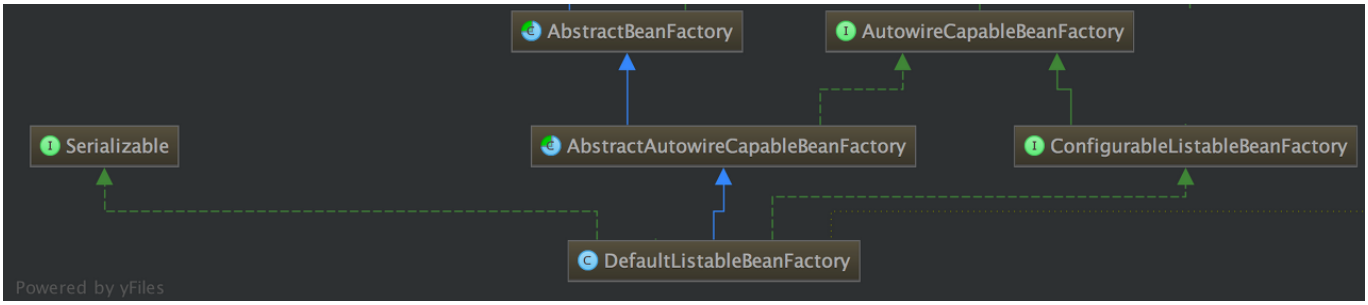
GenericApplicationContext

- 진짜 빈들을 등록하여 관리하고 찾아주는 DefaultListableBeanFactory를 생성

```
public class GenericApplicationContext extends AbastractApplicationContext
implements BeanDefinitionRegistry{
    private final DefaultListableBeanFactory beanFactory;

    @Nullable
```

```
    private ResourceLoader resourceLoader;

    private boolean customClassLoader=false;

    private final AtomicBoolean refreshed = new AtomicBoolean();

    public GenericApplicationContext(){
        this.beanFactory=new DefaultListableBeanFactory();
    }

    public GenericApplicationContext(DefaultListableBeanFactory
beanFactory){
        Assert.notNull(beanFactory,"BeanFactory must not be null")
    }

    public GenericApplicationContext(@Nullable ApplicationContext parent)
{
        this();
        setParent(parent);
    }

    public GenericApplicationContext(DefaultListableBeanFactory
beanFactory, ApplicationContext parent) {
        this(beanFactory);
        setParent(parent);
    }
}
```

- 애플리케이션 컨텍스트에 빈을 등록하거나 찾아달라는 빈 처리 요청이 오면, BeanFactory로 이러한 요청을 위임해 처리
- @AutoWire처리를 해주는 빈 팩토리를 반환하는 getAutowireCapableBeanFactory를 호출하면, DefaultListableBeanFactory이다.



ConfigurableApplicationContext

- 거의 모든 애플리케이션 컨텍스트가 갖는 공통 인터페이스
  - ApplicationContext, LifeCycle, Closable 인터페이스를 상속받음.
- 앞서 살펴본 3가지 인터페이스 모두 ConfigurableApplicationInterface를 직접 구현함.
  - GenericApplicationContext
  - ServletWebServerApplicataionContext
  - AnnotationConfigServletWebServerApplicationContext

- 스프링 부트 애플리케이션을 실행하는 run메소드를 호출하면 받는 반환 타입이 ConfigurableApplicationContext 이다.
- ConfigurableApplicationContext는 Closable을 상속받아, try-with-resources를 사용할 수 있게된다.

```
@SpringBootApplication(proxyBeanMethods = false)
public class TestingApplication {

    public static void main(String[] args) {
        try (ConfigurableApplicationContext ctx =
SpringApplication.run(TestingApplication.class,args)) {

        } catch (Exception e) {

        }
    }
}
```

- try-with-resources에 의해 자동 close메소드가 호출됨.
- 때문에 1번 실행후 자동종료가 가능

```
@SpringBootApplication(proxyBeanMethods = false)
public class TestingApplication {

    public static void main(String[] args) {
        try (ConfigurableApplicationContext ctx =
SpringApplication.run(TestingApplication.class,args)) {

        } catch (Exception e) {

        }
    }
}
```