외부 라이브러리 클래스를 빈으로 등록하기 실습

실습 환경

- https://github.com/slipp/jwp-basic 저장소를 자신의 계정으로 fork한다.
- Fork한 저장소를 https://youtu.be/xid_GG8kL_w 동영상 참고해 로컬 개발 환경을 구축한다.
- jwp-basic 저장소의 **step13-3rd-di-framework** 브랜치로 변경한다.
  - 브랜치를 변경하는 방법은 https://youtu.be/VeTjDYI7UVs 동영상을 참고한다.
- src/test/java 디렉토리의 next.WebServerLauncher를 실행한 후 브라우저에서 http://localhost:8080으로 접속해 질문/답변 게시판 서비스 화면이 나타나는지 확인한다.

요구사항

JdbcTemplate를 분석하다보니 데이터베이스 Connection을 생성하는 부분도 static으로 구현되어 있다. 또한 데이터베이스 설정 정보 또한 하드 코딩으로 관리하고 있어 특정 데이터베이스에 종속되는 구조로 구현되어 있다. 데이터베이스에 종속되지 않도록 구현하고 Connection Pooling을 지원하기 위해 Connection 대신 javax.sql.DataSource 인터페이스에 의존관계를 가지도록 지원하면 좋겠다.

이 문제를 해결하는 좋은 방법은 개발자가 직접 빈을 생성해 관리할 수 있는 별도의 설정 파일을 만드는 방법이다. 예를 들어 설정 파일에 빈 인스턴스를 생성하는 메소드를 구현해 놓고 애노테이션으로 설정한다. DI 프레임워크는 이 설정 파일을 읽어 BeanFactory에 빈으로 저장할 수 있다면 ClasspathBeanDefinitionScanner을 통해 등록한 빈과 같은 저장소에서 관리할 수 있겠다.

설정 파일 예제

```
@Configuration
@ComponentScan({ "next", "core" })
public class MyConfiguration {
    @Bean
    public DataSource dataSource() {
        BasicDataSource ds = new BasicDataSource();
        ds.setDriverClassName("org.h2.Driver");
        ds.setUrl("jdbc:h2:~/jwp-basic;AUTO_SERVER=TRUE");
        ds.setUsername("sa");
        ds.setPassword("");
        return ds;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

요구사항

자바 클래스가 설정 파일이라는 표시는 @Configuration으로 한다. 각 메소드에서 생성하는 인스턴스가 BeanFactory에 빈으로 등록하라는 설정은 @Bean 애노테이션으로 한다.

ClasspathBeanDefinitionScanner에서 사용할 기본 패키지에 대한 설정을 하드코딩했는데 설정 파일에서 @ComponentScan으로 설정할 수 있도록 지원하면 좋겠다.

위와 같이 @Configuration 설정 파일을 통해 등록한 빈과 ClasspathBeanDefinitionScanner를 통해 등록한 빈 간에도 DI가 가능해야 한다.

1단계 힌트

—

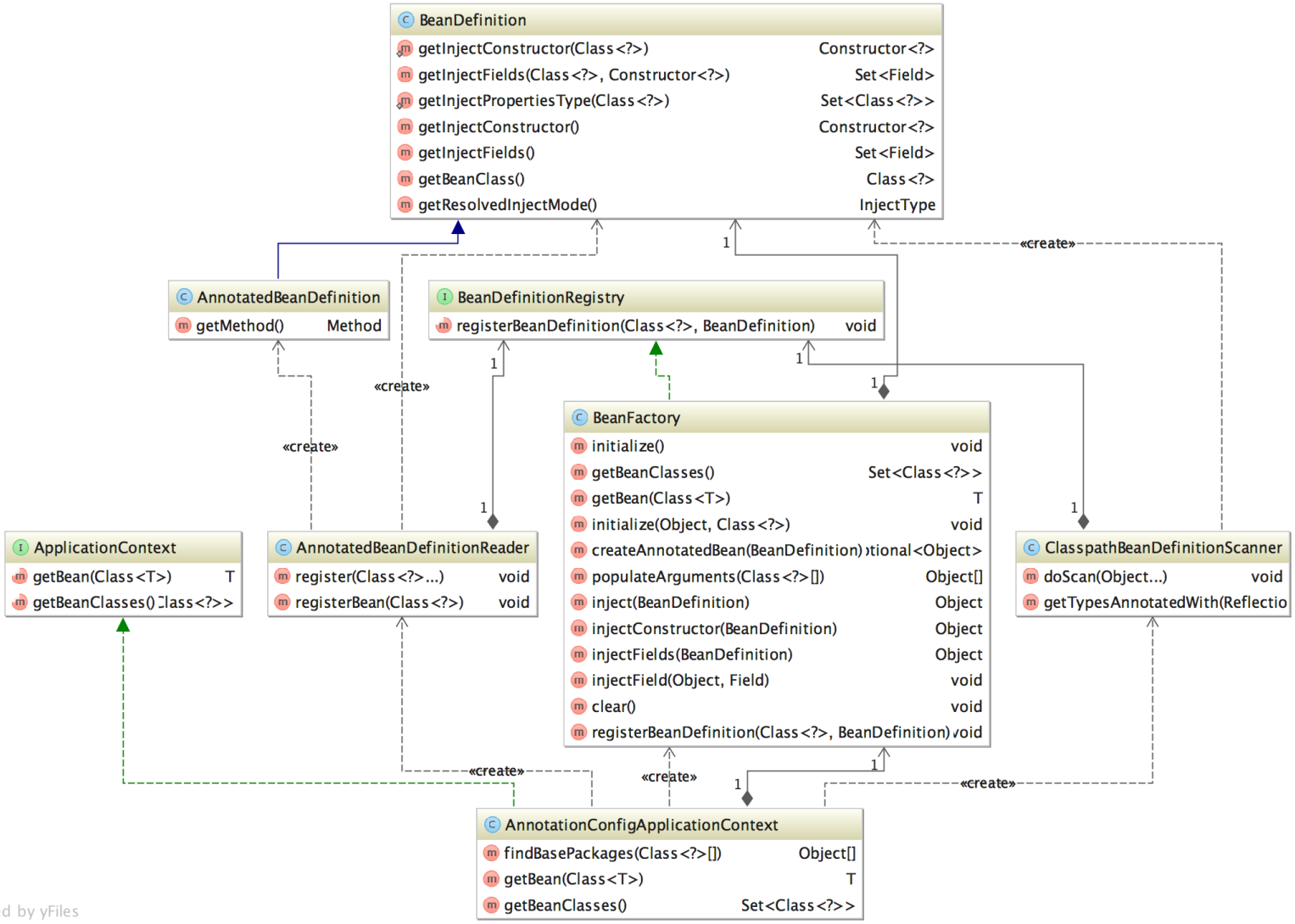
@Configuration 설정 파일을 읽어 BeanFactory에 BeanDefintion을 등록하는 역할을 하는 새로운 AnnotatedBeanDefinitionReader 클래스를 추가한다. AnnotatedBeanDefinitionReader는 @Configuration가 설정되어 있는 클래스에서 @Bean이 설정되어 있는 메소드를 찾는다. BeanFactory에 등록할 빈은 @Bean으로 설정되어 있는 메소드의 반환 값에 해당하는 클래스이다. 따라서 메소드 반환 클래스에 해당하는 BeanDefintion을 생성한 후에 BeanFactory에 등록하면 된다.

BeanDefintion을 생성할 때 고려할 부분은 빈 인스턴스를 생성하려면 @Bean으로 설정되어 있는 메소드(리플렉션의 Method) 정보가 필요하다. 따라서 기존의 BeanDefinition을 상속하는 AnnotatedBeanDefinition를 추가한 후 메소드 정보까지 같이 전달해야 한다.

이 과정까지 끝나면 가장 중요한 BeanFactory의 getBean() 메소드를 구현해야 한다. getBean() 메소드는 인자로 전달되는 빈 클래스에 해당하는 BeanDefinition이 AnnotatedBeanDefinition 인스턴스일 경우 기존과 다른 방식으로 빈 인스턴스를 생성하면된다.

마지막 작업은 ApplicationContext가 ClasspathBeanDefinitionScanner에서 사용할 기본 패키지 정보를 받는 것이 아니라 @Configuration에 설정되어 있는 설정 파일을 인자로 받도록 수정한 후 @ComponentScan 값을 구해 ClasspathBeanDefinitionScanner로 전달하고 AnnotatedBeanDefinitionReader을 통해 설정 파일에 등록된 @Bean을 BeanFactory에 등록하도록 구현해야 한다.

구현을 완료했을 때의 클래스 다이어그램



2단계 힌트

—

실습

다음과 같은 단위 테스트 코드 만들어 pass하도록 구현한다.

힌트

```
public class AnnotatedBeanDefinitionReaderTest {
    @Test
    public void register_simple() {
        BeanFactory beanFactory = new BeanFactory();
        AnnotatedBeanDefinitionReader abdr = new AnnotatedBeanDefinitionReader(beanFactory);
        abdr.register(ExampleConfig.class);
        beanFactory.initialize();

        assertNotNull(beanFactory.getBean(DataSource.class));
    }
}
```

힌트

```
@Configuration
public class ExampleConfig {
    @Bean
    public DataSource dataSource() {
        BasicDataSource ds = new BasicDataSource();
        ds.setDriverClassName("org.h2.Driver");
        ds.setUrl("jdbc:h2:~/jwp-basic;AUTO_SERVER=TRUE");
        ds.setUsername("sa");
        ds.setPassword("");
        return ds;
    }
}
```

실습

BeanFactory에 BeanDefinition 추가한다.

힌트

```
public class AnnotatedBeanDefinition extends BeanDefinition {
    private Method method;

    public AnnotatedBeanDefinition(Class<?> clazz, Method method) {
        super(clazz);
        this.method = method;
    }

    public Method getMethod() {
        return method;
    }
}
```

실습

BeanFactory에 BeanDefinition을 추가했으니 빈 인스턴스 생성과 의존관계를 주입해야 한다. BeanFactory의 getBean() 메소드에서 AnnotatedBeanDefinition을 고려해 개발한다.

힌트

```
public <T> T getBean(Class<T> clazz) {
    Object bean = beans.get(clazz);
    if (bean != null) {
        return (T)bean;
    }

    BeanDefinition beanDefinition = beanDefinitions.get(clazz);
    if (beanDefinition != null && beanDefinition instanceof AnnotatedBeanDefinition) {
        bean = createAnnotatedBean(beanDefinition);
        beans.put(clazz, bean);
        return (T)bean;
    }

    [...]
}
```

실습

ApplicationContext에서 @Configuration 설정 파일을 생성자 인자로 전달받을 수 있도록 수정하고 ClasspathBeanDefinitionScanner에서 사용할 기본 패키지 정보를 @ComponentScan에서 찾는다.

힌트

```
public class AnnotationConfigApplicationContext implements ApplicationContext {
    [...]

    private Object[] findBasePackages(Class<?>[] annotatedClasses) {
        List<Object> basePackages = Lists.newArrayList();
        for (Class<?> annotatedClass : annotatedClasses) {
            ComponentScan componentScan = annotatedClass.getAnnotation(ComponentScan.class);
            if (componentScan == null) {
                continue;
            }
            for(String basePackage : componentScan.value()) {
                log.info("Component Scan basePackage : {}", basePackage);
            }
            basePackages.addAll(Arrays.asList(componentScan.value()));
        }
        return basePackages.toArray();
    }
}
```

실습

AnnotationHandlerMapping에서 ApplicationContext를 직접 생성하고 있는데 ApplicationContext도 DI 하도록 수정한다. 이 둘 사이의 관계를 느슨한 결합으로 유지하려면 ApplicationContext라는 인터페이스를 추출하고, 기존의 ApplicationContext 클래스를 AnnotationConfigApplicationContext으로 이름을 변경한다.

힌트

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        ApplicationContext ac = new AnnotationConfigApplicationContext(MyConfiguration.class);
        AnnotationHandlerMapping ahm = new AnnotationHandlerMapping(ac);
        ahm.initialize();
        [...]
    }
}
```