Spring DM Reference Korean

번역, 편역, 요약, 편집, 배포: 백기선

블로그: http://whiteship.tistory.com

이메일: whiteship2000@gmail.com

KSUG 포럼: http://forum.ksug.org/

KSUG 블로그: http://www.ksug.org/

상업적인 용도만 아니라면, 수정 및 배포 자유롭습니다. 본문에 사소한 오타나 잘못된 띄어쓰기부터 어색한 의역 또는 잘못된 내용까지 모든 지적을 환영합니다.

Space Details

Key:	WT
Name:	Writing
Description:	
Creator (Creation Date):	keesun (3월 22, 2008)
Last Modifier (Mod. Date):	keesun (3월 22, 2008)

Available Pages

- 스프링 DM 레퍼런스
  - 1장. 왜 스프링 DM 인가?
  - 2장. 필요한 것
  - 3장. 새로 추가된 내용
    - 3.1. 1.1.X
      - 3.1.1. Web Support
        - 3.1.1.1. Spring-MVC 통합
      - 3.1.2. 클래스패스 Resource 추상화
      - 3.1.3. 변경이 편리한 Extender 설정
      - 3.1.4. 향상된 클래스 로딩
  - 4장. 번들과 Application Context
    - 4.1. 스프링 DM Extender 번들
    - 4.2. Appication Context 생성
      - 4.2.1. 필수 서비스 의존성들
      - 4.2.2. Application Context 서비스 공개하기
    - 4.3. 번들 라이프사이클
    - 4.4. 리소스 추상화
    - 4.5. BundleContext에 접근하기
    - 4.6. Application Context 없애기
    - 4.7. Extender 번들 멈추기
  - 5장. 스프링 기반 OSGi 애플리케이션 패키징기와 배포하기
    - 5.1. 번들 형식과 Manifest 헤더
    - 5.2. Extender 설정 옵션
      - 5.2.1. Listening to Extender events
    - 5.3. 필요한 스프링 프레임워크와 스프링 DM 번들들
    - 5.4. 스프링 XML 지원 기능
    - 5.5. 패지키 가져오기와 공개하기
    - 5.6. 외부 라이브러리를 사용할 때 고려해야 할 것
    - 5.7. 문제 진단하기
  - 6장. 서비스 레지스트리
    - 6.1. 스프링 빈을 OSGi 서비스로 공개하기
      - 6.1.1. Controlling the set of advertised service interfaces for an exported service
        - 6.1.1.1. Detecting the advertised interfaces at runtime
      - 6.1.2. 공개할 서비스에 프로퍼티 설정하기
      - 6.1.3 depends-on 설정
      - 6.1.4 context-class-loader 속성

        - 6.1.5. ranking 속성
        - 6.1.6. service 엘리먼트 속성
        - 6.1.7. 서비스 등록과 해지 라이프사이클
            - 6.1.7.1. OsgiServiceRegistrationListener 인터페이스
    - 6.2. OSGi 서비스 레퍼런스 정의하기
        - 6.2.1. 단일 서비스 참조하기
            - 6.2.1.1. 가져온 서비스의 인터페이스 제어하기
            - 6.2.1.2. filter 속성
            - 6.2.1.3. bean-name 속성
            - 6.2.1.4. cardinality 속성
            - 6.2.1.5. depends-on 속성
            - 6.2.1.6. context-class-loader 속성
            - 6.2.1.7. reference 엘리먼트 속성
            - 6.2.1.8. reference와 OSGi 서비스 동적특성Dynamics
            - 6.2.1.9. 관리하고 있는 서비스 레퍼런스 참조하기
        - 6.2.2. 서비스 콜렉션 참조하기
            - 6.2.2.1. Greedy Proxing
            - 6.2.2.2. 콜렉션(list와 set) 엘리먼트 속성
            - 6.2.2.3. list, set 그리고 OSGi 서비스 동적특성
            - 6.2.2.4. Iterator 제약사항과 서비스 콜렉션
        - 6.2.3. 가져온import OSGi 서비스의 동적특성 다루기
        - 6.2.4. 리스너와 서비스 프록시들
        - 6.2.5. 호출하는 BundleContext에 접근하기
    - 6.3. Exporter, Importer listener 베스트프랙티스
        - 6.3.1. 리스너와 cyclic depencdency
    - 6.4. Service importer global defaults
    - 6.5. 서비스 Exporter와 서비스 Importer의 관계
- 7장. 번들 다루기
- 8장. 웹 지원기능
    - 8.1. 지원하는 웹 컨테이너
    - 8.2. Web 지원기능 사용법
    - 8.3. OSGi 내에서 WAR 클래스패스
        - 8.3.1. Static Recsources
        - 8.3.2. 서블릿
        - 8.3.3. JSP
            - 8.3.3.1. 태그 라이브러리들
    - 8.4. web extender 설정하기
        - 8.4.1. 웹 배포자 변경하기
    - 8.5. 표준 배포자 커스터마이징 하기
    - 8.6. OSGi에 배포할 수 있는 라이브러리와 웹 개발
        - 8.6.1. Deploying web containers as OSGi bundles
            - 8.6.1.1. Tomcat 5.5.x, 6.0.x
            - 8.6.1.2. Jetty 6.1.8+ , 6.2.0
        - 8.6.2. Common libraries

- 8.7. 스프링 MVC 통합
- 9장. OSGi 기반 애플리케이션 테스트하기
    - 9.1. OSGi Mocks
    - 9.2. 통합 테스트
        - 9.2.1. 간단한 OSGi 통합 테스트 작성하기
        - 9.2.2. 테스트에 필요한 것들 설치하기
        - 9.2.3. Advanced testing framework topics
            - 9.2.3.1. 테스트 manifest 커스터마이징
            - 9.2.3.2. 테스트 번들 컨텐츠 커스터마이징
            - 9.2.3.3. MANIFEST.MF 생성 이해하기
        - 9.2.4. OSGi application context 만들기
        - 9.2.5. 사용할 OSGi 플랫폼 설정하기
        - 9.2.6. 테스트 의존성 대기하기
        - 9.2.7. 테스팅 프레임워크 성능

This page last changed on 7월 01, 2008 by keesun.

- 소개
- 목차

소개

옮긴이: 백기선
이메일: whiteship2000@gmail.com
블로그: http://whiteship.tistory.com

스프링 DM 레퍼런스를 편역한 문서입니다. 상업적인 용도가 아니라면, 수정, 배포는 자유롭습니다.

목차

- 1장. 왜 스프링 DM 인가?
- 2장. 필요한 것
- 3장. 새로 추가된 내용
  - 3.1. 1.1.X
    - 3.1.1. Web Support
      - 3.1.1.1. Spring-MVC 통합
    - 3.1.2. 클래스패스 Resource 추상화
    - 3.1.3. 변경이 편리한 Extender 설정
    - 3.1.4. 향상된 클래스 로딩
- 4장. 번들과 Application Context
  - 4.1. 스프링 DM Extender 번들
  - 4.2. Appication Context 생성
    - 4.2.1. 필수 서비스 의존성들
    - 4.2.2. Application Context 서비스 공개하기
  - 4.3. 번들 라이프사이클
  - 4.4. 리소스 추상화
  - 4.5. BundleContext에 접근하기
  - 4.6. Application Context 없애기
  - 4.7. Extender 번들 멈추기
- 5장. 스프링 기반 OSGi 애플리케이션 패키징기와 배포하기
  - 5.1. 번들 형식과 Manifest 헤더
  - 5.2. Extender 설정 옵션
    - 5.2.1. Listening to Extender events
  - 5.3. 필요한 스프링 프레임워크와 스프링 DM 번들들
  - 5.4. 스프링 XML 지원 기능
  - 5.5. 패지키 가져오기와 공개하기
  - 5.6. 외부 라이브러리를 사용할 때 고려해야 할 것
  - 5.7. 문제 진단하기
- 6장. 서비스 레지스트리
  - 6.1. 스프링 빈을 OSGi 서비스로 공개하기
    - 6.1.1. Controlling the set of advertised service interfaces for an exported service
      - 6.1.1.1. Detecting the advertised interfaces at runtime
    - 6.1.2. 공개할 서비스에 프로퍼티 설정하기
    - 6.1.3 depends-on 설정
    - 6.1.4 context-class-loader 속성
    - 6.1.5. ranking 속성
    - 6.1.6. service 엘리먼트 속성
    - 6.1.7. 서비스 등록과 해지 라이프사이클
      - 6.1.7.1. OsgiServiceRegistrationListener 인터페이스
  - 6.2. OSGi 서비스 레퍼런스 정의하기
    - 6.2.1. 단일 서비스 참조하기

                - 6.2.1.1. 가져온 서비스의 인터페이스 제어하기
                - 6.2.1.2. filter 속성
                - 6.2.1.3. bean-name 속성
                - 6.2.1.4. cardinality 속성
                - 6.2.1.5. depends-on 속성
                - 6.2.1.6. context-class-loader 속성
                - 6.2.1.7. reference 엘리먼트 속성
                - 6.2.1.8. reference와 OSGi 서비스 동적특성Dynamics
                - 6.2.1.9. 관리하고 있는 서비스 레퍼런스 참조하기
            - 6.2.2. 서비스 콜렉션 참조하기
                - 6.2.2.1. Greedy Proxing
                - 6.2.2.2. 콜렉션(list와 set) 엘리먼트 속성
                - 6.2.2.3. list, set 그리고 OSGi 서비스 동적특성
                - 6.2.2.4. Iterator 제약사항과 서비스 콜렉션
            - 6.2.3. 가져온import OSGi 서비스의 동적특성 다루기
            - 6.2.4. 리스너와 서비스 프록시들
            - 6.2.5. 호출하는 BundleContext에 접근하기
        - 6.3. Exporter, Importer listener 베스트프랙티스
            - 6.3.1. 리스너와 cyclic depencdency
        - 6.4. Service importer global defaults
        - 6.5. 서비스 Exporter와 서비스 Importer의 관계
- 7장. 번들 다루기
- 8장. 웹 지원기능
    - 8.1. 지원하는 웹 컨테이너
    - 8.2. Web 지원기능 사용법
    - 8.3. OSGi 내에서 WAR 클래스패스
        - 8.3.1. Static Recsources
        - 8.3.2. 서블릿
        - 8.3.3. JSP
            - 8.3.3.1. 태그 라이브러리들
    - 8.4. web extender 설정하기
        - 8.4.1. 웹 배포자 변경하기
    - 8.5. 표준 배포자 커스터마이징 하기
    - 8.6. OSGi에 배포할 수 있는 라이브러리와 웹 개발
        - 8.6.1. Deploying web containers as OSGi bundles
            - 8.6.1.1. Tomcat 5.5.x, 6.0.x
            - 8.6.1.2. Jetty 6.1.8+ , 6.2.0
        - 8.6.2. Common libraries
    - 8.7. 스프링 MVC 통합
- 9장. OSGi 기반 애플리케이션 테스트하기
    - 9.1. OSGi Mocks
    - 9.2. 통합 테스트
        - 9.2.1. 간단한 OSGi 통합 테스트 작성하기
        - 9.2.2. 테스트에 필요한 것들 설치하기
        - 9.2.3. Advanced testing framework topics
            - 9.2.3.1. 테스트 manifest 커스터마이징
            - 9.2.3.2. 테스트 번들 컨텐츠 커스터마이징
            - 9.2.3.3. MANIFEST.MF 생성 이해하기
        - 9.2.4. OSGi application context 만들기
        - 9.2.5. 사용할 OSGi 플랫폼 설정하기
        - 9.2.6. 테스트 의존성 대기하기
        - 9.2.7. 테스팅 프레임워크 성능

1장. 왜 스프링 DM 인가?

This page last changed on 7월 01, 2008 by keesun.

스프링 프레임워크는 full-stack Java/JEE를 이끌고 있는 애플리케이션 프레임워크다. 의존성 주입, AOP, 그리고 포터블 서비스 추상화 계층을 통해 가벼운 컨터이너를 제공하며 비침략적인 프로그래밍 모델을 가능케 한다. 스프링 DM은 스프링 애플리케이션을 OSGi 실행 환경(모듈을 실행시에 설치, 수정, 제거 할 수 있으며 모듈화와 버전관리 기능 지원이 뛰어나다.)에 쉽게 배포 할 수 있게 해준다.

엔터프라이즈 애플리케이션은 스프링 다이내믹 모듈과 OSGi 플랫폼을 사용하여 다음과 같은 장점을 얻을 수 있다.

- 보다 나은 모듈화. (모듈이라는 경계를 강화하여 애플리케이션 로직을 모듈 단위로 경계를 긋는다.)
- 모듈을 버전에 따라 동시에 여러개를 배포할 수 있다.
- 동적으로(런타임에) 시스템에 있는 다른 모듈이 제공하는 서비스를 사용하고 발견할 수 있다.
- 동적으로 모듈을 실행 중에 설치, 업데이트, 제거 할 수 있다.
- 스프링 프레임워크를 사용하여 여러 모듈에 걸쳐 컴포넌트들을 생성하고, 설정하고, 묶고assemble, 꾸밀decorate 수 있다.
- 개발자들이 쉽게 OSGi 플랫폼의 기능을 사용할 수 있도록 간단하고 익숙한 프로그래밍 모델을 제공한다.

2장. 필요한 것

This page last changed on 6월 27, 2008 by keesun.

스프링 다이내믹 모듈(DM) 1.0은 JDK 1.4 이상과 OSGi R4 이상을 지원한다. 스프링 DM을 사용해서 배포하는 번들은 반드시 MENIFEST.MF에 "Bundle-ManifestVersion: 2"라고 명시해야 한다.

> 기선's 코멘트
>
> 저게 무슨 뜻이냐면, OSGi R4를 따르는 번들이라는 뜻입니다. 스프링 DM이 OSGi R4 이상을 지원하니까 저렇게 설정해야겠죠. 2 말고 1이라고 할 수도 있는데 그건 OSGi R3 스펙을 따르는 번들이라는 뜻입니다.

스프링 DM은 Continuous Integration의 일부로 Equinox 3.2.2, Felix 1.0.3, Knopflerfish 2.0.4에서 테스트를 한다.

3장. 새로 추가된 내용

This page last changed on 7월 01, 2008 by keesun.

스프링 DM이 젊은 프로젝트(young project)이다 보니, 매 버전 마다 새로운 기능이 추가 된다. 이 번 챕터에서 새로 추가된 기능과 진화한 기능들을 매우 고도 압축하여 짧게 요약해둘 것이다. 자세한 내용은 그와 관련된 링크를 참조하기 바란다.

3.1. 1.1.X

This page last changed on 7월 01, 2008 by keesun.

3.1.1. Web Support

This page last changed on 7월 01, 2008 by keesun.

The biggest feature in Spring Dynamic Modules 1.1.x is the transparent support for web applications on OSGi platforms.

기선's 코멘트

멋진 단어 중에 하나인 Transparent 나왔습니다. non-invasive와 같은 맥락으로 이해하면 되겠죠. 투명하게 지원한다.. 뭐 이렇게 해석하시거나 이해하시면 곤란합니다.

톰캣이나 제티같은 웹 컨테이너를 직접 통합하여 WAR를 바로 배포할 수 있다. 자세한 내용은 8장. 웹 지원기능에서 자세히 다룬답니다.

3.1.1.1. Spring-MVC 통합

This page last changed on 7월 01, 2008 by keesun.

추가적으로, 1.1.x 부터 스프링 MVC 애플리케이션을 OSGi 환경 내에서 실행하는 것이 가능하다. 자세한 정보는 8.7. 스프링 MVC 통합을 참조하라.

3.1.2. 클래스패스 Resource 추상화

This page last changed on 7월 01, 2008 by keesun.

OSGi Resource를 찾을 때 classpath: 나 classpath*:를 사용할 수 있다. 이것은 마치 스프링의 component scanning처럼 OSGi 플랫폼에 올라와있는 여러 번들들에 있는 클래스패스 리소스들을 찾아준다. 자세한 내용은 4.4. 리소스 추상화를 참조하라.

3.1.3. 변경이 편리한 Extender 설정

This page last changed on 7월 01, 2008 by keesun.

1.1.X는 스프링 DM이 사용하는 여러 extender 기본 설정을 쉽게 변경할 수 있는 기능이 추가 되었다. fragment를 사용하여 사용자들이 웹 애플리케이션 컨텍스트를 시작하는 방법, 배포 또는 스프링 애플리케이션이 사용할 쓰레드 풀에 대한 설정을 변경할 수 있다. 추가적으로 OSGi 스프링 애플리케이션 컨텍스트 라이프사이클에 대응하는 이벤트를 받는 것도 가능하다. 4.1. 스프링 DM Extender 번들에서 모든 가용한 옵션과 그것들을 설명하고 있다.

기선's 코멘트

extender: OSGi 애플리케이션에서 쓰이는 패턴 이름이기도 하고 구현체(번들) 이름이기도 합니다. 스프링 DM에서는 기본적으로 스프링을 사용한 번들의 application context를 만들어주는 spring-extender.jar와 웹 application context를 만들어주는 spring-web-extender.jar 두 개의 구현체를 제공해 줍니다. 이들에 대해서는 뒤에 더 자세하게 나오니까 지금은 이정도로 넘어가시면 됩니다.

fragment: 단순 무식하게 말하자면 빌붙기라고 할까요. 자신의 클래스로더를 만들지 않으며, BundleAcvator도 안되고 자신이 호스트로 설정한 번들에 이미 있는 것들을 재정의 할 수도 없습니다. 확장 용도로 사용하는 겁니다. 예를 들어, 국제화 같은 경우 새로운 locale 파일을 fragement로 추가하면 호스트에서 이를 가져다가 사용할 수 있는 구조입니다. OSGi R4에 추가된 스펙입니다. 제세한 내용은 Appendix E.1 OSGi Fragment에 있습니다.

3.1.4. 향상된 클래스 로딩

This page last changed on 7월 01, 2008 by keesun.

1.1.x에서 프록시 생성이 좀 더 향상되서, 관리하고 있는 번들에 패키지를 묶어주는것이 좀 더 나아졌다. 자세한 내용은 FAQ 참조.

4장. 번들과 Application Context

This page last changed on 7월 01, 2008 by keesun.

OSGi 번들

OSGi에서 배포 단위는 번들.
번들은 런타임시에 세 가지 steady 상태 중 하나가 됨. installed, resolved, active.
번들은 서비스를 OSGi 서비스(객체) 레지스트리에 등록하여 다른 번들이 해당 서비스를 사용하도록 할 수 있다.
번들은 패키지를 노출 시켜서 다른 번들들이 노출 된 타입들을 import 해서 사용하도록 할 수 있다.

스프링 Application Context

스프링의 기본적인 모듈화 단위는 application context.
이 안에 스프링이 관리할 빈들을 담고 있다.
상속 구조를 가질 수 있다. 하위 context에서 상위 context에 있는 빈들을 참조할 수 있지만, 그 반대는 안 된다.
(Spring MVC를 잘 보면 이 구조가 있죠.)
스프링의 exporter나 factory bean들은 외부 클라이언트쪽 application context에서 해당 bean에 대한 레퍼런스를 사용할 수 있도록 제공해준다.
(Spring의 Expoter들은 JMS나 RPC 쪽을 보면 잘 나와 있습니다. 아주 일관된 패턴으로 Exporter들을 제공해 줍니다.)

OSGi 번들과 스프링 Application Context

잘 어울린다.
스프링 DM을 사용해서, 번들은 번들 내부에 있는 객체들(bean)을 생성하고, 설정하고, 묶고, 꾸며줄 applicaion context를 가질 수 있다.
그런 Bean들 중에 일부를 선택적으로 OSGi 서비스로 노출시켜서 외부의 다른 번들에서 사용할 수 있도록 할 수 있고, 그런 빈들은 깔끔하게Transparently 다른 OSGi 서비스에 주입 될 수 있다.

4.1. 스프링 DM Extender 번들

This page last changed on 7월 01, 2008 by keesun.

스프링 DM은 org.springframework.osgi.bundle.extender라는 번들을 제공한다. 이 번들은 번들에서 사용할 스프링 application context를 생성해준다. 스프링 웹 애플리케이션의 ContextLoaderListener와 같은 역할을 한다. extender 번들이 설치 installed 되고 시작 started 되면 이미 Activce인 상태의 번들 중에 스프링 DM을 사용하는 번들을 찾아서 application context를 만들어 준다. 또한, 번들 시작 이벤트를 주시 listening 하다가 스프링 DM을 사용한 번들이 시작 되면 application context를 만들어 준다.

5.1에서 extender가 어떻게 스프링 DM을 사용한 번들로 인식하는지 알아본다.

Extender Pattern

"allow other bundles to extend the functionality in a specific domain"
특정 도메인에 있는 기능을 다른 번들들이 확장할 수 있도록 하는 패턴.

4.2. Appication Context 생성

This page last changed on 7월 01, 2008 by keesun.

Extender 번들은 application context를 비동기적으로 생성한다.

- OSGi 서비스 플랫폼 시작 속도를 빠르게 해준다.
- 서비스들 간의 종속성으로 인한 데드락의 위험이 없다.
- 따라서 스프링 DM을 사용한 번들의 application context가 만들어지기 전에 STARTED 상태가 될 수 있다.
- 5.1을 참조하면, 동기적으로 번들을 생성하도록 설정하는 방법을 참조할 수 있다.
- 만약 application context를 무슨 이유에서건 생성하지 못했다면, 로그를 남기고 번들은 여전히 STARTED 상태다. 단, 해당 번들이 export한 서비스는 하나도 없는 상태로..

4.2.1. 필수 서비스 의존성들

This page last changed on 7월 01, 2008 by keesun.

OSGi 서비스의 가용성에 필수적인(mandatory) 의존성을 선언하면, 해당 application context의 생성은 OSGi 서비스 레지스트리를 통해서 필수적인 의존성을 만족 시킬 때까지 대기(block) 한다.

- 6장 서비스 레지스트리에서 필수적인 서비스 레퍼런스를 찾지 못할 때 어떤일이 벌어지는 지 설명한다.
- 이런 원리로 스프링 DM은 번들들이 어떤 순서대로 시작되는 상관없이 필수적인 종속성을 가지도록 보장한다.
- 타임아웃을 설정할 수 있다. 기본은 5분이다. timeout 지시자에 그 값을 설정할 수 있다. 5.1 참조.
- 해당 번들 실행시 필요한 서비스를 못찾으면 바로 application context 생성을 실패하도록 설정할 수 있다.(fail-fast) 이 경우 해당 번들은 ACTIVE 상태가 아니라 RESOLVED 상태로 남게 된다.

4.2.2. Application Context 서비스 공개하기

This page last changed on 7월 01, 2008 by keesun.

번들의 application context 생성이 완료되면, application context 객체는 자동으로 OSGi 서비스 레지스트리에 하나의 서비스로 등록이 된다.

- org.springframework.context.ApplicationContext 인터페이스 아래로 공개된다.
- 공개된 서비스들은 org.springframework.context.service.name 이라는 서비스 속성을 가지고 있다. 이 값은 해당 application context를 가지고 있는 번들의 Symbolic Name으로 설정된다.
- application context를 서비스로 등록하는 걸 막을 수도 있는데, 이건 5.1에서 살펴본다.

노트

Application context가 서비스로 등록되는 이유는 testing, administration, management를 하기 위해서다. context 객체에 런타임시 접근하여 getbean() 같은 것을 호출하는 행위를 하지 말기 바란다. 다른 번들의 application context에 등록되어 있는 Bean을 가져오고 싶을 때 권장하는 방법은 일단 OSGi 서비스로 해당 bean을 export하고, 그걸 참조하고 싶어하는 context에서 서비스로 import하는 것이다. 이렇게 서비스 레지스트리를 통해서 가져와야 그에 상응하는 버전에 해당하는 서비스만 참조하는 것을 보장하며, OSGi 플랫폼이 동적으로 관리하는 서비스를 가져올 수 있다.

4.3. 번들 라이프사이클

This page last changed on 7월 01, 2008 by keesun.

OSGi는 다이내믹 플랫폼으로, 프레임워크가 동작하고 있는 도중에 번들을 설치, 시작, 업데이트, 멈춤, 제거 할 수 있다.

번들이 멈추면be stopped

- 번들이 등록한 서비스들은 모두 등록이 해지되고unregistered 번들은 RESOLVED 상태가 된다.
- 번들이 가지고 있던 자원을 반납하고 쓰레드도 종료한다.
- 번들이 노출 시켰던 패키지들은 번들이 멈추더라도 계속해서 다른 번들들에 의해 사용될 수 있다.

번들은 RESOLVED 상태에서 업데이트 할 수 있다.

- 업데이트하는 과정은 같은 번들을 특정 버전에서 다른 버전으로 이관migrate하는 것이다.

번들은 RESOLVED 상태에서 시작be started 될 수 있다.

- 시작되면 번들은 ACTIVE 상태가 된다.

OSGi의 PackageAdmin refreshPackages 명령어

- 전체 OSGi 프레임워크 또는 설치되어 있는 번들들의 모든 패키지를 리프래시한다.
- 리프래시하는 동안에 그 대상이 되는 번들의 Application Context는 멈췄다가 재시작한다.
- refreshPackages 명령 처리 후, 수정된 번들의 이전 버전 패키지 또는 제거된 번들의 패키지는 더이상 사용할 수 없다. 자세한 사항은 OSGi 스펙 참조.

(다시) 번들이 멈추면..

- application context는 자동으로 제거된다.
- 서비스들도 OSGi 서비스 레지스트리에서 제거된다.
- application context의 종료 라이프사이클(DisposableBean, destroy-method, @Post머시기..)이 진행된다.
- 멈춘담에 바로 다시 시작시키면, 새로운 application context를 만든다.

4.4. 리소스 추상화

This page last changed on 7월 01, 2008 by keesun.

스프링은 리소스 추상화 계층을 제공한다.

- Application Context는 그 녀석을 기본으로 탑재하고 있다.
- 모든 리소스는 application context가 가지고 있는 org.springframework.core.io.ResourceLoader가 읽어온다. 물론 이 녀석을 별도의 bean에 주입하고 직접 코딩을 통해서 리소스를 읽어와도 된다.
- 리소스의 경로 앞에 classpath: 또는 file: 접두어prefix를 사용하여 가져올 리소스의 경로를 지정할 수 있는데, 사용하는 application context의 구현체에 따라 기본으로 사용하는 접두어가 다르다.

OSGi 4.0.X 스펙에는 리소스를 읽어올 세 종류의 공간(space)을 정의하고 있습니다.

- 스프링 DM은 이 모든 것들을 적잘한 OSGi 맞춤 application context와 적잘한 접두어를 사용하여 지원합니다.

접두어	OSGi 리소스 로딩 방법	설명
classpath:	Class Space	번들 클래스로더(the bundle, all imported packages and required bundles)를 찾는다. 해당 번들들이 RESOLVED 상태가 되도록 강제한다. 이 방법은 OSGi API Bundle 클래스의 getResource(String)과 유사한 방법이다.
classpath*:	Class Space	번들 클래스로더(the bundle and all imported packages and required bundles)를 찾는다. 해당 번들들이 RESOLVED 상태가 되도록 강제한다. 이 방법은 OSGi API Bundle 클래스의 getResource(String)과 유사한 방법이다.
osgibundlejar:	JAR File(or JarSpace)	번들 jar만 찾는다. 번들을 RESOLVED 상태로 만들 필요 없이 low-level 접근을 제공한다.
osgibundle:	Bundle Space	번들 jar와 그것에 붙어있는 조각(fragment)들까지 찾는다. 클래스로더를 생성하거나 번들이 RESOLVED 상태가 되도록 강제하지 않는다.

노트

만약 접두어를 붙이지 않으면, 기본으로 bundle space(osgibundle:)를 사용한다.

노트

OSGi의 동적인 특징 때문에, 번들 클래스패스가 생명주기 동안 변경 될 수 있다. 이것으로 인해 실행 중이 환경 또는 타겟 플랫폼에 기반해서 패턴 매칭을 할 때 전혀 다른 클래스패스 리소스가 반환될 가능성이 있다.

file: 이나 http: 같은 일반적인 스프링 리소스 접두어들도 물론 지원하며, 리소스 로딩이 되는 대상은 위치에 있더라고 상관없다. resource-loading 번들 또는 거기에 붙어있는 fragment에 국한되진 않는다.

OSGi 플랫폼들이 번들에 들어있는 내용물을 찾기 위한 자신들 만의 독특한 접두어들을 가지고 있을 수도 있다. 예를들어, Equinox는 bundleresource: 와 bundlentry: 접두어를 가지고 있다. 물론 이렇게 플랫폼이 정의한 프리픽스도 스프링 OSGi(여긴 DM이라고 안 했네.ㅋㅋ)과 함께 사용할 수 있다. (특정, OSGi 구현체에 여러분들이 직접 타이핑하는 수고는 해야하다.)

4.5. BundleContext에 접근하기

This page last changed on 7월 01, 2008 by keesun.

스프링 DM을 사용하면 OSGi API에 의존하지 않아도 된다.

- 정말로 OSGi의 BundleContext 객체에 접근하고 싶다면, 스프링이 도와줄 것이다.
- Extender가 만들어준 application context에는 BundleContext 타입의 bean 하나를 bundleContext라는 이름으로 항상 가지고 있다.
- 이 bean을 application context내의 다른 bean에 얼마든지 중입inject 할 수 있다.
- 거기다가, BundleContextAware라는 인터페이스까지 마련해 뒀다.
- 이 인터페이스를 구현한 bean들은 스프링이 bundleContext를 끼워넣어줄 것이다. 이 기능을 사용하려면, org.springframework.osgi.context 패키지를 import 해야한다.

```
public interface BundleContextAware {
  public void setBundleContext(BundleContext context);
}
```

4.6. Application Context 없애기

This page last changed on 7월 01, 2008 by keesun.

application context의 삶은 자신이 포함되어 있는 번들에 달려있다. 따라서 만약에 번들을 제거uninstall 하면, application context도 제거되고, export 했던 서비스들도 레지스트리에서 내리고, import 했던 서비스들도 제거한다.

번들만 별도로 닫히거나 전체 OSGi 플랫폼을 끄는 것과 같은 매우 큰 이벤트의 일부로 닫힐 수가 있다. 이런 경우이거나 extender 번들이 닫히는 경우에는, 서비스들 사이의 종속성에 근거하여 관리하에 닫게 된다. 자세한 내용은 다음 섹션에서...

4.7. Extender 번들 멈추기

This page last changed on 7월 01, 2008 by keesun.

Extender 번들이 멈추게stop 되면, Extender가 만든 모든 application context가 제거 될 것이다. Application context가 제거되는 순서는 다음과 같다.

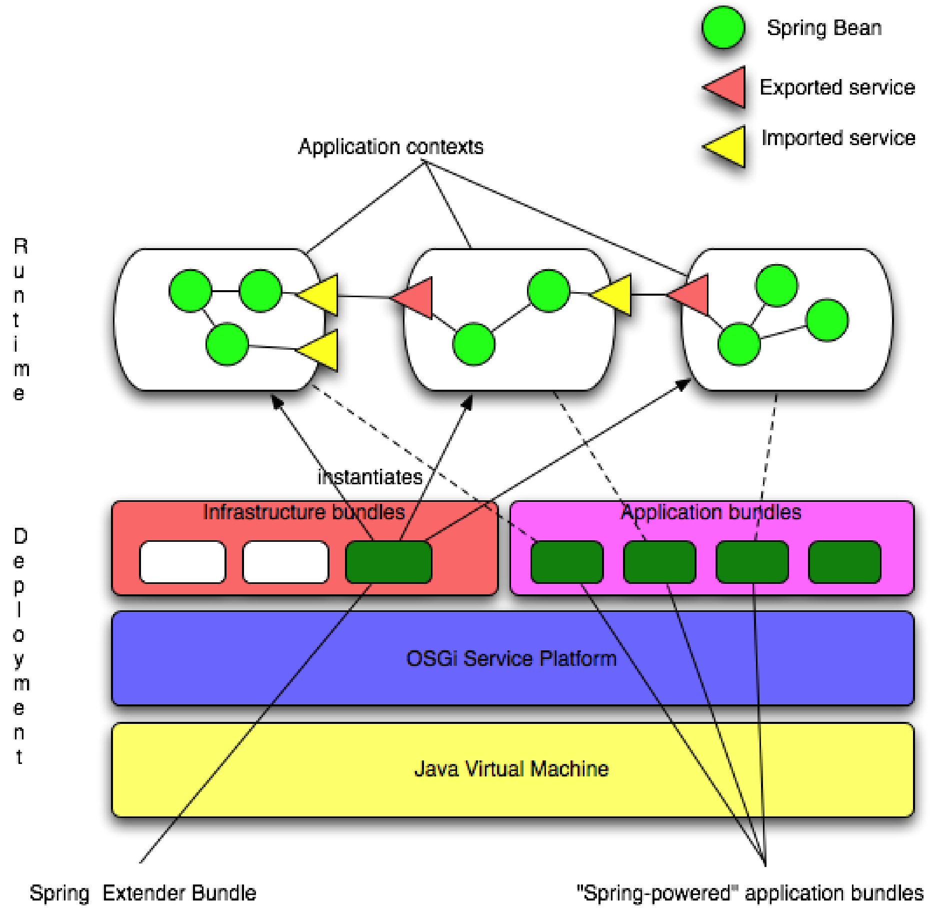
1. 어떤 서비스도 export하지 않았거나, export 했지만 다른 번들들이 참조하지 않고 있는 application context를 가장 먼저 없앤다. 없앨 때는 번들 id의 역순으로 가장 최근에 설치된 번들의 application context부터 없앤다.
2. 1번 과정에서 import한 서비스들에 대한 레퍼런스가 끊기면서 새롭게 다시 1번 과정이 대상이 되는 application context가 생길것이다. 그렇다면 1번을 다시 수행한다.
3. 더이상 동작 중인 application context가 없다면, 멈춘다. 만약 여전히 남아있는 application context가 있다면, 이건 상호 참조cyclic depedency of references가 있다는 것이다. 이 경우에는 서비스 랭킹을 확인해서 랭킹이 낮은 녀석을 내린다. 그리고 1번부터 다시 한다.

5장. 스프링 기반 OSGi 애플리케이션 패키징기와 배포하기

This page last changed on 7월 01, 2008 by keesun.

보통의 스프링 애플리케이션들은 단일 Application Context를 사용하거나, 서비스 계층, 데이터 계층 그리고 도메인 객체를 가지고 있는 부모 Context와 웹 계층 컴포넌트들을 가지고 있는 자식 Context로 나눠서 사용했을 것이다. Application Context는 여러 설정 파일들을 뭉쳐서 생성할 수 있었다.

애플리케이션을 OSGi에 배포할 때 자연스러운 방법은 애플리케이션을 OSGi 서비스 레지스트리를 사용하여 상호 작용하는 번들 집합체로 패키징 하는 것이다. 독립적인 하위 시스템은 독립적인 번들 또는 번들들의 집합체로 패키징해야 한다.(수직 파티셔닝) 하위 시스템은 단일 번들로 묶을 수도 있고, 레이어별로 번들을 구성할 수도 있다.(수평 파티셔닝) 예를 들어, 웹 애플리케이션은 간단하게 네 조각으로 나눌 수가 있다. 웹 번들, 서비스 계층 번들, 데이터 계층 번들, 도메인 모델 번들.



이 예제에서 데이터 계층 번들(분홍색 Application bundles의 첫 번째 녹색 네모)은 데이터 계층 Application Context에 여러 내부 컴포넌트(bean들)들을 가지고 있으며 그 들 중에서 두 개의 빈을 공개적으로 외부에서 접근 가능하도록 그들을 OSGi 서비스 레지스트리에 등록했다.

서 비스 계층 번들(Application bundles의 두 번째 녹색 네모)은 역시 여러 개의 내부 커포넌트를 가진 Application Context를 가지고 있고, 그 중 몇개가 데이터 계층 서비스들을 사용하려고 import 하고 있다. 서비스 계층 컴포넌트 중에 두 개를 외부에서 이용할 수 있도록 OSGi 서비스 레지스트리에 등록했다.(레지스트리 그림은 빠져있어서 2% 아쉽.. 아니면 서비스로 노출시킨 bean은 다른 색으로 그려주던지...)

웹 컴포넌트 번들(Application bundles의 세 번째 녹색 네모)은 Web application context에 여러 컴포넌트들을 가지고 있고 그 중 몇 개는 서비스를 OSGi 서비스 레지스트리로부터 참조해 온다.

도메인 모델 번들(Application bundles의 네 번째 녹색 네모)은 오직 도메인 모델 타입은 컴포넌트(걍 빈이라고 하지..)가 될 필요가 없기 때문에 이 번들에는 Application context가 없다.

5.1. 번들 형식과 Manifest 헤더

This page last changed on 7월 01, 2008 by keesun.

각각의 애플리케이션 모듈은 OSGi 번들로 패키징해야 한다. 번들은 META-INF/MANIFEST.MF 파일을 가지고 있는 jar 파일이다. OSGi 서비스 플랫폼 핵심 스팩 3.2에서 그 자세한 내용을 살펴볼 수 있다. 몇몇 OSGi 구현체들은 풀어헤쳐진 jar 파일을 지원하지만 형식은 동일하다.

스프링 Extender는 번들은 "스프링이 가미된"Spring-Powered 번들을 인식하고 해당 번들에 연관된 애플리케이션 컨텍스트를 번들이 시작할 때 다음의 조건 중에 하나라도 만족하면 만들어 준다.

- META-INF/spring 폴더에 하나 이상의 .xml 파일을 가지고 있을 경우.
- META-INF/MENIFEST.MF 파일에 Spring-Context라는 헤더가 있을 경우.

보 태자면, 만약 부가적으로 SpringExtender-Version 헤더를 manifest 파일에 추가하면, extender는 명시된 버전 조건을 충족시키는 번들만 인식할 것이다. 버전은 Bundle-Version에 명시되어 있다. SpringExtender-Version의 값은 OSGi 스펙을 따라야 한다.

Spring-Context 헤더가 빠져있는 경우 extender는 META-INF/spring 폴더 안에 있는 모든 파일을 유효한 스프링 설정 파일로 인식한다.

Application Context는 이런 파일들의 집합으로 구성된다. 권장하는 방법은 Applicaion Context 설정을 최소한 두 개의 파일로 나누는 것이다. 하나는 모듈이름-contet.xml 이고 다른 하나는 모듈이름-osgi-context.xml 로 말이다.

- 모듈이름-context.xml - OSGi와 상관없는 일반적인 빈 설정들을 담는다. 최상위 엘리먼트를 bean으로 사용.
- 모듈이름-osgi-context.xml - OSGi 서비스로 import/export 하는 빈들을 설정한다. 최상위 엘리먼트를 Spring OSGi 네임스페이스로 사용한다.

menifest 파일의 Spring-Context 헤더는 설정 파일 집합을 기술하기 위해 사용한다. 자원 경로는 상대 경로로 인식하고 여기에 하나라도 설정을 하면 META-INF/spring 폴더에 있는 파일들은 무시한다. 다음과 설정할 수 있다.

```
Spring-Context: config/account-data-context.xml, config/account-security-context.xml
```

가용한 설정 옵션

- create-asynchronosly(false|true): 애플리케이션 컨텍스트를 비동기적으로 생성(이게 기본값)할지 동기적으로 생성할지 설정.

```
Spring-Context: config/account-data-context.xml;create-asynchrously:=false
```

- wait-for-dependencies (true|false): 필수 서비스 의존성을 만족 할 때 까지 대기할지(이게 기본값) 말지 설정.

```
Spring-Context: config/osgi-*.xml;wait-for-dependencies:=false
```

- timeout (300): 대기 시간 설정 기본값은 5분. 300초. 초 단위로 설정. wait-for-dependencies가 false면 이 값은 무시함.

```
Spring-Context: *;timeout:=60
```

- publish-context (true|false): application context 객체 자체를 서비스 레지스트리에 등록할지(이게 기본값) 말지 설정.

```
Spring-Context: *;publish-context:=false
```

5.2. Extender 설정 옵션

This page last changed on 7월 01, 2008 by keesun.

번들과 관련된 설정이외에, 스프링 DM은 extender의 기본 행위를 설정할 수도 있다. 이것은 Spring-DM을 관리하는 환경Managed Environment에 포함시키거나 여러 번들에 걸친 기능이 필요할 때 유용하다. 확장 가능한 설정을 위해, extender는 OSGi의 fragment를 사용하여 기본 값을 재정의 할 수 있다. extender는 META-INF/spring 밑에 있는 모든 XML 파일을 읽어들이며 application context(OsgiBundleXmlApplicationContext)를 생성한다. extender의 기본 설정을 재정의하려면, 아래 테이블을 보고 원하는 빈을 읽어와서 정의를 하고 spring-osgi-extender.jar의 Fragment로 붙이면 된다.

```
Fragment-Host: org.springframework.bundle.osgi.extender
```

표생략

5.2.1. Listening to Extender events

This page last changed on 7월 01, 2008 by keesun.

번들의 application context 시작이나 실패시 로그를 남기고 싶을 수 있다. 이런 경우, 스프링 DM이 제공하는 org.springframework.osgi.context.event 패키지를 정의하여 OSGi application context가 보낼 수 있는 이벤틀를 정의할 수 있다.

현재 지원하는 이벤트는 딱 두개

- OsgiBundleContextRefreshedEvent
- OsgiBundleContextFailedEvent

이 이벤트들을 받길 원하는 쪽에서는 OsgiBundleApplicationContextListener를 구현해야 한다. 그리고 해당 서비스를 OSGi 서비스로 공개하야 한다. 그럼 스프링 DM이 알아서 리스너들을 찾아서 이벤트를 그쪽으로 보내줄 것이다.

이런식의 이벤트 처리는 OSGi의 장점을 모두 누릴 수 있다. 즉 extender가 이벤트 배포자나 이벤트 리스너로 부터 완전히 분리되어 있으며, 등록/해지 과정도 간단하다.

스프링 DM 이벤트와 스프링 이벤트와는 조금 차이가 있는데, 스프링 이벤트는 이벤트가 발생한 applicaion context 내부의 빈으로 이벤트를 보내지만, OSGi 이벤트는 자신을 지켜보고 있는 다른 application context의 빈으로 보내진다.

5.3. 필요한 스프링 프레임워크와 스프링 DM 번들들

This page last changed on 7월 01, 2008 by keesun.

스프링 Extender가 제대로 동작하려면 OSGi 플랫폼에 배포해야 할 몇몇 번들 아티팩트들이 있다.

- org.springframework.osgi.extender : extender 번들 자신
- org.springframework.osgi.core : 스프링 DM을 지원하는 핵심 구현체 번들
- org.springframework.osgi.io : 스프링 DM I/O 지원 라이브러리 번들

여기에 추가로 스프링 라이브러리가 필요한데, 스프링 2.5부터는 OSGi 플랫폼에 바로 배포가 가능한 형태로 배포하고 있다. 그 중에 다음을 필요로 한다.

- spring-core.jar
- spring-context.jar
- spring-beans.jar
- spring-aop.jar

추가로 다음의 라이브러리 번들을 필요로 한다. OSGi 플랫폼에 배포 가능한 형태의 JAR 파일들을 Spring DM With Dependencies 형태로 배포한 파일에 들어있다.

- aopalliance
- backport-util(JDK 1.4에 돌릴 때만 필요)
- cglib-nodep
- commons-logging API(SLF4J 강추)
- logging 구현체

5.4. 스프링 XML 지원 기능

This page last changed on 7월 01, 2008 by keesun.

스프링 2.0은 보다 쉬운 XML 설정과 확장 가능한 XML을 도입했다. 이 중 후자는 커스텀 스키마를 만들어 스프링 XML 인프라가 자동으로 해당 설정을 읽을 수 있는 것이 가능해졌다. 그냥 그것들을 클래스패스에 두기만 하면 된다.(이 부분은 토비 사부님이 1회 KSUG에서 발표하셨던 내용) 스프링 DM은 이 기본 지식을 활용해서 OSGi 환경에서 부가적인 코드나 menifest 선언 필요 없이 커스텀 스키마를 사용할 수 있도록 했다.

Spring DM은 OSGi 공간에 배포되는 모든 번들(스프링 DM 번들이든 아니든 상관없이)들을 조사하여 커스텀 스프링 네임스페이스 선언을 가지고 있는지 스캔한다.(META-INF/spring.handlers 와 META-INF/spring.schemas 번들 영역을 확인한다.) 만약에 해당하는 선언을 찾으면, 스프링 DM은 해당 스키마를 만들고 해당 네임스페이스는 OSGi 서비스를 통해서 자동으로 스프링 번들에 의해 사용이 가능해진다

=> 즉, 커스텀 스키마를 사용하는 번들을 배포할 때 필요한 건, 네임스페이스 파서와 스키마를 제공하는 라이브러리를 OSGi 플랫폼에 배포하기만 하면 된다는 것이다.

번들의 클래스 패스 내부에 있는 커스컴 스키마는 OSGi 공간에서 다른 번들들에 의해 사용될 수 있다. 하지만, 번들 내부 라이브러리의 네임스페이스는 다른 번들들에의해 공유되지 않는다. 다른 번들들이 볼 수 없다.

=> 번들로 배포된 네임스페이스만 공개 되고, 내장된 라이브러리의 네임스페이스는 공개하지 않음.

스 프링 DM을 사용하면, 커스컴 네임스페이스 기능을 어떠한 부가작업도 필요없이 투명하게 지원한다. 내장된 네임스페이스 제공자(Embedded namespace provider)들은 우선권을 가지지만 공유하지는 않는다. 이와 반대로 번들로 배포된 제공자(providers deployed as bundle)들은 다른 번들들에서 참조할 수 있다.

5.5. 패지키 가져오기와 공개하기

This page last changed on 7월 01, 2008 by keesun.

Import-Package와 Export-Package manifest 헤더에 대한 자세한 내용은 OSGi 스펙을 참조하도록.

번들은 자신이 의존성을 가지는 모든 외부 패키지들을 Import-Package를 사용해서 설정한다.

만약 다른 번들들이 사용할 필요가 있는 타입을 제공해야 한다면, Export-Package를 사용하여 외부 번들에서 사용할 수 있는 모든 패키지를 설정한다.

5.6. 외부 라이브러리를 사용할 때 고려해야 할 것

This page last changed on 7월 01, 2008 by keesun.

대부분의 엔터프라이즈 애플리케이션 라이브러리들은 context class loader를 통해서 타입과 리소스를 로딩할 수 있다고 가정한다. 보통 개발자가 이 것을 직접 다루지는 않지만, 애플리케이션 서버, 컨테이너 또는 멀티 쓰레드로 동작하는 애플리케이션들은 context class loader를 사용하고 있다.

OSGi R4에서는 context class loader를 통해 가용한 타입이나 리소스 집합에 대해 정의되어 있지 않다. 이것은 OSGi 플랫폼이 쓰레드 context class loader 값을 보장하지 못한다는 것이고 다시 말하자면, ccl을 관리하지 않는다는 것이다.

따라서 클래스 로딩을 하거나 동적으로 새로운 클래스를 생성하는 코드가 OSGi 환경에서는 제대로 동작하지 않을 수 있다.

스프링 DM은 해당 번들의 application context를 생성하는 도중에 번들의 클래스패스에서 가용한 모든 타입과 리소스들을 ccl을 통해 접근할 수 있도록 보장한다. 또한 스프링 DM은 외부 서비스를 호출할 때와 공개한 서비스에 대한 요청에 서비스를 할 때 CCL을 통해서 접근 가능한 클래스와 리소스를 설정할 수 있다. 방법은 5.5에서 설명했다.

OSGi R5에서는 제 3의 라이브러리에 의해 추가된 암묵적인 클래스 패스와 생성된 클래스를 지원하기 위한 스펙을 정하고 있다. 그전까지는 DynamicImport-Package manifest 헤더를 사용하거나, Equinox의 버디buddy 매커니즘을 사용할 수 있을 것이다.

5.7. 문제 진단하기

This page last changed on 7월 01, 2008 by keesun.

여러분이 사용하기로 선택한 OSGi 플랫폼 구현체는 현제 OSGi 환경 상태에 대한 정보를 잘 제공해 주어야 한다. 예를 들어, -console 아규먼트를 Equinox 시작시 추가하면 커맨드 라인 콘솔을 제공한다. 이것을 통해 어떤 번들들이 설치되어 있고 그들의 상태와, 해당 번들에 의해 공개된 패키지와 서비스, 번들 Resolve 실패 원인, 번들 라이프사이클 다루기 등을 할 수 있다.

게다가, 스프링 자체와 스프링 DM 번들은 문제의 원인을 조사할때 사용할 수 있는 확장 가능한 로깅 체계가 마련되어 있다. Simple Logging Facade for Java(slf4j) slf4j.jar 와 slf4j-log4j13.jar 번들을 설치하길 권장한다. 그렇게만 하면, 번들 클래스패스 루트에 log4j.properties 파일을 생성하여 사용할 수 있다.

스프링 DM 모듈은 commons-logging API를 내부적으로 사용하고 있는데, 이것은 로깅 구현체가 완전히 끼워맞출 수 있는pluggble 형태로 개발되었다는 것을 의미한다.

6장. 서비스 레지스트리

This page last changed on 7월 01, 2008 by keesun.

OSGi 서비스 레지스트리는 번들이 객체를 공유 레지스트리로 공개할 수 있고 이를 인터페이스를 통해 접근하도록 하는 기반 시설이다. 그렇게 공개된 서비스들은 레지스트리 내부에서 그들과 관련된 서비스 속성들을 가지고 있다.

스프링 DM은 osgi 네임스페이스를 제공하여 스프링이 빈을 OSGi 서비스로 공개할 수 있도록 한다. osgi 네임스페이스는 다른 최상위 네임스페이스 내부에 선언해도 되고, 최상위 네임스페이스로 사용해도 된다.

OSGi 관련 설정을 한 곳에 모아둔 osgi 네임스페이스를 최상위 네임스페이스로 사용하는 것이 편하다.(5.1 참조)

6.1. 스프링 빈을 OSGi 서비스로 공개하기

This page last changed on 7월 03, 2008 by keesun.

service 엘리먼트를 사용해서 OSGi 서비스로 공개할 빈을 설정할 수 있다. 최소한 공개시킬 빈과 서비스 인터페이스를 설정해야 한다.

```
<service ref="beanToPublish" interface="com.xyz.MessageService"/>
```

위 의 설정은 beanToPublish라는 빈을 com.xyz.MessageService 인터페이스를 통해서 사용할 수 있도록 서비스로 공개하겠다는 것이다. 그렇게 공개한 서비스는 org.springframework.osgi.bean.name 이라는 속성을 타겟 빈의 이름(여기서는 beanToPublish)을 설정한다.

서비스 엘리먼트로 정의한 빈은 org.osgi.framework.ServiceRegistration 타입의 빈이고 즉 서비스 레지스트리에 ServiceRegistration 객체가 등록된다. 이 빈에 id를 설정하고 다른 빈에서 해당 ServiceRegistration 객체를 참조 하도록 설정할 수 도 있다.

```
<service id="myServiceRegistration" ref="beanToPublish"
    interface="com.xyz.MessageService"/>
```

서비스로 공개할 빈 이름을 참조하는 대신에 서비스 빈 내부에 익명 내부 빈으로 등록할 수도 있다. 최상위 네임스페이스가 bean일 경우에 아래와 같을 것이다.

```
<osgi:service interface="com.xyz.MessageService">
  <bean class="SomeClass">
     ...
  </bean>
</osgi:service>
```

org.osgi.framework.ServiceFactory 인터페이스를 구현한 빈을 공개하면 OSGi Service Platform Core Specification 5.6에 있는 ServiceFactory 제약대로 동작한다.(해당 서비스 객체를 요구할 때마다 서비스팩토리에서 만들어서 줌) OSGi API를 구현하는 대신, 스프링 DM이 도입한 새로운 bean 스콥을 사용할 수도 있다. bundle 스콥으로 해당 빈을 이 스콥으로 OSGi 서비스로 공개하면 각각의 클라이언트(OSGi 서비스 레지스트리에서 서비스를 import하는 번들들)마다 독립적인 객체를 만들어서 제공해 줄 것이다. 해당 빈을 서비스로 임포트한 번들이 동작을 멈추면, 해당 빈 객체는 사라질 것이다. bundle 스콥을 사용하려면 다음과 같이 설정하면 된다.

```
<osgi:service ref="beanToBeExported" interface="com.xyz.MessageService"/>

<bean id="beanToBeExported" scope="bundle" class="com.xyz.MessageServiceImpl"/>
```

6.1.1. Controlling the set of advertised service interfaces for an exported service

This page last changed on 7월 03, 2008 by keesun.

OSGi 서비스 플랫폼 코어 스펙에는 "서비스 인터페이스"라는 용어를 정의하고 있는데, 이는 서비스의 public 메소드들 규약을 표현하는 용어다. 일반적으로 자바 인터페이스가 되지만, 스펙에서는 클래스 이름으로 서비스 객체를 등록 할 수도 있도록 지원한다. 따라서, "서비스 인터페이스"라는 말은 클래스와 인터페이스 둘 모두를 가리킬 수 있다.

공개할 서비스의 서비스 인터페이들을 기술하는데 몇가지 옵션들이 있다. 가장 간단하게는 interface 속성에 전체 패키지 경로가 붙은 인터페이스 이름을 설정할 수 있다. 해당 서비스를 여러 인터페이스에 등록하려면 interfaces라는 내부 엘리먼트를 사용한다.

```
<osgi:service ref="beanToBeExported">
  <osgi:interfaces>
     <value>com.xyz.MessageService</value>
     <value>com.xyz.MarkerInterface</value>
  </osgi:interfaces>
</osgi:service>
```

동시에 둘 다 사용하는 것은 문법에 어긋난다. 둘 중 하나만 사용하라.

6.1.1.1. Detecting the advertised interfaces at runtime

This page last changed on 7월 03, 2008 by keesun.

auto-export 속성을 사용하면 명시적으로 서비스 인터페이스들을 등록할 필요가 없다. 스프링 DM이 클래스 상속구조와 인터페이스를 분석하여 설정할 것이다.

auto-export 속성은 네 개중 하나의 값을 가질 수 있다.

- disabled: 기본값으로, 자동으로 찾지 않으므로, interface 또는 interfaces 설정을 해야 한다.
- interfaces: 해당 빈이 구현한 모든 인터페이스를 등록한다.
- class-hierarchy: 빈이 구현한 모든 타입과 슈퍼 타입을 등록한다.
- all-classes: 위에 두개 합친거

auto-export와 interfaces 옵션은 상호배타적이지 않다. 두 설정을 동시에 사용할 수 있다. 하지만, 대부분의 경우 다음의 설정이 유용할 것이다.

```
<service ref="beanToBeExported" auto-export="interfaces"/>
```

이렇게 설정하여 인터페이스 상속구조에서 모든 인터페이스 타입으로 서비스를 참조할 수 있다.

```
public interface SuperInterface {}

public interface SubInterface extends SuperInterface {}
```

이 런 코드에서 SupoerInterface로 등록한 서비스는 SubInterface로 참조할 수가 없는데, 이런 이유로 인해 해당 서비스에 auto-export="interfaces"로 설정하여 모든 인터페이스를 지원하도록하는 것이 베스트 프랙티스다.

6.1.2. 공개할 서비스에 프로퍼티 설정하기

This page last changed on 7월 03, 2008 by keesun.

앞서 언급했듯이, 공개한 서비스는 항상 org.springframework.osgi.bean.name 서비스 속성을 공개할 빈의 이름으로 설정되어 있다. service-properties 내부 엘리먼트를 사용하여 부가적인 속성을 설정할 수 있다. service-properties 엘리먼트는 키-값 쌍을 가지고 있는데 이는 서비스에서 속성을 사용할 수 있다. 키는 반드시 문자열이어야 하고 값은 OSGi 필터가 인식할 수 있는 타입이어야 한다. OSGi 서비스 플랫폼 코어 스팩 5.5를 보면 필터 표현식과 프로퍼티 값이 어떻게 대응하는지 참조할 수 있다.

service-properties 엘리먼트는 반드시 최소한 하나의 entry 엘리먼트를 가지고 있어야 한다.

```
<service ref="beanToBeExported" interface="com.xyz.MyServiceInterface">
  <service-properties>
    <beans:entry key="myOtherKey" value="aStringValue"/>
    <beans:entry key="aThirdKey" value-ref="beanToExposeAsProperty"/>
  </service-properties>
</service>
```

스프링 DM 로드맵에는 OSGi Configuration Administration 서비스에 안에 등록되어 있는 속성들을 등록된 서비스의 속성으로 공개하는 기능을 포함하고 있다. Appendix F. 로드맵에 자세한 내용이 있다.

6.1.3 depends-on 설정

This page last changed on 7월 03, 2008 by keesun.

스프링은 명시적으로 서비스 요소들간의 의존성을 관리한다. 예를 들어 서비스로 공개한 빈이 공개하기 전에 완전히 설정을 마치고 만든 다음에 공개한다. 만약에 서비스가 다른 컴포넌트(다른 서비스를 포함한)에 의존성을 가지고 있다면 반드시 해당 서비스를 공개하기 전에 그것들을 초기화 해야 하는데, 그럴 때 depends-on 속성을 사용여 그들의 의존성을 표현할 수 있다.

```
<service ref="beanToBeExported" interface="com.xyz.MyServiceInterface"
     depends-on="myOtherComponent"/>
```

6.1.4 context-class-loader 속성

This page last changed on 7월 03, 2008 by keesun.

OSGi 서비스 플랫폼 코어 스펙(현제 4.1이 작성 중이다.)에는 서비스 레지스트리에서 받아온 서비스에 어떤 요청을 했을 때 context class loader를 통해서 가용한 리소스나 타입을 명시하고 있지 않다. 따라서, 몇몇 서비스들은 context class loader에 특정 가정을 한 상태에서 라이브러리를 사용하고 있다. 스프링 DM은 서비스 실행 중에 context class loader를 명시적으로 제어할 수 있는 방법을 제공한다. context-class-loader 속성을 통해 제어할 수 있다.

context-class-loader 속성에는 unmanaged(기본값)와 service-provider를 사용할 수 있다. service-provider 값을 사용하여 스프링 DM으로 하여금 context class loader가 번들이 공개한 서비스의 클래스패스에 있는 모든 리소스를 참조할 수 있다는 것을 보장해준다.

context-class-loader를 service-provider로 설정하면, 클래스로더를 다루기 위해 서비스 객체가 프록시로 된다. 이 때 만약 서비스가 구현 클래스 일 경우 CGLIB을 필요로 한다.

6.1.5. ranking 속성

This page last changed on 7월 03, 2008 by keesun.

서비스 레지스트들에 여러 개의 같은 종류의 서비스가 존재할 경우, 우선 순위가 높은 서비스를 사용하게 된다.(OSGi 서비스 플랫폼 스펙 5.2.5) 이 값을 ranking 속성으로 설정할 수 있다. 기본값은 0

```
<service ref="beanToBeExported" interface="com.xyz.MyServiceInterface"
  ranking="9"/>
```

6.1.6. service 엘리먼트 속성

This page last changed on 7월 03, 2008 by keesun.

요약할 겸, 위에서 살펴본 모든 속성들과 그 값들을 표로 나타내면 다음과 같다.

표 생략

6.1.7. 서비스 등록과 해지 라이프사이클

This page last changed on 7월 03, 2008 by keesun.

service 엘리먼트로 등록된 서비스느는 애플리케이션 컨텍스트가 처음 만들어 질때 OSGI 서비스 레지스크리에 등록된다. 번들이 멈추면 서비스가 자동적으로 해지되고 애플리케이션 컨텍스트가 소멸한다.

만약 의존성이 충족되지 않아서 등록시 또는 해지시에 어떤 행위를 해야하는 상황이 필요할 수 있다. 그럴 때는 registration-listener 엘리먼트를 사용해서 리스너 빈을 정의할 수 있다.

```
<service ref="beanToBeExported" interface="SomeInterface">
  <registration-listener ref="myListener"                                  (1)
    registration-method="serviceRegistered"                                (2)
    unregistration-method="serviceUnregistered"/>                          (2)
  <registration-listener
     registration-method="register">                                       (3)
     <bean class="SomeListenerClass"/>                                     (4)
  </registration-listener>
</service>
```

(1) 최상위 빈 정의를 참조하는 리스너 선언
(2) 등록과 해지 메소드 설정
(3) 해당 리스너의 등록 메소드만 설정
(4) 내부 리스너 빈 등록

registration- method와 unregistration-method 속성은 리스너 빈에 정의한 메소드 이름을 나타내고 이들은 등록 및 해지시에 자동으로 호출된다. 등록과 해지는 콜백 메소드로 반드시 다음의 시그너처 중 하나를 따라야 한다.

```
public void anyMethodName(ServiceType serviceInstance, Map serviceProperties);

public void anyMethodName(ServiceType serviceInstance, Dictionary serviceProperties);
```

serviceType은 공개할 서비스의 인터페이스에 상응하는 어떤 타입이든지 될 수 있다.

등록 콜백은 서비스가 초기에 시작시 등록될 때 호출되고, 계속해서 다시 등록될 때마다 호출된다. 해지 콜백은 이유에 상관없이 서비스 해지 과정 중에 호출된다.

스프링 DM은 ServiceType 인자 타입을 보고 그에 상응하는 서비스가 등록/해지 될 때에만 등록/해지 메소드를 호출한다.

serviceProperties는 등록/해지 서비스의 모든 속성을 담고 있는 맵을 나타낸다. OSGi 스펙에 호환하기 위해 이 인자는 필요시에 java.util.Dictionary로 캐스팅될 수도 있다.

6.1.7.1. OsgiServiceRegistrationListener 인터페이스

This page last changed on 7월 03, 2008 by keesun.

위에서 설명한 방법말고 OsgiServiceRegistrationListener 인터페이스를 직접 구현하는 방법도 있는데, 이렇게 하면 XML 설정은 줄어드는 대신 스프링 DM에 종속적인 코드가 생긴다.

둘 다 사용할 수 있는데, 그럴 때는 먼저 OsgiServiceRegistrationListener 인터페이스의 메소드가 먼저 호출되고, 그 다음 커스텀 메소드가 호출된다.

6.2. OSGi 서비스 레퍼런스 정의하기

This page last changed on 7월 03, 2008 by keesun.

스프링 DM은 OSGi 서비스 레지스트리를 통해 사용할 수 있는 서비스를 나타내는 빈을 선언하는 기능을 제공한다. 이런 방법으로 통해 OSGi 서비스는 애플리케이션 컴포넌트로 주입될 수 있다. 서비스를 룩업할때는 해당 서비스가 지원하는 서비스 인터페이스 타입과 레지스트리에 등록된 서비스 속성에 부합하는 부가적인 필터 표현식을 사용한다.

몇몇 경우에, 간단하게 애플리케이션에서 하나의 서비스만을 필요로 할 때가 있다. 이 때는 reference 엘리먼트를 정의해서 단일 서비스를 참조하도록 정의할 수 있다. 다른 경우, 특히 OSGi 화이트보드 패턴을 사용할 때, 모든 가용한 서비들을 필요로 할 때가 있는데, 스프링 DM은 이거슬 List나 Set 콜렉션으로 이들 집합을 관리할 수 있는 기능을 제공한다.

6.2.1. 단일 서비스 참조하기

This page last changed on 7월 03, 2008 by keesun.

reference 엘리먼트는 서비스 레지스트리에 등록된 서비스를 참조할 때 사용한다.

선 언한 것에 부합하는 서비스가 여러 개일 수 있기 때문에, BundleContext.getServiceReference에 의해 반환되는 서비스를 참조하게 된다. 이게 무슨 뜻이냐면 가장 높은 랭킹의 서비스가 반환된다는 것이다. 또는 만약 랭킹이 같을 때는 서비스 id가 가장 낮은 순서(프레임워크에 먼저 등록될 수록 id가 낮다.)로 반환된다.(OSGi 스펙에 보면 보다 자세하게 서비스 선택 알고리즘이 설명되어 있다.)

6.2.1.1. 가져온 서비스의 인터페이스 제어하기

This page last changed on 7월 03, 2008 by keesun.

interface 속성은 해당 서비스가 반드시 구현한 인터페이스를 나타낸다. 예를 들어, 다음의 선언은 messageService 빈 하나를 등록한건데, 이것은 MessageService 인터페이스를 제공하는 서비스를 서비스 레지스트리로부터 질의하여 반환 받은 서비스가 된다.

```
<reference id="messageService" interface="com.xyz.MessageService"/>
```

service 선언과 마찬가지로, 여러개의 인터페이스를 기술할 때는 interface 속성 대신에 내부 엘리먼트로 interfaces를 사용하면 된다.

```
<osgi:reference id="importedOsgiService">
  <osgi:interfaces>
     <value>com.xyz.MessageService</value>
     <value>com.xyz.MarkerInterface</value>
  </osgi:interfaces>
</osgi:reference>
```

interface 속성과 interfaces 엘리먼트를 둘 다 사용하는 것은 문법에 어긋나며, 항상 둘 중 하나만 사용하도록 한다.

reference 엘리먼트로 정의한 빈은 번들이 참조할 수 있는 서비스의 인터페이스들을 구현하고 있다.(greedy proxing 이라고 한다.) 만약 등록된 서비스 인터페이스가 자바 클래스 타입을 포함하고 있다면(인터페이스가 아니라), 해당 타입들은 스프링 AOP를 따르게 되ㅇㄴ다. 간략하게 말하자면, 명시된 인터페이스들이 인터페이스가 아니라 클래스라면, cglib이 반드시 가용한 상태여야 하며, final 메소드가 없어야 한다.

6.2.1.2. filter 속성

This page last changed on 7월 03, 2008 by keesun.

부가적인 속성 filter는 OSGi 필터 표현식을 명시하고 서비스 레지스트리 룩업을 할 때 오직 주어진 필터에 해당하는 것들에서만 찾도록 제약을 가할 수 있다.

```
<reference id="asyncMessageService" interface="com.xyz.MessageService"
  filter="(asynchronous-delivery=true)"/>
```

이것은 asynchronous-delivery 속성이 true이고 MessageService 인터페이스로 등록해둔 OSGi 서비스만 참조할 것이다.

6.2.1.3. bean-name 속성

This page last changed on 7월 03, 2008 by keesun.

bean-name 속성은 service 엘리먼트를 사용하여 공개한 빈의 이름에 해당하는 서비스를 찾아주는 필터 표현식의 단축키 정도에 해당한다.

```
<bean id="(1)messageServiceBean" scope="bundle" class="com.xyz.MessageServiceImpl"/>
<!-- service exporter -->
<osgi:service id="messageServiceExporter" ref="(1)beanToBeExported"
 interface="com.xyz.MessageService"/>

<osgi:reference id="messageService" interface="com.xyz.MessageService"
   bean-name="(1)messageServiceBean"/>
```

(1) bean의 name속성에 사용한 이름

위 에 선언한 reference 빈은 MessageService 인터페이스로 등록된 서비스 중에서 org.springframework.osgi.bean.name 속성에 이름이 messageServiceBean 인 OSGi 서비스를 찾는다. 다시 간략하게 말하자면, 모든 공개된 빈 중에서 MessageService 인터페이스를 구현했고, 빈의 이름이 messageService인 빈을 참조하게 된다.

6.2.1.4. cardinality 속성

This page last changed on 7월 03, 2008 by keesun.

cardinality 속성은 항상 해당하는 서비스가 존재해야 하는지 아닌지를 기술할 때 사용한다. cardinality 값이 1..1(이게 기본값) 이면 해당하는 서비스가 반드시 가용해야 한다는 것을 뜻한다. cardinality 값이 0..1이면, 해당하는 서비스가 항상 필요한 것은 아니라는 것을 뜻한다.(4.2.1.6에 자세히 나와있다.) reference에 cardinality 1..1로 설정되어 있는 것은 필수mandatory 서비스 레퍼런스라고도 하며, 기본적으로 해당 레퍼런스가 참조 가능할 때까지 애플리케이션 컨텍스트 생성이 지연된다.

노트

해당 번들 자신이 공개한 서비스를 필수 서비스 레퍼런스로 참조하는 것은 에러다. 이런 행위로 인해 애플리케이션 컨텍스트 생성이 데드락이나 타임아웃으로 실패하게 될 것이다.

6.2.1.5. depends-on 속성

This page last changed on 7월 03, 2008 by keesun.

dependes-on 속성은 서비스 레지스트리에서 해당 서비스 레퍼런스를 룩업하기 전에 이 속성에 명시한 빈을 먼저 생성하라는 것이다.

6.2.1.6. context-class-loader 속성

This page last changed on 7월 03, 2008 by keesun.

OSGi 서비스 플랫폼 코어 스펙(현재 4.1 작성중)에는 서비스 레지스트리에서 얻어온 서비스를 통하여 컨텍스트 클래스 로더에서 가용한 타입이나 리소스들을 명시하고 있지 않다. 따라서 어떤 서비스들은 컨텍스트 클래스 로더로부터 특정 라이브러리가 가용하리라고 예상하고 작성하기도 한다. 스프링 DM은 서비스 호출 시점에 컨텍스트 클래스 로더 제어대해 명시적으로 제어한다. reference 엘리먼트의 context-class-loader 속성으로 이를 달성할 수 있다.

context-class-loader에 가용한 값은 다음과 같다.

- client - 서비스 호출 기간중에, 컨텍스트 클래스 로더는 서비스를 호출하고 있는 번들의 클래스패스에 있는 타입들을 볼 수 있다. 기본값이다.
- service-provider - 서비스 호출 기간중에, 컨텍스트 클래스 로더는 서비스를 공개한 쪽 번들의 클래스패스에 있는 타입들을 볼 수 있다.
- unmanaged - 서비스 호출 기간중에 컨텍스트 클래스 로더 관리를 하지 않음

6.2.1.7. reference 엘리먼트 속성

This page last changed on 7월 03, 2008 by keesun.

지금까지 살펴본 속성들 표 생략.

6.2.1.8. reference와 OSGi 서비스 동적특성Dynamics

This page last changed on 7월 03, 2008 by keesun.

reference 엘리먼트로 정의한 빈은 애플리케이션 컨텍스트의 생명주기 동안 바뀌지 않는다. 하지만, 해당 빈이 참조하는 OSGi 서비스는 언제든지 서비스 레지스트리에서 나가고 등록될 수 있다. 필수 서비스 레퍼런스(cardinality 1..1)인 경우에는, 해당 서비스가 가용할 때 까지 애플리케이션 컨텍스트 생성을 일정 시간 지연시킨다. 부가 서비스 레퍼런스(cardinality 0..1)인 경우에는 현재 해당 서비스가 가용한지 여부와 관계없이 해당 빈이 바로 생성된다.

만약 참조해야 하는 서비스가 없어지면, 스프링 DM은 해당 reference에 정의한 것에 해당하는 다른 서비스로 기존 서비스를 교체 시도를 한다. 애플리케이션은 리스너를 등록하여 자신이 참조하는 서비스의 변경을 알아차려야 한다. 만약 해당하는 서비스가 없다면, reference는 unsatisfied가 되고, 이 reference에 의존하는 다른 공개된 서비스들을 해당 reference의 의존성이 해결될때까지 서비스 레지스트리에서 해지한다. 자세한건 6.5에서..

만약 unsatistied 상태인 reference 빈에 어떤 요청이 발생했다면, 해당 요청은 레퍼런스가 다시 satisfied 상태가 될때까지 잠시 대기시킨다. timeout 속성에 이 때 대기할 시간(millisesecond)을 설정할 수 있다. 몇시한 타임아웃 값이 지나고 나서도 해당 빈이 가용하지 않다면 ServiceUnavailableException을 던진다.

6.2.1.9. 관리하고 있는 서비스 레퍼런스 참조하기

This page last changed on 7월 03, 2008 by keesun.

스프링 DM은 자신이 관리하고 있는 서비스 레퍼런스를 ServiceReference 타입으로 자동변환해준다. 따라서, ServiceReference 타입의 속성을 가지고 있는 빈에 해당 서비스를 (명시된 인터페이스 타입이 아니라 ServiceReference 타입으로)injection 할 수 있다.

```
public class BeanWithServiceReference {
    private ServiceReference serviceReference;
    private SomeService service;

    // getters/setters ommitted
}
```

```
<reference id="service" interface="com.xyz.SomeService"/>

<bean id="someBean" class="BeanWithServiceReference">
  <property name="serviceReference" ref="service"/>                                    (1)
  <property name="service" ref="service"/>                                             (2)
</bean>
```

1 Automatic managed service to ServiceReference conversion.
2 Managed service is injected without any conversion

노트

주입된 ServiceReference 타입의 빈은 참조하고 있는 OSGi 서비스 객체가 바뀔 때마다 같이 바뀐다.

6.2.2. 서비스 콜렉션 참조하기

This page last changed on 7월 03, 2008 by keesun.

때때로 애플리케이션은 간단하게 특정 범위 중에서 하나의 서비스를 참조할 뿐만 아니라, 범위 해당하는 모든 서비스를 참조해야 할 때도 있다. 스프링 DM은 이를 List 또는 Set(정렬은 부가적으로) 엘리먼트로 지원한다.

List 와 Set의 차이는 오직 동일성뿐이다. 레지스트리에 등록되어 있는 두 개 이상의 서비스들(각자 다른 서비스 id를 가지고 있을 것이다.)이 해당 서비스의 equals 메소드 구현에 따라 같을 수도 있는데. Set에는 그들 중에 하나만 가지고 있을 것이고, List는 전부다 반환해 준다.

set과 list 스키마 엘리먼트는 서비스 집합체를 나타낼 때 사용한다.

이 엘리먼트들도 interface, filter, bean-name, cardinality 그리고 context-class-loader를 지원한다. 단 cardinality의 값으로는 0..n 과 1..n 만 사용할 수 있다.

0..n은 해당 콜렉션에 대응하는 서비스가 하나도 없어도 상관없다는 것이고 1..n은 필수 레퍼런스를 나타낸다.

list는 java.util.List 타입으로 정의되고, set 엘리먼트는 java.util.Set 타입의 빈으로 정의된다.

다음의 예제는 EventListener 인터페이스를 구현한 모든 등록된 서비스들을 List 타입으로 반환할 것이다.

```
<list id="myEventListeners" interface="com.xyz.EventListener"/>
```

리스트의 요소들은 스프링에의해 동적으로 관리될 것이다. 대응하는 서비스가 레지스트리에 등록되고 해지됨에 따라 콜렉션의 구성요소도 갱신될 것이다.

스프링 DM은 정렬된 콜렉션도 지원한다. set과 list 둘다.

정렬하는 방법은 두 가지인데, 하나는 Comparator를 사용하는 방법(custom ordering)이고 다른 하나는 Comparable 인터페이스를 구현한 것 끼리 비교하는 것(natural ordering)이다.

6.2.2.1. Greedy Proxing

This page last changed on 7월 03, 2008 by keesun.

스프링 DM 서비스 콜렉션으로 가져온 모든 OSGi 서비스는 interface 속성에 설정한 클래스에 타입 호환가능하다. 하지만, 때때로 서비스가 부가적으로 설정한 클래스들을 사용하고 시을 수 있다.

그 런 경우, 스프링 DM 콜렉션은 greedy-groxing 속성을 지원하는데, 이 속성을 사용하여 참조하는 서비스에 부가적으로 설정한 모든 클래스들을 사용할 수 있도록 프록시들을 생성하게 할 수 있다. 따라서, 가져온 프록시를 interface에 설정한 클래스가 아닌 다른 타입으로 캐스팅을 할 수 있다. 다음의 예제를 보자.

```
<list id="services" interface="com.xyz.SomeService" greedy-proxying="true"/>
```

다음과 같이 할 수 있다.

```
for (Iterator iterator = services.iterator(); iterator.hasNext();) {
    SomeService service = (SomeService) iterator.next();
    service.executeOperation();
    // if the service implements an additional type
    // do something extra
    if (service instanceof MessageDispatcher) {
        ((MessageDispatcher)service).sendAckMessage();
    }
}
```

노트

greedy proxy와 instanceof를 사용하기 전에 다른 인터페이스나 클래스 사용을 고려해 보아라. 보다 나은 다형성과 객체 지향 스러운 서비스를 구성할 수 있을 것이다.

6.2.2.2. 콜렉션(list와 set) 엘리먼트 속성

This page last changed on 7월 03, 2008 by keesun.

timeout 이 없는거만 빼면 reference 랑 똑같다.

6.2.2.3. list, set 그리고 OSGi 서비스 동적특성

This page last changed on 7월 03, 2008 by keesun.

OSGi 서비스들의 집합은 OSGi 공간의 상태를 반영해야 하기 때문에 애플리케이션 컨텍스트의 라이프사이클 동안 계속해서 변한다. 서비스가 등록되고 해지되는 동안 콜렉션에 해당 서비스들이 추가 또는 삭제 된다.

reference 선언은 자신이 참조하는 서비스가 해지 되면 대체제를 찾는 반면, 콜렉션은 그냥 서비스를 콜렉션에서 제거한다. reference와 마찬가지로, 1..n 인 cardinality는 필수 레퍼런스라고 하며, 0..n은 부가적인 레퍼런스라고 한다. 만약에 대응하는 서비스가 없다면 필수 콜렉션만 unsatisfied 상태가 되고 ServiceUnavailableException을 던진다.

6.2.2.4. Iterator 제약사항과 서비스 콜렉션

This page last changed on 7월 03, 2008 by keesun.

콜렉션을 순회하는 방법중에 권장하는 방법은 Iterator를 사용하는 것이다. 하지만, OSGi 서비스는 언제든지 등록되고 해지될 수 있으므로, 콜렉션 또한 그에 따라 변경되어야 한다. 스프링 DM은 깔끔하게Transparenlty 사용자가 참조하는 모든 Iterator들을 수정해 준다. 따라서 사용자는 콜렉션이 변경되는 와중에도 안전하게 순회할 수 있다. 게다가, 그 Iterator들은 콜렉션의 모든 변경 사항을 반영한다. 변경이 Iterator 객체를 만든 후에 발생했더라도 반영된다. 만약 순회를 시작하자 마자 서비스들이 왕창 내려갔다고 했을때, 이 상태에서 순회를 계속하면 "죽은" 서비스들을 호출하게 될 것이다. 스프링 DM은 그래서 이 Iterator들이 스냅샷이 아니라 가장 최신 콜렉션 상태를 반영한 것이 된다. 순회가 얼마나 빠르고 느리냐는 상관없다.

서비스 수정은 Iteraor에서 순회하기 전에 있는 것들에만 반영이 된다는 것에 주의하자. 이미 순회를 마친 서비스에는 어쩔 도리가 없다. 만약 이미 해지된 서비스에 어떤 동작을 요구한다면 ServiceUnavailableException이 발생한다.

정리하자면, reference 선언은 자신이 참조하는 서비스가 해지되면 대체제를 찾지만 콜렉션은 대체제를 찾지 않는다. 그냥 자신의 콜렉션에서 해당 서비스를 제거할 뿐이다. 다음 순회할 때 그들을 사용하지 않도록.

Iterator 제약사항은 next() 메소드가 항상 hasNext() 호출 결과를 따른다는 것을 알아두자.

hasNext()가 true일 때 next()를 하면 항상 null이 아닌 값을 반환한다.
hasNext()가 false일 때 next()를 하면 NoSuchElementException을 던진다.

간단하게 Iterator를 리프레쉬하려면 hasNext()를 다시 호출하면 된다. 그럼 Iterator가 현재 콜렉션 에서 다음 순회 엔트리를 확인할 것이다.

6.2.3. 가져온import OSGi 서비스의 동적특성 다루기

This page last changed on 7월 03, 2008 by keesun.

reference나 set, list를 사용할 때 스프링 DM이 뒷단의 서비스를 관리할 것이다. 하지만, 하지만 때론 애플리케이션이 뒷단의 서비스 변경을 알고 싶을 수도 있다.

언 제 reference 빈이 묶이고 풀리는지 알고 싶은 그런 애플리케이션들은 내부 엘리먼트로 하나 이상의 listener 엘리먼트를 등록할 수 있다. 이 엘리먼트는 reference, set, list에서 사용할 수 있다. 서비스 공개 리스너 설정과 비슷하다. listener 엘리먼트에 org.springframework.osgi.service.importer.OsgiServiceLifecycleListener 인터페이스를 구현한 빈을 참조하도록 설정하면, 해당 인터페이스의 bind와 unbind 메소드를 호출하게 된다. 커스텀 바인드 언바인드 메소드를 사용하려면 메소드 이름을 사용하면 된다.

```
<reference id="someService" interface="com.xyz.MessageService">
  <listener ref="aListenerBean"/>
</reference>

<reference id="someService" interface="com.xyz.MessageService">
  <listener bind-method="onBind" unbind-method="onUnbind">
     <beans:bean class="MyCustomListener"/>
  </listener>
</reference>
```

만약 OsgiServiceLifecycleListener와 커스텀 메소드 둘다 등록되어 있다면 인터페이스 구현체 먼저 호출하고 그 다음 커스텀 메소드를 호출한다.

커스텀 메소드의 시그너쳐는 다음과 같다.

```
public void anyMethodName(ServiceType service, Dictionary properties);

public void anyMethodName(ServiceType service, Map properties);

public void anyMethodName(ServiceReference ref);
```

ServiceType 자리에는 어떤 타입이든 선언할 수 있다. 해당 타입에 해당하는 서비스가 묶이거나 풀릴때 호출된다. 만약 모든 타입에 대해 콜백을 호출하고 싶으면 java.lang.Object 타입으로 선언하면 된다.

properties 파라미터는 등록된 서비스가 가지고 있는 속성들의 집합을 나타낸다.

리스너가 reference에 등록되어 있다면:

- 레퍼런스가 초기에 서비스와 묶일 때와 서비스가 새로운 서비스로 교체될 때마다 bind 메소드가 실행된다.
- 현재 서비스가 해지되거나, 그 즉시 교체가능한 대체 서비스가 없을 때 unbind 콜백이 호출된다.(물론 해당 reference는 unsatisfied 상태가 된다.)

리스너가 콜렉션에 등록되어 있다면:

- 새로운 서비스가 콜렉션에 추가되면 bind 메소드를 호출한다.
- 서비스가 해지되고 콜렉션에서 제거될 때 unbind 메소드를 호출한다.

콜렉션에는 서비스 교체가 없다는 것을 주목하라. 콜렉션은 그냥 서비스를 추가하고 제거할 뿐이다.

뒷단의 OSGi 서비스에 대한 OSGi serviceChanged 이벤트 처리의 일부로 바인드와 언바인드 콜백은 동기적으로 처리된다.

아래의 테이블은 reference listener 서브 엘리먼트로 가용한 속성들이다.

생략

6.2.4. 리스너와 서비스 프록시들

This page last changed on 7월 03, 2008 by keesun.

임폴트 리스너들은 특정 시점에 묶이는 OSGi 서비스에 접근할 수 있는 방법을 제공하지만, 여기서 중요한 것은 해당 메소드로 넘어온 아규먼트가 실제 서비스 객체가 아닌 프록시라는 것이다. 프록시를 사용하는 이유는 언제 어떻게 바뀔지 모른느 객체에 대하여 강력한 레퍼런스를 가지는 것을 방지하기 위함이다. 서비스들을 추적하는 것이 주목적인 리스너들은 instance equality나 object equlity에 연연해서는 안된다. 이 들 메소드가 해당 인터페이스나 클래스의 public 메소드로 구현한 것이 아니라면, 프록시의 메소드가 호출될 것이다.

따라서 추적은 그냥 서비스 인터페이스나, 서비스 속성(org.osgi.framework.Constants#SERVICE_ID 참조) 또는 서비스 노티(바인드/언바인드)로만 하는 것을 추천한다.

6.2.5. 호출하는 BundleContext에 접근하기

This page last changed on 7월 03, 2008 by keesun.

가져온 서비스가 특정 시점에 어떤 번들을 사용하고 있는지 알고 싶을 수 있다. 이런 시나리오를 돕기 위해, 스플이 DM 이 가져온 서비스는 가져온 번들 BundleContext를 LocalBundleContext 클래스로 공개한다. 가져온 것에 대한 메소드가 호출될 때마다, ThreadLocal을 사용하여 호출하는 BundleContext를 사용할 수 있다. getInvokerBundleContext() 를 통해서..

단 이걸 사용할 때 해당 클래스가 스프링 DM 코드에 의존하게 된다는 것을 주의해야 한다.

6.3. Exporter, Importer listener 베스트프랙티스

This page last changed on 7월 03, 2008 by keesun.

위에서 언급했듯이, 스프링 DM 리스너들은 서비스가 묶이고, 풀리고, 등록되고, 해지될 때를 알기 위한 용도다. 이런 리스너들을 다룰 때 다음의 가이드라인이 도움이 될 수 있을 것이다.

- 오래 걸리는 작업을 리스너 안에서 실행하지 말아라. 만약 그래야만 한다면, 별도의 쓰레드에서 작업하도록 하라. 리스너들이 동기적으로 처리되기 때문에 가능한 빨리 처리되도록 해야 한다. 해당 리스너 안에서 작업을 하는 동안에는 다른 이벤트들과 서비스의 활동이 대기상태가 된다.
- 왠만하면 커스텀 리스너 콜백을 사용하라. 그래야 스프링 DM API에 묶이지 않으며, 특정 이름을 가용하지도 않는다.
- bind/unbind 설정을 반복하고 있다면, 빈 설정 상속 기능을 사용하는 것을 한번 고려해 보아라. 공통적인 설정을 재사용할 수 있다.
- java.util.Dictionary보다는 java.util.Map을 사용하라. 전자는 페기처분 상태다. 그래도 호환성을 위해서 스프링 DM은 리스너에 필요시 Dictionary로 캐스팅할 수 있는 Map을 제공한다.
- 과도한 메소드 사용에 주의하라. 원치 않게도 모든 메소드들이 특정 서비스 타입에 모두 호출 될 수 있다.

```
public class MyListener {
    void register((1)Object service, Map properties);
    void register((2)Collection dataService, Map properties);
    void register((3)SortedSet orderedDataService , Map properties);
}
```

1 Object type - 이 메소드는 항상 실행된다.
2 Collection type - 이 메소드가 실행되면, 바로 위에 있는 메소드도 실행된다.
3 SortedSet type - 이 메소드가 실행되면, 위에 있는 거 모두 실행된다.

6.3.1. 리스너와 cyclic depencdency

This page last changed on 7월 03, 2008 by keesun.

리스너가 자신을 사용하는 서비스를 사용하려는 빈을 참조하려는 경우가 있을 수 있다.

```
<bean id="listener" class="cycle.Listener">                                         (1)
    <property name="target" ref="importer" />                                       (2)
</bean>

<osgi:reference id="importer" interface="SomeService">                              (3)
    <osgi:listener bind-method="bind" ref="listener" />                             (4)
</osgi:reference>
```

1 Listener bean
2 Dependency listener -> importer
3 Importer declaration
4 Dependency importer -> listener

위 선언은 유효하다. 그러나 리스너를 만들려고 하면 레퍼런스 빈을 만들어야 하고 레퍼런스 빈을 만들려면 리스너를 만들어야 한다. 이 원이 깨져서 어느 한쪽이라고 먼저 생성을 하고 설정되어야 한다. 이런 시나리오는 스프링 DM에서 지원한다. 즉 알아서 만들어 준다. 하지만 다음과 같이 내부 빈을 사용한 경우에는 그렇치 않다.

```
<osgi:reference id="importer" interface="SomeService">                              (1)
    <osgi:listener bind-method="bind">                                              (2)
        <bean class="cycle.Listener">                                               (3)
            <property name="target" ref="importer" />                               (4)
        </bean>
    </osgi:listener>
</osgi:reference>
```

1 OSGi service importer
2 Dependency between importer -> listener
3 Nested listener declaration
4 Dependency nested listener -> importer

내부 빈은 자신의 이름이나 라이프사이클도 없이 외부 빈에 종속적인 삶을 살 뿐이기 때문에, 내부 리스너를 캐시 할 수 없다. 따라서 cyclic 참조 원을 깨트리지못하고 동작하지 않게 되는것이다.

정 리하자면, 만약 리스너가 정말로 자신을 사용하는 서비스에 대한 참조가 필요하다면 리스너를 탑 레벨 빈으로 등록하거나 dependency lookup을 사용하라. 대신 후자는 좀 더 많으 설정과 빈 이름을 사용할 것이기 때문에 dependency injection 보다 깨지기 쉽다.

6.4. Service importer global defaults

This page last changed on 7월 03, 2008 by keesun.

osgi 네임스페이스는 모든 가져올 레퍼런스에 설정한 전역 설정을 선언할 수 있는 두 개의 속성을 제공한다.

따라서, osgi 네임스페이스를 사용할 때 내부에 있는 set, list, reference 엘리먼트는 다음 속성을 사용할 수 있다.

- default-timeout: 타임아웃을 명시하지 않은 모든 importer에 기본 타임아웃을 설정할 수 있다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:osgi="http://www.springframework.org/schema/osgi"                    (1)
          osgi:default-timeout="5000">                                           (2)

  <reference id="someService" interface="com.xyz.AService"/>                     (3)

  <reference id="someOtherService" interface="com.xyz.BService"
       timeout="1000"/>                                                          (4)

</beans:beans>
```

(1) osgi 네임스페이스 프리픽스 선언
(2) default-timeout을 루트 엘리먼트에 선언. 만약 기본값이 설정되어 있지 않으면 5분이다. 여기서는 5초로 설정했다. 즉 밀리세컨이라는거..
(3) 이 reference는 기본값을 상속 받아서 타임아웃이 5초다.
(4) 이 reference는 기본값을 재정의해서 1초가 된다.

- default-cardinality: 연관유형을 설정하지 않은 것들의 기본 연관유형을 설정할 수 있다. 가용한 값은 0..X와 1..X다. X는 런타입시에 reference일 경우에는 1로 list나 set일 경우에는 N으로 바뀐다.

```
<beans:beans
      xmlns="http://www.springframework.org/schema/osgi"                         (1)
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:beans="http://www.springframework.org/schema/beans"                  (2)
      xmlns:osgi="http://www.springframework.org/schema/osgi"                    (3)
         osgi:default-cardinality="0..X"                                         (4)
         default-lazy-init="false">                                              (5)

  <reference id="someService" interface="com.xyz.AService"/>                     (6)

  <set id="someSetOfService" interface="com.xyz.BService"/>                      (7)

  <list id="anotherListOfServices" interface="com.xyz.CService" cardinality="1..N"/>      (8)

</beans:beans>
```

(1) 스프링 DM 스키마를 기본 네임스페이스로 선언
(2) 스프링 프레임워크 beans 스키마를 가져오고 해당 네임스페이스 프리픽스를 정한다.(위 예제에선 beans로 했음)
(3) 스프링 DM 스키마를 가져오고 osgi 네임스페이스 프리픽스를 설정한다.
(4) default-cardinality를 루트 엘리먼트에 설정한다. 만약에 기본값을 설정하지 않으면 1..X으로 설정한다. 위의 경우 기본 값을 0..X으로 했고. 이 때 osgi 프리픽스를 붙인것에 주목하라.
(5) beans 엘리먼트 속성(default-lazy-init)은 프리픽스를 사용하지 않았다. since they are declared as being local and unqualified (see the beans schema for more information).
(6) reference 선언은 연관유형을 설정하지 않았기 때문에 기본값을 상속받는다. 즉 0..1이 된다.
(7) set 선언도 연관유형을 설정하지 않았기 때문에 기본값을 상속받는다. 즉 0..N이 된다. 0..1이 아니다
(8) list 선언은 연관유형이 기본값을 재정의하여 설정한대로 1..N 이 된다.

default-* 들을 사용하면 선언을 보다 쉽게 할 수 있고 기본 행위 변경을 간단하게 할 수 있다.(타임아웃 시간 줄이기나 늘리기와 같은..)

6.5. 서비스 Exporter와 서비스 Importer의 관계

This page last changed on 7월 03, 2008 by keesun.

공개한 서비스가 기능을 수행하기 위해 다른 서비스에 의존할 수 있다. 이때 만약 이들 서비스가 필수 레퍼런스라고 가정하고 해당 서비스들이 없어지고 대체제를 찾지 못해서 unsatisfied 상태가 되면 공개한 서비스는 자동으로 서비스 레지스트리에서 해지가된다. 즉 더이상 클라이언트에서 해당 서비스를 사용할 수 없게 된다. 그러나, 해당 필수 레퍼런스가 다시 가용해지면, 공개한 서비스도 다시 서비스 레지스트리에 등록된다.

이런 공개한 서비스의 자동 해지 및 자동 재등록은 오직 명시적인 선언에 의해서만 사용할 수 있다. 만약 서비스 A 빈을 공개한 서비스 S가 서비스 M을 사용하고 있을 때 아래 처럼 명시적으로 선언을 해 둬야. M이 없어지면 S가 해지되고, M이 사용가능 해지면 S도 다시 서비스 레지스트리에 등록된다.

```
<osgi:service id="S" ref="A" interface="SomeInterface"/>

<bean id="A" class="SomeImplementation">
   <property name="helperService" ref="M"/>
</bean>

<!-- the reference element is used to refer to a service
     in the service registry -->
<osgi:reference id="M" interface="HelperService"
     cardinality="1..1"/>
```

하지만 만약 A에서 M으로 의존성이 명시되어 있지 않고, 런타임 시에 M에 대한 레퍼런스를 만들어서 A로 넘긴다면 스프링 컨테이너는 아무일도 해주지 않는다. 스프링 DM은 의존성을 추적하지 않을 것이다.

7장. 번들 다루기

This page last changed on 6월 30, 2008 by keesun.

스프링 DM은 기존에 존재하는 번들 또는 새로운 것을 설치하기 위한 스키마 엘리먼트를 제공한다. 적절한 OSGi 서비스를 대체하기 위해 사용하는 용도가 아니고, bundle은 해당 번들에 application context 라이프사이클에 따라 특정 행위를 할 수 있는 방법을 제동한다.

bundle 엘리먼트는 org.osgi.framework.Bundle 타입의 빈을 정의할 때 사용한다. 그들의 라이프사이클을 주도하는 등의 직접 번들을 다룰 수 있는 간편한 방법이다. 가장 간단하게는 symbolic-name 속성만 기술하면 된다.

```
<bundle id="aBundle" symbolic-name="org.xyz.abundle"/>
```

aBundle이라는 빈은 Bundle 타입의 어떤 속성이든지 주입될 수 있다.

만약 필요한 번들이 설치되어 있지 않다면, location 속성을 사용하여 설치할 위치를 나타낼 수 있으며 또한 action/destroy-action 속성을 사용하여 번들 라이프사이클을 제어할 수 있다. location 속성은 번들 jar 파일이 존재하는 위치를 명시할때 사용한다. action 속성은 번들 객체에 호출되어야 할 라이프사이클 액션을 설정한다. action의 값으로는 insatll, start, update, stop, uninstall이 있다. 이런 액션들은 Bundle 인터페이스에 있는 메소드들 이름과 일치한다.

각각의 action 값들이 현재 번들 상태에 따라 어떻게 해석될지는 다음 표와 같다.

표생략

예제:

```
<!-- ### ### ### ## ### ###.-->
<bundle id="aBundle" symbolic-name="org.xyz.abundle"
   location="http://www.xyz.com/bundles/org.xyz.abundle.jar"
   action="start"/>
```

스프링 DM 예제에 들어있는 virtual-bundle 엘리먼트를 사용하여 OSGi 번들을 실제 존재하는 파일과 별개로 생성하여 사용할 수 있다.

8장. 웹 지원기능

This page last changed on 7월 01, 2008 by keesun.

1.1.0 배포판 부터 도입된 주요 기능은 웹 애플리케이션 지원이고 이로인해 웹 애플리케이션을 OSGi에 배포하기 쉬워졌다.

웹 애플리케이션을 OSGi 위에서 돌리는 가장 큰 문제는 클로스로딩이다. 웹 애플리케이션에는 번들 영역(Bundle Space)라던지 가져온 패키지(Imported pachages)라는 개념이 없다. 각각의 웹 컨테이너들은 자신만의 클래스 로딩 계층구조를 가지고 있는 클래스패스를 사용하여 OSGi 공간과 출돌을 발생시킬 수 있다. 스프링 DM은 이런 문제들을 웹 컨테이너와 OSGi 공간 사이에 다리 역할을 하여 로딩이 더이상 문제가 되지 않도록 하고 있다. 그 기능과는 별개로, 스프링DM에서 웹 지원은 웹 컨테이너와 직접 통합하여 WAR 처리가 말그대로 서버에서 처리된다. 모든 설정과 기능 (non-blocking vs blocking IO, thread pool, specification support(Servlet 2.3, 2.4, 2.5) 등등등)이 가용하다. 모든 타겟 컨테이너의 커스텀 설정 파일과 web.xml 문법이 모두 사용 가능하다.(스프링 DM은 전혀 이걸 파싱하지 않는다.) 요약하자면, 타겟 컨테이너가 지원하는 모든 것들이 스프링 DM을 사용한 OSGi WAR로 가용하다는 것이다.

8.1. 지원하는 웹 컨테이너

This page last changed on 7월 03, 2008 by keesun.

현재까지는, Apache Tomcat 5.5.X/6.0.X 그리고 Jetty 6.1.8+ /6.2.X를 사용할 수 있다.(다른 컨테이너들도 쉽게 끼워넣을 수 있다.). 웹 지원은 JDK 1.4 호환가능하다. 선택한 컨테이너가 필요로 하는 JVM을 확인하라. 일반적으로, Servlet 2.4/JSP 2.0은 JDK 1.4를 사용하고 Servlet 2.5/JSP2.1은 JDK 1.5를 필요로 한다.

8.2. Web 지원기능 사용법

This page last changed on 7월 03, 2008 by keesun.

WAR가 아닌 번들처럼, 스프링 DM 웹은 extender pattern을 사용하여 WAR를 찾고 설치한다. 하지만, 표준 스프링 DM Extender와 가장 큰 차이는 오직 스프링 DM이 WAR를 설치와 제거를 실행한다는 것이다. 실제 웹 애플리케이션 생성과 쓰레드 관리는 WAR가 설치된 웹 컨테이너에게 맡긴다. 스프링 DM 웹은 단지 언제 WAR를 웹 컨테이너로 배포할지와 언제 내릴지를 알려줄 뿐이다. 웹 애플리케이션을 생성하고 관리하는 것은 컨테이너에게 달려있다.

스프링 DM 웹을 사요하려면 다음을 설치해야한다.:

- spring-osgi-web.jar: 스프링 DM 웹 지원
- spring-osgi-web-extender.jar: 스프링 DM 웹 extender

시작한 OSGi WAR 번들을 찾고 그들을 지원하는 웹 컨테이너에 설치하기 위한 번들들이다. 기본적으로 Tomcat이 사용된다 하지만 이걸 Jetty나 다른 커스텀 서버로 변경할 수 있다. war 번들이 멈추면, 스프링 DM은 그와 관련된 웹 애플리케이션을 중지시키고 내린다. 기존의 웹 개발과의 차이점은, 서블릿 클래스들이 항상 우선권을 가지는 OSGi 클래스패스에 명시적으로 import 되야 한다는 것이다.

웹 애플리케이션을 실행하는 건 웹 컨테이너기 때문에, 스프링 설정 파일들을 META-INF/spring 폴더에 넣거나 스프링 DM manifest들을 사용할 필요가 없다. 그냥 간단하게 WAR에 있는 파일들을 번들로 묶고 웹 프레임워크를 사용하여 스프링 컨테이너를 시작시키면 된다. 스프링 MVC 연동을 할 수 있다. 스프링 Extender도 계속해서 필요한데, 이 게 있어야 네임스페이스를 처리할 수 있다. 만약에 이게 없으면, 스프링 네임스페이스(osgi:, aop: 심지어 beans: 까지도)를 처리할 수 없다.

8.3. OSGi 내에서 WAR 클래스패스

This page last changed on 7월 03, 2008 by keesun.

서블릿 스펙에는 WAR 내부에 특별한 의미를 지니는 위치와 몇가지 규칙들을 정의하고 있다. 이번 섹션에서는 이것들이 OSGi 환경에서 어떻게 처리되는지 살펴볼 것이다.

8.3.1. Static Recsources

This page last changed on 7월 03, 2008 by keesun.

WAR 번들을 설치할 때, static resource들의 경우, 스프링 DM은 번들 영역에서 가용한 것이 무엇인지만 신경쓴다. 이게 무슨 뜻이냐면 번들 jar 내부에 있는 것들과 거기에 붙어있는 fragment만을 사용할 수 있다는 것이다. 서블릿 스펙에 보면, 웹 애플리케이션에 접속한 클라이언트가 아니라 WEB-INF 폴더 이하에 있는 자원들만ServletContext API를 통해 사용 가능하다고 되어있다. 게다가 클래스패스(imported pachages, required bundles 또는 dynamic imports)에 있는 모든 자원은 애플리케이션 코드에서 읽어들여 사용할 수 있지만 그 밖에서는 참조할 수 없다.

8.3.2. 서블릿

This page last changed on 7월 03, 2008 by keesun.

기존의 WAR 배포와의 주요 차이점은 서블릿 패키지들을 WAR 번들에서 사용할 수 있도록 명시적으로 import되어야 한다는 것이다. 해당 패지키들을 Import -Package 속성에 추가한다.

```
Import-Package: javax.servlet,javax.servlet.http,javax.servlet.resources
```

추가적으로, 서블릿 스펙은 WAR의 클래스패스를 몇몇 미리 정의된 위치에 근거하고 있다. 빠르게 살펴보면 다음과 같다:

- WEB-INF/classes: 모든 리소드들은 WEB-INF/Classes 밑에 있다
- WEB-INF/lib/*.jar: 모든 jar 파일들은 WEB-INF/lib 밑에 있다.

여 기에 추가로, 각각의 컨테이너 구현체들은 WAR 클래스패스에 추가되는 common 라이브러리들을 제공할 수 있다. OSGi 이기 때문에, WAR 클래스패스는 다시 구성한다. 스프링 DM은 미리 정의된 위치는 무시하고 항상 OSGi 클래스패스를 사용한다. 즉 WEB-INF/classes에 포함되어 있거나 WEB-INF/lib에 포함되어 있다 하더라도 WAR 번들이 import 한 패키지들만 사용할 수 있다는 것이다. 다시 말하자면, WEB-INF/classes 밑에 있는 모든 클래스들중에 WAR OSGi 클래스패스에서 가용하지 않은 것들은 없는거나 마찬가지다.

미리 정의된 WAR 위치들을 설정하는 가장 쉬운 방법은, 그것들을 번들 클래스패스에 추가하는 것이다.

```
Bundle-Classpath: .,WEB-INF/classes,WEB-INF/lib/some.jar,WEB-INF/lib/lib.jar
```

번들 클래스패스에 추가하기 전에 그들을 OSGi 번들로 설치할 수 있는지 없는지 생각해야한다. 그렇게 함으로써 글르을 다른 WAR에서도 사용할 수 있고 OSGi 버전잉을 할 수 있다. 동일한 VM에 같은 라이브러리의 상이한 버전이 여럿 존재하는 것이 가능해 진다.

8.3.3. JSP

This page last changed on 7월 03, 2008 by keesun.

JSP 를 처리하려고, 스프링 DM은 Tomcat Jasper2 엔진을 통합했다. 즉 JSP 1.2, 2.0, 2.1을 지원한다는 것이다. OSGi 화된 버전들은 스프링 DM 리파지토리에서 가용하다. OSGi 번들이 Jasper 클래스들을 import할 필요가 없다.

8.3.3.1. 태그 라이브러리들

This page last changed on 7월 03, 2008 by keesun.

JSP 스펙은 태그 라이브러리 생성하여 "JSP 페이지에서 재사용할 수 있는 기들을을 모듈화 하여 선언할 수 있도록"했다. 또한 재사용 가능한 taglib들은 자바 클래스들(Tag 구현체)와 어떤 태그들을 사용할 수 있는 기술한 파일을 컴포넌트로 가지고 있다. 스프링 DM은 taglib들을 jar 형태로 WEB-INF/lib 밑에 두거나 풀어져있는 상태로 WEB-INF/classes 밑에 두는 JSP 컨벤션을 확장하여, 번들 클래스패스(imported pachages, required bundles)에 있는 모든 taglib들을 찾아낸다.

스프링 DM은 번들 클래스패스 내에서 자동으로 taglib 파일(*.tld)들을 찾아서 그들을 Jasper 엔진에서 사용할 수 있게 해준다. 하지만, tag 정의가 자동으로 찾아지는 반면 tag 클래스들은 그렇지 않다. 여기서도 OSGi 클래스패스가 우선한다. 즉, 태그를 사용하려면, war 번들이 tag에 해당하는 클래스를 import 해야한다. 안그러면 태그를 사용할 수 없다.

많은 태그를 방출하는 라이브러리들을 다룰 때, Impoert-Pachage 대신 Required-Bundle 헤더를 사용할 수 있다.

```
Require-Bundle: org.springframework.osgi.jstl.osgi
```

위의 헤더를 사용하면, JSP Standard Tag Library(JSTL)이 내보내는 모든 클래스들을 war 번들이 참조할 수 있고 따라서 JSP 내에서 사용할 수 있다.

주의할 것

Require-Bundle을 사용하기 전에 그 용법에 대해 OSGi 스펙 3.13 참조하라.

war 클래스패스에 어떤 매카니즘을 사용하든 상관없이, 여러 WAR들이 그것들을 공유하게 할 수 있다. 각각의 번들들은 오직 패키지만 import 할 수 있고 라이브러리 jar를 통째로 가져오진 못한다. 사실, 다른 번들들이나 Jar에 들어있는 패지키들을 선택적으로 사용하여 원하는 행위를 도출해 낼 수 있다. 매우 막강한 기능으로 웹 애플리케이션 배포를 용이하게 한다.

8.4. web extender 설정하기

This page last changed on 7월 03, 2008 by keesun.

core extender와 마찬가지로, 웹 Extender도 OSGi fragment로 설정될 수 있다. 같은 패턴을 사용해서, 웹 extender는 자신의 번들 영역에있는 META-INF/spring 폴더 밑에 있는 XML 파일들을 찾고 그것들을 하나의 application context로 조립하여 자신의 설정파일로 내부적으로 사용한다. 이런 기본 설정을 재정의 하려면, 아래 표에서 적당한 빈 이름을 찾아서, 그것들을 원하는 대로 정의한 다음 그걸 다음의 코드를 사용하여 fragment로 spring-osgi-web.jar에 붙이면 된다.

Fragment-Host: org.springframework.bundle.osgi.web.extender

다음의 빈들이 현재 Web extender에 의해 인식가능하다.

표생략

8.4.1. 웹 배포자 변경하기

This page last changed on 7월 03, 2008 by keesun.

톰캣 배포자에서 제티로 바꾸는 예제는 다음과 같다. META0INF/spring/jetty-deployer.xml에 다음과 같이 설정한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="warDeployer"                                                    (1)
        class="org.springframework.osgi.web.deployer.jetty.JettyWarDeployer" />          (2)
</beans>
```

1 web extender에 의해 사용되고 있는 미리 정의한 빈 이름
2 org.springframework.osgi.web.deployer.WarDeployer 인터페이스를 구현하고 있는 Bean 구현체

파일을 만들었으면, 스프링 DM 웹 Extender 번들d에 OSGi fragment로 붙어있는 번들이 되어야 한다. 따라서 Fragment-Host 헤더를 사용한다.

```
Fragment-Host: org.springframework.bundle.osgi.web.extender
```

자 이제 위의 조각(Fragment)이 spring-osgi-web.jar 번들에 끼워졌기 때문에 웹 애플리케이션을 제티에 배포한다.

미리 만들어둔 제티 Fragment를 스프링 DM 메이븐 저장소에서 jetty.extender.fragment.osgi 아티팩트id로 사용할 수 있다.

8.5. 표준 배포자 커스터마이징 하기

This page last changed on 7월 03, 2008 by keesun.

디플로이어는 시작시에 필요한 서비스들을 룩업한다. 필수 요소인 경우 디플로이어는 해당 서비스를 참조할 수 있을 때까지 기다리고, 만약 시간이 지나도 참조할 수 없을 때는 예외를 발생시킨다. 만약에 기본 타임아웃 설정이나 서비스 룩업 필터가 정의되어 있지 않으면, 스프링 DM의 reference 엘리먼트를 사용하여 설정할 수 있다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:osgi="http://www.springframework.org/schema/osgi"
   xmlns:p="http://www.springframework.org/schema/p"                                    (4)
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/osgi
     http://www.springframework.org/schema/osgi/spring-osgi.xsd">

    <osgi:reference id="myTomcatServer"                                                 (1)
        interface="org.apache.catalina.Service"
        filter="(environment=testing)"
        cardinality="0..1"/>

    <bean id="warDeployer"                                                              (2)
        class="org.springframework.osgi.web.deployer.tomcat.TomcatWarDeployer"
        p:service-ref="myTomcatServer"/>                                                (3)

</beans>
```

1 사용자 정의 OSGi 서비스 룩업
2 디플로이어 정의Deployer definition (이름이 중요하다.)
3 서비스 속성 할당(p namespace 사용)
4 스프링의 p 네임스페이스 선언

위의 설정에 필요한 패키지를 import 해야 한다는 것을 주의 하자. 그리고 위는 웹 Extender에서 사용해야 하기 때문에 Fragment로 설정한다. 우선 import 해야 하는 패키지는 Catalina 패키지고, 그안에 있는 Service 인터페이스는 다른 패키지에 있는 Connector라는 것의 의존하고 있다. 따라서 다음과 같이 설정한다. 그러지 않으면, ClassNotFoundException 이나 NoClassDefFoundException이 발생한다.

```
# Catalina packages
Import-Package: org.apache.catalina,org.apache.catalina.connector
# Spring-DM Web Extender
Fragment-Host: org.springframework.bundle.osgi.web.extender
```

8.6. OSGi에 배포할 수 있는 라이브러리와 웹 개발

This page last changed on 7월 03, 2008 by keesun.

불행히도, 현재 대부분의 웹 개발용 라이브러리들은 OSGi 번들이 아니다. 즉 그 라이브러리들을 OSGi 공간에서 사용하려면 다른 번들에 내장된 채로 사용하는 수밖에 없다. 이 문제를 해결하기 위해서, 스프링 DM 프로젝트는 흔히 사용하는 라이브러리들을 OSGi에서 사용할 수 있는 형태로 바꾼 것들을 Maven 리파지토리에 올려서 제공하고 있다.

8.6.1. Deploying web containers as OSGi bundles

This page last changed on 7월 03, 2008 by keesun.

스프링 DM은 웹 컨테이너를 OSGi 서비스로 설치하는 것이 가능하게 한다. Tomcat 이나 Jetty 배포판이 그런 일을 하지는 않기 때문에 스프링 DM은 간단하지만 유용한 두 개의 OSGi Activator를 스프링 DM 리파지토리에서 제공한다. 어떤게 설치되면, 프로그래밍적으로 기본 설정(커스터마이징 할 수 있다. activator는 공용이지만 설정은 쉽게 변경할 수 있다.)에 따라 적절한 웹 컨테이너를 시작시킨다.

8.6.1.1. Tomcat 5.5.x, 6.0.x

This page last changed on 7월 03, 2008 by keesun.

아파치 톰캣 버전 5.5.x 와 6.0.x는 OSGi 구성물 형태로 catalina.osgi artifactId를 가지고 있다. 이 jar들은 오직 commons-logging, JMX, Servlet/JSP 라이브러리만을 필요로 한다.

게다가 저장소에는 톰캣 Activaor도 있는데 이들의 이름은 catalina.osgi.start로 되어있다. activator는 톰캣 XML 설정을 이해하며 서버를 localhost, 8080 포터에서 실행하기위한 기본 설정을 가지고 있다. 이런 기본 설정은 conf/server.xml 을 톰캣 activator에 Fragment로 붙여서 기본 설정을 재정의할 수 있다.

fragment를 붙이려면, 다음과 같이 manifest에 추가한다.

```
Fragment-Host: org.springframework.osgi.catalina.start.osgi
```

8.6.1.2. Jetty 6.1.8+ , 6.2.0

This page last changed on 7월 03, 2008 by keesun.

Jetty 는 기본적으로 OSGi에서 사용할 수 있는 형태다. 따라서 어떤 변형을 하지 않아도 OSGi 플랫폼에 설치할 수 있다. 하지만, activator가 없기 때문에 스프링 DM이 톰캣 activator같은 걸 하나 제공해준다. 이 activator는 jetty.start.osgi 라는 이름을 가지고 있다. 톰캣 activator와 마찬가지로 이녀석도 localhost에 8080 포트로 기본 설정을 포함하고 있다. 이 기본 설정을 바꾸려면 etc/jetty.xml 에 설정을 한다음 fragment로 붙이면 된다.

```
Fragment-Host: org.springframework.osgi.jetty.start.osgi
```

extender처럼, 각각의 activator들은 사용자가 아무것도 제공하지 않으면 기본값을 사용한다.

8.6.2. Common libraries

This page last changed on 7월 03, 2008 by keesun.

Servlet, JSP, Standard Taglib, Commons-EL 및 기타 라이브러리들을 스프링 DM 저장소에서 이용할 수 있다.

8.7. 스프링 MVC 통합

This page last changed on 7월 03, 2008 by keesun.

1.1부터 스프링 DM은 스프링 MVC 프레임워크와 밀접하게 통합되었다. 이번 섹션에서 어떻게 스프링 MVC 애플리케이션을 OSGi 환경에서 실행할 수 있는지 살펴보겠다.

OSGi 플랫폼에서 제대로 사용되려면, 스프링 MVC를 새로운 환경으로 이전해야 한다. 스프링 DM은 OSGi를 인식하는 application context를 제공한다.(OsgiBundleXmlWebApplicationContext) 이것은 스프링 MVC의 XmlApplicationContext와 동일한 역할을 한다. application context는 웹 애플리케이션 BundleContext를 알고 있고 따라서 OSGi 영역에 있는 자원들을 읽고 OSGi 서비스를 공개하고 BundleContextAware를 지원하고 클래스패스에 포함되어있는 번들들을 컴포넌트스캔할 수 있다.

이 애플리케이션 컨텍스트를 사용하려면 스프링의 ContextLoaderListener의 contextClass라는 파라미터와 DispatcherServlet을 사용하면 된다.

```
<context-param>
  <param-name>contextClass</param-name>
        (1)
  <param-value>
org.springframework.osgi.web.context.support.OsgiBundleXmlWebApplicationContext
</param-value>    (2)
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
             (3)
</listener>

<servlet>
  <servlet-name>petclinic</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
             (4)
  <load-on-startup>2</load-on-startup>
  <init-param>
    <param-name>contextClass</param-name>
    (5)
    <param-value>
org.springframework.osgi.web.context.support.OsgiBundleXmlWebApplicationContext
</param-value>  (2)
  </init-param>
</servlet>
```

(1) 스프링의 ContextLoaderListner가 최상위 웹 애플리케이션 컨텍스트 타입을 알기 위한 context-param 이름
(2) OSGi를 인식하는 web application context 풀네임
(3) 스프링 설정 읽어들이는 리스너
(4) 스프링 MVC 젤 앞단 컨트롤러
(5) 스프링의 DispatcherServlet이 웹 애플리케이션 컨텍스트 타입을 알기 위한 init-param

위와같이 설정하면, 스프링 DM 번들은 구동중인 BundleContext를 가져다 사용할 수 있으며 OSGi 환경을 인식할 수 있다.

노트

물론 스프링 MVC 애플리케이션에는 적절한 import 문들이 들어가있어야 한다. 즉 WAR도 번들이기 때문에 manifest 파일에 적절하게 설정되어 있지 않으면, 클래스패스가 맞지 않게되고 따라서 제대로 동작하지 않을 것이다.

9장. OSGi 기반 애플리케이션 테스트하기

This page last changed on 7월 01, 2008 by keesun.

베스트 프랙티스를 잘 따르고 스프링 DM 지원 기능을 사용한다면, 여러분들이 작성한 bean 클래스들은 단위 테스트가 용이할 것이다. OSGi를 직접 참조하지도 않을 것이고, 최소한의 인터페이스 기반의 OSGi API(BundleContext와 같은..) 만을 사용하여 mock 객체를 만드는 것도 쉬워질 것이다. 단위 테스트와 통합 테스트 둘 다 스프링 DM이 지원한다.

9.1. OSGi Mocks

This page last changed on 7월 03, 2008 by keesun.

대부분의 OSGi API들이 인터페이스고 EasyMock같은 라이브러리를 사용하여 목객체를 만들어 사용하는 것이 간단할 수도 있지만, 실제로는 상당히 많은 코드를 필요로 한다.(특히 JDK 1.4에서..) 테스트를 짧고 간결하게 유지하기위해, 스프링 DM은 org.springframework.osgi.mock 패키지 아래에 OSGi 목들을 제공한다.

이것들이 유용한지 아닌지는 여러분들에게 달려있다. 우린 스프링 DM 테스트 스위트를 만들 때 이들을 상당히 많이 사용했다. 아래의 코드는 그런 테스트 코드에서 흔히 볼 수 있는 코드 조각이다.

```
private ServiceReference reference;
private BundleContext bundleContext;
private Object service;

protected void setUp() throws Exception {
    reference = new MockServiceReference();
    bundleContext = new MockBundleContext() {

        public ServiceReference getServiceReference(String clazz) {
            return reference;
        }

        public ServiceReference[] getServiceReferences(String clazz, String filter)
                throws InvalidSyntaxException {
            return new ServiceReference[] { reference };
        }

        public Object getService(ServiceReference ref) {
            if (reference == ref)
               return service;
            super.getService(ref);
        }
    };

    ...
}

public void testComponent() throws Exception {
    OsgiComponent comp = new OsgiComponent(bundleContext);

    assertSame(reference, comp.getReference());
    assertSame(object, comp.getTarget());
}
```

마무리 하자면, 스프링 DM이 제공하는 라이브러리들을 사용해보고 여러분들이 가장 편한다고 느끼는 라이브러리를 사용하라. 우리가 작성한 테스트 스위트에서는 EasyMock 라이브러리와 통합 테스트를 많이 사용했다.

9.2. 통합 테스트

This page last changed on 7월 03, 2008 by keesun.

OSGi처럼 제한적인 환경에서는 여러분들이 작성한 클래스의 가시성과 버전관리, 번들이 다른 번들과 제대로 동작하는지를 확인하는 것이 중요하다.

통합 테스트를 편하게 하기위해, 스프링 DM 프로젝트는 테스트 클래스 계층구조를 제공한다 (org.springframework.osgi.test.AbstractOsgiTests를 기반으로) . 이걸 사용해서 자동으로 OSGi 환경에서 실행되는 일반적인 JUnit 테스트 케이스를 작성할 수 있다.

보통 스프링 DM 프로젝트가 지원하는 시나리오는 다음과 같다:

- OSGi 프레임워크 시작하기(Equinox, Knopflerfish, Felix)
- 테스트에 필요하다고 기술된 번들 설치 및 시작
- 테스트 케이스를 fly 번들 안으로 패키징하고, manifest 작성하고 OSGi 프레임워크에 해당 번들을 설치하기
- OSGi 프레임워크 안에서 테스트 케이스 실행하기
- 프레임워크 종료하기
- 테스트 결과를 OSGi 밖에 있는 원래 테스트 케이스 객체에 전달하기

> 주의할 것
>
> 테스팅 프레임워크의 목적은 OSGi 통합 테스트를 OSGi 환경 밖에서 테스트 하는 것이다(Ant/Maven 같은 곳에서). 테스팅 프레임워크는 OSGi 번들로 사용하기 위해 만들어 둔 것이 아니다.

다음을 쭉 살펴보면 JUnit 기반 통합 테스트를 작성하는 것이 쉬워질 것이다.

본 챕터의 나머지는 스프링 DM 테스팅 스위트가 제공하는 기능들을 설명한다.

9.2.1. 간단한 OSGi 통합 테스트 작성하기

This page last changed on 7월 03, 2008 by keesun.

진단 프레임워크에는 특정 기능을 제공하는 여러 클래스들이 있는데, 여러분이 작성한 대부분의 클래스들은 org.springframework.osgi.test.AbstractConfigurablBundleCreatorTests를 상속하면 된다.

이 클래스를 확장하고 BundleContext 필드를 사용하여 OSGi 플랫폼을 사용해보자.

```
public class SimpleOsgiTest extends AbstractConfigurableBundleCreatorTests {

public void testOsgiPlatformStarts() throws Exception {
    System.out.println(bundleContext.getProperty(Constants.FRAMEWORK_VENDOR));
    System.out.println(bundleContext.getProperty(Constants.FRAMEWORK_VERSION));
    System.out.println(bundleContext.getProperty(Constants.FRAMEWORK_EXECUTIONENVIRONMENT));
    }
}
```

간단하게 JUnit 테스트를 실행하듯이 테스트를 실행하면 된다. Equinox 3.2.x 에서 실행하면 다음과 같은 결과를 얻을 수 있다

```
Eclipse
1.3.0
OSGi/Minimum-1.0,OSGi/Minimum-1.1,JRE-1.1,J2SE-1.2,J2SE-1.3,J2SE-1.4}
```

보시다시피, 테스팅 프레임워크는 테스트 실행에 필요한 번들들 필요한 번들들(스프링, 스프링 DM, slf4j ...)을 자동으로 설치해준다.

9.2.2. 테스트에 필요한 것들 설치하기

This page last changed on 7월 03, 2008 by keesun.

스프링 DM jar와 테스트 자체를 제외하면 몇몇 라이브러리나 통합 테스트 대상이 되는 코드에 대한 의존도가 높다.

아파치 커먼스 Lang에 의존하고 있는 다음의 테스트 코드를 살펴보자.

```
import org.apache.commons.lang.time.DateFormatUtils;
    ...
      public void testCommonsLangDateFormat() throws Exception {
        System.out.println(DateFormatUtils.format(new Date(), "HH:mm:ssZZ"));
    }
}
```

위 코드를 실행하면, 다음과 같은 예외가 발생할 것이다.

```
java.lang.IllegalStateException: Unable to dynamically start generated unit test bundle
     ...
Caused by: org.osgi.framework.BundleException: The bundle could not be resolved.
Reason: Missing Constraint: Import-Package: org.apache.commons.lang.time; version="0.0.0"
    ...
    ... 15 more
```

테스트는 org.apache.commons.lang.time 패키지를 필요로 하지만, 이걸 공개한 번들은 존재하지 않는다. 이 문제를 해결하기 위해 commons-lang 번들을 설치해본자.(2.4 이상의 버전을 설치해야 한다. 그래야 OSGi 라이브러리다.)

설치되어야 할 번들을 getTestBundlesNames 또는 getTestBundles를 사용해서 명시할 수 있다.

기 본적으로 테스트 스위트는 로컬 maven2 저장소를 사용하여 구성물을 위치시킨다. 구성물 그룹 아이디, 이름, 버전을 콤마로 구분한 문자열로 나열한 것을 기본 locator가 읽어서 해당 구성물을 찾아서 설치한다. 커스텀 locator를 구현하여 설정할 수 있다. org.springframework.osgi.test.provisioning.ArtifactLocator 인터페이스를 구현한 다음 끼워 맞추면 된다.

통합 테스트를 다음과 같이 수정하여 필요한 번들을 설치하자.

```
protected String[] getTestBundlesNames() {
     return new String[] { "org.springframework.osgi, cglib-nodep.osgi, 2.1.3-SNAPSHOT",
         "org.springframework.osgi, jta.osgi, 1.1-SNAPSHOT",
         "commons-lang, commons-lang, 2.4" };
     };
}
```

테스트를 다시 실행하면 위의 번들들이 OSGi 플랫폼에 설치된다.

노트

위에서 언급한 구성물들이 여러분의 로컬 메이븐 저장소에 있어야 한다.

9.2.3. Advanced testing framework topics

This page last changed on 7월 03, 2008 by keesun.

테스팅 프레임워크는 커스터마이징을 많이 할 수 있다. 이 번 챕터에서 여러분에게 유용한 후킹을 소개하고자 한다. 하지만, 이런 팁들이 테스트 코드의 복잡도를 높일 수도 있다.

9.2.3.1. 테스트 manifest 커스터마이징

This page last changed on 7월 03, 2008 by keesun.

자동 생성된 manifest가 테스트에 적당하지 않을 수도 있다. 예를 들어 manifest에 몇몇 새로운 헤더나 필수가 아니라 부가적인 import가 필요할때가 있다.

간단한 경우, 자동 생성된 manifest를 조작할 수 있다. 아래의 예제에서 번들 클래스패스를 명시적으로 설정하고 있다.

```
protected Manifest getManifest() {
      // let the testing framework create/load the manifest
      Manifest mf = super.getManifest();
      // add Bundle-Classpath:
      mf.getMainAttributes().putValue(Constants.BUNDLE_CLASSPATH, ".,bundleclasspath/simple.jar");
      return mf;
}
```

또 다른 방법으로 여러분이 직접 작성한 manifest 파일을 getManifestLocation() 메소드에 설정할 수도 있다.

```
protected String getManifestLocation() {
      return "classpath:com/xyz/abc/test/MyTestTest.MF";
}
```

둘 모두 manifest에 다음과 같이 설정되어 있어야 한다.

`"Bundle-Activator: org.springframework.osgi.test.JUnitTestActivator"`

저렇게 설정되어 있지 않으면, 테스팅이 제대로 동작하지 않을 것이다. 또한 JUnit, Spring, Spring DM의 특정 패키지들을 import 해줘야 된다.

```
Import-Package: junit.framework,
  org.osgi.framework,
  org.apache.commons.logging,
  org.springframework.util,
  org.springframework.osgi.service,
  org.springframework.osgi.util,
  org.springframework.osgi.test,
  org.springframework.context
```

테스트 클래스가 사용할 패키지 import에 실패하면 테스트는 실패할 것이고 NoDefClassFoundError가 발생할 것이다.

9.2.3.2. 테스트 번들 컨텐츠 커스터마이징

This page last changed on 7월 03, 2008 by keesun.

기본적으로, 테스트 대상이 되는 번들을 테스트 할 때 테스팅 인프라는 ./target/test-classes 폴더 밑에 있는 모든 클래스, xml, 프로퍼티 파일들을 사용한다. 이건 메이븐 구조에 맞게 설계된거다. 이런 설정을 두 가지 방법으로 변경할 수 있다.

- AbstractConfigrableBundleCreatorTests getXXX 메소드를 구현하는 프로그래밍을 하는 방법.
- 테스트 케이스와 비슷한 이름을 가진 프로퍼티 파일을 작성하는 선언적인 방법. 예를 들어, com.xyz.MyTest 가 사용할 설정 프로퍼티 파일은 com/xyz/MyTest-bundle.properties 파일이어야 한다.

표 9.1. 기본 테스트 jar 컨텐츠 설정

생략

이 옵션은 특정 리소스를 필요로 하는 테스트를 작성할 때 유용하다. AbstractConfigurableBundleCreatorTests와 AbstractOnTheFlyBundleCreatorTests를 살펴보면 보다 많은 후킹 메소드를 찾을 수 있을 것이다

9.2.3.3. MANIFEST.MF 생성 이해하기

This page last changed on 7월 03, 2008 by keesun.

테스트 번들 컨텐츠를 기반으로 테스트 manifest를 자동 생성 해주는 테스팅 프레임워크의 유용한 기능 중 하나다. 바이트코드를 분석하여 필요로 하는 패키지들을 import 하는 manifest를 작성해준다. 생성된 번들은 테스트를 실행하기 위한 용도기 때문에, 다음과 같은 가정을 전제로 한다.

- 아무런 패키지도 export하지 않는다.
- 패키지를 나누는 기능(서로 다른 번들에서 같은 패키지에 클래스들을 제공하는 기능)을 지원하지 않는다. Split package는 OSGi 스펙 3.13에 따라서 권장하지 않는다.
- 테스트 번들은 오직 테스트 클래스들만 가지고 있다. 기본 설정을 변경하려면, createManifestOnlyFromTestClass 가 true를 반환하도록 재정의하라.

```
protected boolean createManifestOnlyFromTestClass() {
    return true;
}
```

노트

번들에 담겨있는 클래스들의 수와 크기에 Manifest를 작성하는데 필요한 시간이 비례한다.

일반적인 OSGi 번들용 manifest를 작성하는 것이 목적이 아니라, 빠르고 간단하게 테스트를 위한 것을 만들기 위한 용도이다.

9.2.4. OSGi application context 만들기

This page last changed on 7월 03, 2008 by keesun.

스프링 DM 테스팅 스위트는 스프링 테스팅 클래스들을 기반으로 하고 있다. application context를 만들려면, getConfigLocation 메소드를 재정의하여 application context 설정파일 위치를 알려주면 된다. 실행시에, OSGi application context가 생성되고 테스트 케이스를 실행할 동안 캐쉬될 것이다.

```
protected String[] getConfigLocations() {
   return new String[] { "/com/xyz/abc/test/MyTestContext.xml" };
}
```

9.2.5. 사용할 OSGi 플랫폼 설정하기

This page last changed on 7월 03, 2008 by keesun.

테스팅 프레임워크는 3개의 OSGi 4.0 구현체, Equinox, Knopflefish, Felix를 지원한다. 이걸 사용하려면 테스트 클래스패스에 이들이 존재해야 한다. 기본적으로 테스팅 프레임워크는 Equinox를 사용한다. 이 설정을 바꾸는 방법은 두 가지다.

- getPlatformName() 메소드를 사용하는 프로그래밍적인 방법.

Platforms 인터페이스 구현체의 이름을 설정하여 사용할 플랫폼을 알려준다.

```
protected String getPlatformName() {
   return Platforms.FELIX;
}
```

- org.springframework.osgi.test.framework 시스템 속성을 사용한 선언적인 방법

만약 이 속성이 설정되면, 테스팅 프레임워크는 이 값을 플랫폼 구현체의 풀 네임 값으로 사용할 것이다. 만약 문제가 생기면, Equinox를 다시 사용할 꺼고 경고 메시지를 로깅한다. 이 옵션은 (Ant나 Maven같은) 빌드 툴에게 유용하다. 왜냐면 테스트 코드를 변경하지 않고도 특정 타켓 환경을 알려줄 수 있으니까.

9.2.6. 테스트 의존성 대기하기

This page last changed on 7월 03, 2008 by keesun.

테스팅 프레임워크에 내장된 기능 중 하나는 테스트를 실행할 때 필요한 의존성들이 설치 될 때까지 기다리는 것이다. OSGi 플랫폼의 동시성 때문에, 번들이 설치됐다고 해서 해당 번들이 제공하는 서비스가 동작중이라는 것을 보장할 수는 없다. 테스트가 필요로 하는 서비스가 완전히 가용하기 전에 테스트를 실행하는 건 테스트 결과를 더럽히는 일이다. 기본적으로 테스팅 프레임워크는 사용자가 설치한 번들을 모두 조사해서, 스프링을 사용한 번들일 경우 해당 번들이 완전히 시작될 때까지(시작 상태가 되면 서비스 설치된거니까..) 기다린다. 이 기본 행위를 shouldWaitForSpringBundlesContextCreation 메소드에서 변경할 수 있다. AbstractSynchronizedOsgiTests를 참조하라.

9.2.7. 테스팅 프레임워크 성능

This page last changed on 7월 03, 2008 by keesun.

테스팅 프레임워크가 제공하는 모든 기능을 고려했을 때, 성능적인 측면에서 병목현상을 야기하지는 않을지 의심스러울 것이다. 먼저, 테스팅 프레임워크가 자동으로 해주는 모든 일들은 어떻게 해서든 해야만 하는 일들이다. 그 작업들을 손수하기에는 에러가 발생할 여지가 너무 많고 시간 소비도 만만치 않다.

현재의 인프라는 거의 반년간 사용해 왔다. 통합 테스트(120개 정도)를 실행하는데 노트북에서 3분 30초 가량 걸린다. 대부분의 시간은 OSGi 플랫폼을 시작하고 끄는데 소비된다. "테스팅 프레임워크"는 오직 10%가량의 시간만 소비한다.

하지만 우린 필요한 설정을 보다 최소화하고, 보다 더 빠르고 똑똑하게 만들려고 한다. 좋은 아이디어가 있으면 이슈트래커나 포럼을 통해서 언제든지 제안해달라.