

OSGi Intro

Brief Introduction to OSGi

Topics in this Session



- What is OSGi?
- Why OSGi?
 - Describe the problem
 - Describe the solution
- OSGi Quick Start
- Intro to OSGi Constraint resolution

What is OSGi?



"OSGi technology is the dynamic module system for Java™. The OSGi Service Platform provides functionality to Java that makes Java the premier environment for software integration and thus for development"

Peter Kriens blog: http://www.osgi.org/About/Technology?section=2_

- OSGi module (a.k.a Bundle) – JAR
- Application = collection of Bundles
- Bundles can expose OSGi Services
- OSGi Services are dynamic
 - started/stopped
 - installed/uninstalled
 - updated, etc. . .

- Open Specification managed by OSGi Alliance
 - <http://www.osgi.org>
- Founded in March 1999
 - Based on the realized need for light weight dynamic platform
 - Initially targeted network devices
 - Since 2006, server side adoption
- Member companies
 - IBM, SpringSource, Motorola, Oracle, Tibco etc. . .

- What is OSGi?
- **Why OSGi?**
 - **Describe the problem**
 - Describe the solution
- OSGi Quick Start
- Intro to OSGi Constraint resolution

Software complexity

- Dependency on knowledge of internal structure of the component and its dependencies
- Third party library dependency
 - Could lead to mismatch between versions your component depends on vs. version another component depends on
 - Recursive dependency – depending on other's dependency

Software complexity (cont. . .)

- No concept of optional dependency
- Linear class-path
 - Leads to only one version of the library per deployment
 - Version is not a runtime artifact – no way to specify version at runtime, thus no way to use multiple versions of a particular library per deployment unit

Why OSGi? (problem statement)



- Developer's ideal world
 - Write new Software
- Developer's reality
 - Integrate with existing software
 - Deal with over-frameworked architectures
- Software is just a mechanism which enables automation of real business problems and processes

So what are business problems and processes?

Why OSGi? (problem statement)



Business processes are dynamic:

- Can come and go or be updated at any time
 - **Example:** *Company introduce background check to the employment process*
- Could exist in multiple versions and variations
 - **Example:** *Policy underwriting process might change with passing of new legislation. Need ability for the process to exist as of certain time frame*
- Parts of Business process could be suspended or temporarily unavailable
 - **Example:** *Account transfer system is down while account status system is up*

What about Software itself?

- Deployment structure is monolithic
 - EAR, WAR – requires redeployment of the entire unit
- Single class path dependency
 - Outside of custom class loaders, **class-path** is the only dependency resolution mechanism available
- Software versioning concept is not present at runtime
 - Once JAR is in the class-path, its version is irrelevant

Proprietary solutions

- Heavy dependency on the structure dictated by current application platforms (i.e., JEE) and vendors that supply them
 - Foot-prints of such platforms most of the time larger than application requires
 - All-or-none platforms – modifications to application platforms are limited at best

Why OSGi? (problem statement)



Many vendors provide custom workarounds in the form of proprietary libraries and extensions to address some of these issues which results in a classic vendor lock-in

- What is OSGi?
- **Why OSGi?**
 - Describe the problem
 - **Describe the solution**
- OSGi Quick Start
- Intro to OSGi Constraint resolution

OSGi provides a set of standards around most of the issues described in the problem statement

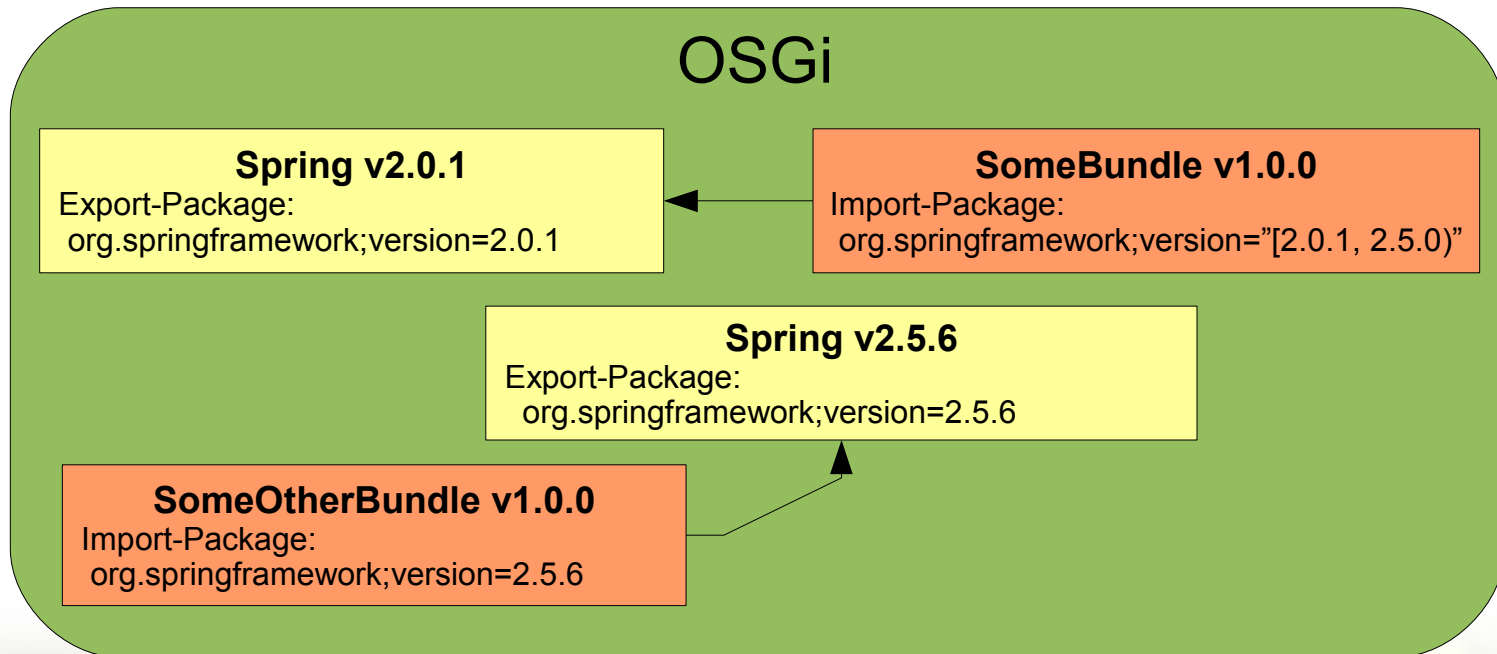
- Reduced software and software integration complexity
- OSGi dynamics suited well to address dynamics of real business processes
- Core OSGi platform is light-weight, but most importantly expandable
- Security is defined to provide infrastructure to build secured OSGi based systems

Software and Software integration complexity

- Internal complexity is encapsulated by OSGi Bundles
- Bundle collaboration with other bundles follows strict dependency resolution mechanism
- Versioning concept is present at runtime, allowing multiple version of the same libraries to coexist without creating a conflict
- Granular constraint resolution process (versions, filters, custom attributes)

Why OSGi? (solution)

Strict visibility rules based on explicit declarations (Import/Export-Package) and versioning



OSGi Services provide ideal environment to model and implement real business processes

- OSGi Services – integration points exposed by the bundles
- OSGi Services – dynamic (start, stop, update etc...)
- OSGi provides event infrastructure to handle service dynamics
- Eager or lazy introduction of services (passive/active dynamics)

Core OSGi platform is light-weight and small

- Core API consists of 1 package with less than 30 classes, while providing infrastructure to address:
 - Module life-cycle management
 - Dependency management and constraint resolution
 - Framework Event infrastructure
 - Security

OSGi core (kernel) has a very small foot-print

- Each deployed component (bundle) constitutes an extension to the platform, allowing for creation of custom platforms capable of addressing specific need

Example:

Tomcat and Servlet-API bundles deployed on OSGi platform will result in a custom platform capable of serving web requests

OSGi success stories:

- **Eclipse IDE** – OSGi based extensible Java IDE
 - More bundles (plug-ins) = more functionality
- **SpringSource dm Server** – OSGi based Spring Enterprise Java Platform

Exposing Operational controls via API and various tools allows to:

- See all modules and their status
- Retrieve wiring information
- Install/uninstall, activate/deactivate bundles
- Start/Stop bundles
- Update and Refresh bundles

All without stopping or restarting the platform

Why OSGi? (solution)



Popular OSGi implementations:

- Eclipse Equinox
- Apache Felix
- Makewave Knopflerfish

Topics in this Session



- What is OSGi?
- Why OSGi?
 - Describe the problem
 - Describe the solution
- **OSGi Quick Start**
- Intro to OSGi Constraint resolution

"Hello OSGi" - Bundle

DEMO

- **Eclipse PDE** – ideal environment to get started with OSGi
- Other tools:
 - **SpringSource Tool Suite**
 - **Maven** integration using maven-bundle-plugin

Topics in this Session



- What is OSGi?
- Why OSGi?
 - Describe the problem
 - Describe the solution
- OSGi Quick Start
- **Intro to OSGi Constraint resolution**

- MANIFEST.MF – describes underlying OSGi bundle
 - Contains OSGi headers describing bundle and its constraint resolution/resource sharing behavior
- Common MANIFEST headers:
 - Bundle-SymbolicName
 - Bundle-Name
 - Bundle-Activator
 - Bundle-Vendor
 - Require-Bundle

Dependency resolution MANIFEST headers:

- **Export-Package**
 - declares types advertised by this bundle
- **Import-Package**
 - declares types required by this bundle
- **Require-Bundle**
 - specifies dependency on all packages advertised by a given bundle (don't use unless only option)
- **Import/Export-Package**
 - may specify rules further addressing granularity for dependency resolution (i.e., "uses")

Constraint Resolution and Resource sharing:

- Bundle defines its **version** and **version** of its packages using **Bundle-Version** and **Export-Package** MANIFEST headers
- Allows multiple versions of the same bundle to co-exist in a single OSGi system

```
1Manifest-Version: 1.0
2Bundle-ManifestVersion: 2
3Bundle-Name: Transportation Calculator Bundle
4Bundle-SymbolicName: osgi.common.transportation.calculator
5Bundle-Version: 2.0.0
6Bundle-RequiredExecutionEnvironment: J2SE-1.5
7Export-Package: osgi.transportation.calculator;version="2.0.0"
8
```

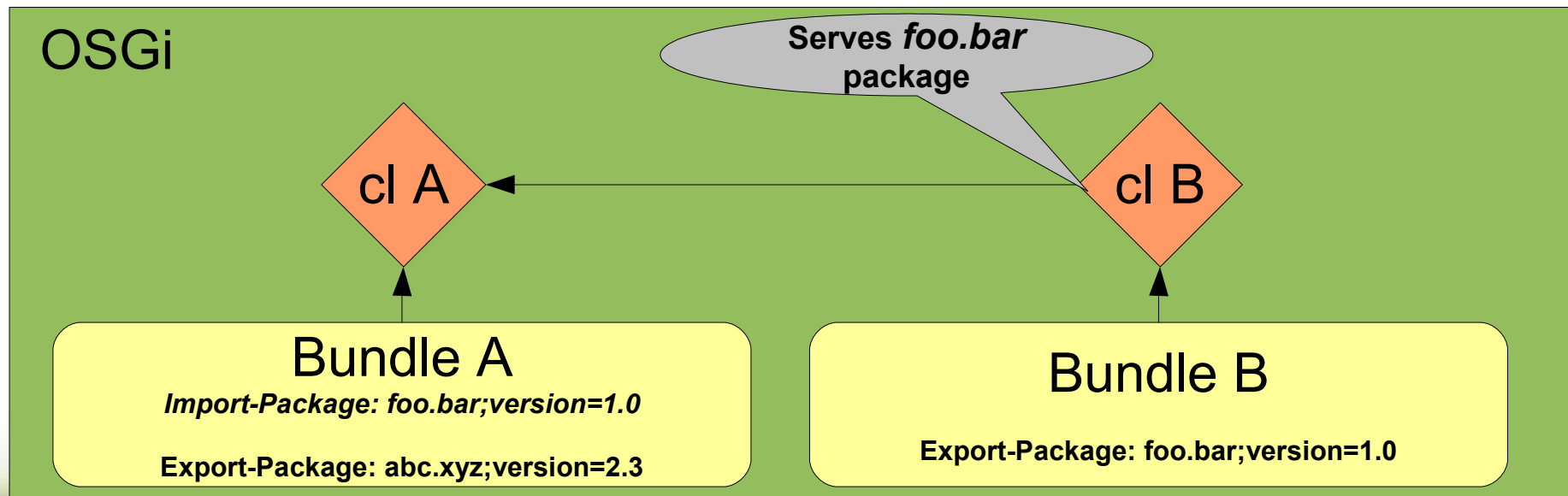
Constraint Resolution and Resource sharing:

- Bundles can specify dependencies on other bundles by using **Import-Package** or **Require-Bundle** MANIFEST headers
- May specify **version** of a dependent package or bundle

```
1Manifest-Version: 1.0
2Bundle-ManifestVersion: 2
3Bundle-Name: Version Plug-in
4Bundle-SymbolicName: osgi.dependency.version.solution
5Bundle-Version: 1.0.0
6Bundle-Activator: osgi.dependency.version.solution.VersionActivator
7Bundle-ActivationPolicy: lazy
8Import-Package: org.osgi.framework;version="1.3.0",
9 org.osgi.service.packageadmin,
10 osgi.common,
11 osgi.transportation.calculator
12Export-Package: osgi.dependency.version.solution;version="1.0.0"
13Bundle-RequiredExecutionEnvironment: J2SE-1.5
```

Constraint Resolution and Resource sharing:

- Bundles can specify dependencies on other bundles by using **Import-Package** or **Require-Bundle** MANIFEST headers
- May specify **version** of a dependent package or bundle



Version and Version Ranges

- Version syntax: *major.minor.micro.qualifier*
 - major ::= number
 - minor ::= number
 - micro ::= number
 - qualifier ::= text (optional)
- Single version means “**at least**”

Example:

- **Import-Package:** org.foo;version=1.0.0 – will resolve successfully to **Export-Package:** org.foo;version=2.0.0
- **Import-Package:** abc.xyz;version=2.0.0 – will NOT resolve successfully to **Export-Package:** abc.xyz;version=1.0.0

Version and Version Ranges

- Version ranges
 - Inclusive: [version, version]
 - Exclusive: (version, version)

Example:

- **Import-Package:** org.foo;version="[1.0.0, 2.0.0]" – will resolve successfully to **Export-Package:** org.foo;version=2.0.0
- **Import-Package:** org.foo;version="[1.0.0, 2.0.0)" – will NOT resolve successfully to **Export-Package:** org.foo;version=2.0.0
- **Import-Package:** org.foo;version="[1.0.0, 1.0.0]" – will NOT resolve successfully to **Export-Package:** org.foo;version=2.0.0

- Inside the OSGi Service Platform, bundle life is dynamic
- Bundles may be:
 - Installed / Uninstalled
 - Updated
 - Started/Stopped
- Life-cycle layer manages a Bundle's states

BundleActivator – life-cycle listener allowing you to hook custom processes into life-cycle events of the OSGi framework (e.g., start any necessary threads)

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
    public void start(BundleContext context) throws Exception {
        System.out.println("Starting Bundle");
    }
    public void stop(BundleContext context) throws Exception {
        System.out.println("Stopping bundle");
    }
}
```

- **BundleContext**

- Represents the execution context of a Bundle and acts as a proxy to the underlying framework
- Used to:
 - Install/Uninstall Bundles programmatically
 - Interrogate other bundles in the Framework
 - Retrieve, register/unregister OSGi services
 - Subscribe/unsubscribe to Service/Framework events

- OSGi introduces highly modular environment
 - allowing for more flexibility while addressing software and software integration complexity
- OSGi defines strict dependency resolution mechanism
 - using explicit declarations such as Import/Export-Package, version and others
- Unique class loading model
 - single class loader per bundle
- Bundle-Activator allows bundles to participate in OSGi's life-cycle mechanism

Lab

OSGi-intro

OSGi-1

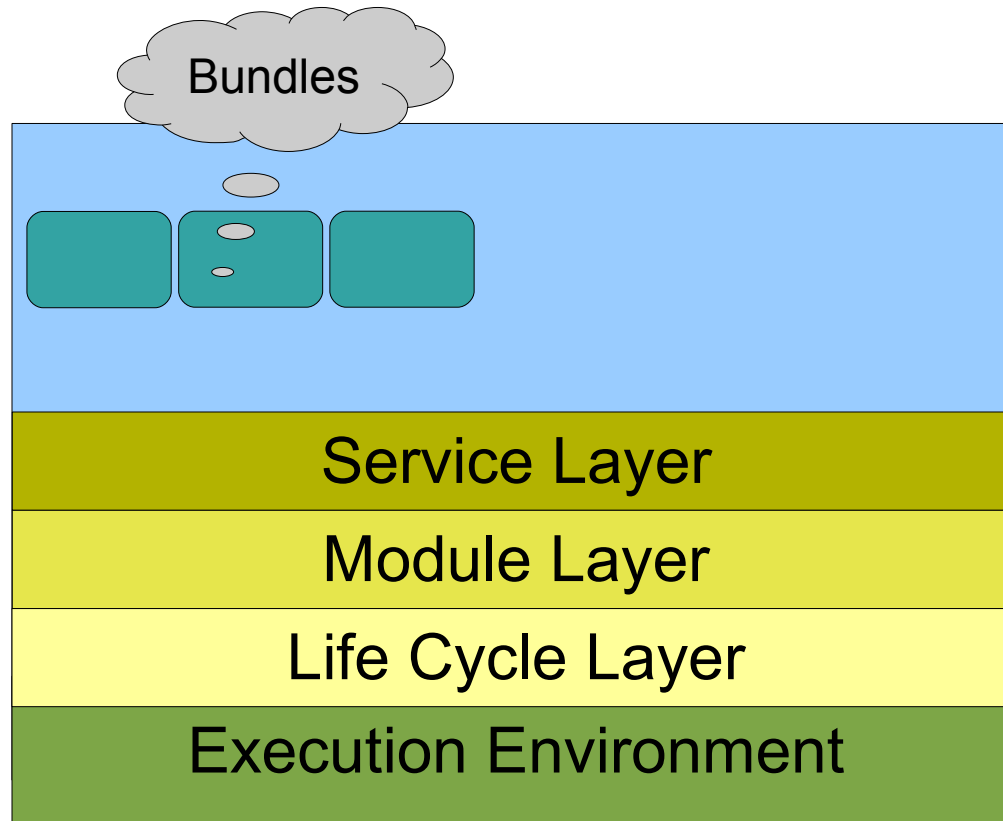
Intro to OSGi Architecture, Package Constraints
&
OSGi Services

Topics in this Session



- OSGi Layers
 - Module Layer
 - “USES” directive
 - Life-cycle Layer
 - Service Layer
 - Security Layer

OSGi Framework



Topics in this Session



- OSGi Layers
 - **Module Layer**
 - “USES” directive
 - Life-cycle Layer
 - Service Layer
 - Security Layer

- Defines Class sharing policies
 - No more single linear class-path
 - Rigidly specified class-loading model
 - Ads notion of Private classes and packages
 - If package is not exported its contents is not visible
- Introduces simple yet powerful Packaging and Deployment structure
 - In OSGi everything is a Bundle which means everything is a JAR

- Managed by declaring MANIFEST headers
 - **Export-Package** - declares types advertised by bundle
 - **Import-Package** - declares types required by bundle
- Defines notion of Package Constraint
 - **Import/Export-Package** - may optionally specify rules to address further granularity for dependency resolution (i.e., “**uses**”)

Topics in this Session



- OSGi Layers
 - **Module Layer**
 - **“USES” directive (Package Constraint)**
 - Life-cycle Layer
 - Service Layer
 - Security Layer

Package Constraint - “uses”



- Allows packages to specify strict dependency on another package
- Ensures single exporter for the package/class
- Preserves Class Space consistency
- Eliminates class loading exceptions
 - LinkageError, NoClassDefFoundError, ClassCastException, etc.

Package Constraint - "uses"

Problem

org.bar-v1.0.0

Export-Package: org.bar;version="1.0.0"

abc.xyz-v1.0.0

Import-Package: org.bar;version="[1.0.0,1.0.0]"

Export-Package: abc.xyz;version="1.0.0"

org.bar-v2.0

Export-Package: org.bar;version="2.0.0"

foo.bar-v1.0.0

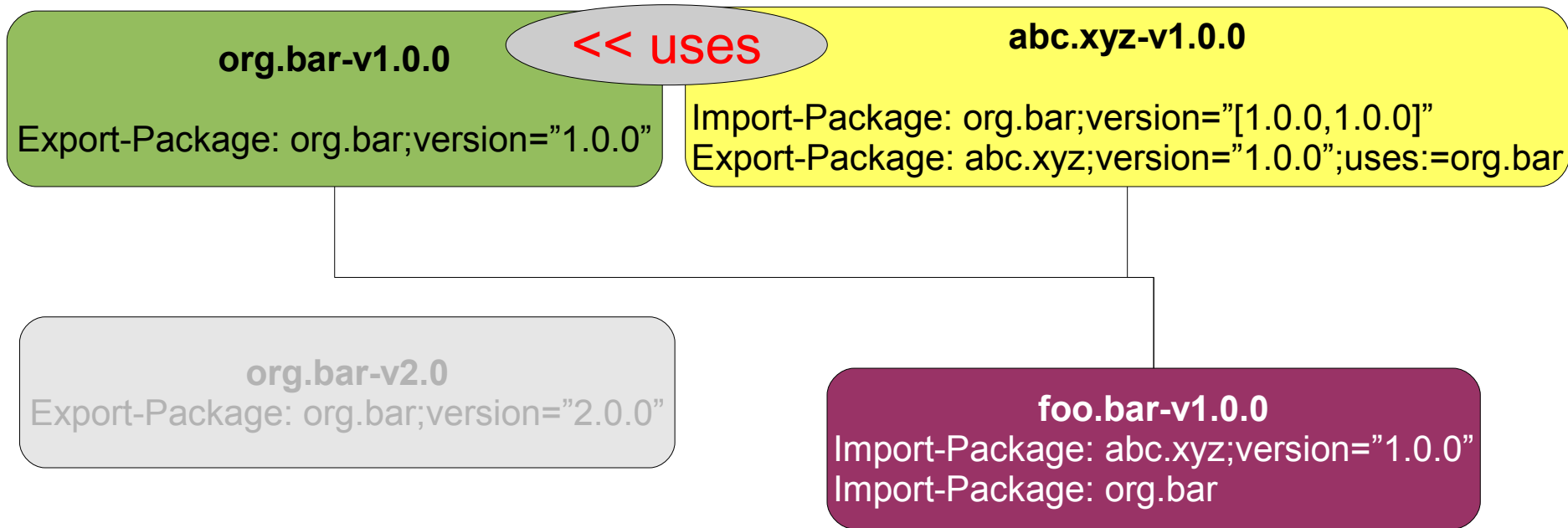
Import-Package: abc.xyz;version="1.0.0"

Import-Package: org.bar

Will result in LinkageError

Package Constraint - "uses"

Solution



Export-Package:
abc.xyz;version="1.0.0";**uses:=org.bar**

- OSGi Layers
 - Module Layer
 - “USES” directive (Package Constraint)
 - **Life-cycle Layer**
 - Service Layer
 - Security Layer

- Adds Bundle dynamics and manages Bundle states
- Allows bundles to be installed, started, stopped, updated and uninstalled
- Relies on Module Layer for class loading/sharing
- Ensures correct operation of the OSGi environment

Bundle States:

- **INSTALLED** – has been installed (deployed)
- **RESOLVED** – all required classes available. Ready to be started or stopped
- **STARTING** – being started. `BundleActivator.start(..)` method has been called, but did not yet return
- **ACTIVE** – is active and ready to be used
- **STOPPING** – being stopped. `BundleActivator.stop(..)` method has been called, but did not yet return
- **UNINSTALLED** - being un-installed, can no longer transition to another state

Active Bundle seen in Equinox console

```
osgi> ss
```

Framework is launched.

id	State	Bundle
0	ACTIVE	org.eclipse.osgi_3.4.2
1	ACTIVE	OSGiSample_1.0.0

```
osgi>
```

Life-cycle layer defines the following entities:

- **Bundle Context** – Bundle's execution context
- **Bundle Activator** – Interface implemented by developer and invoked by the framework, allowing bundle to participate in framework life cycle
- **Bundle Event** – Event that signals life cycle operation on a bundle
- **Bundle/Framework Listener** – Listeners to bundle/framework life-cycle events
- **System Bundle** – Bundle representing the Framework

Bundle Context:

- Represents the execution context of a Bundle
- Acts as a proxy to the underlying framework
- Used to:
 - Install/Uninstall Bundles programmatically
 - Interrogate other bundles in the Framework
 - Retrieve, register/unregister OSGi services
 - Subscribe/unsubscribe to Service/Framework events

Bundle Events and Bundle States

DEMO

- OSGi Layers
 - Module Layer
 - “USES” directive (Package Constraint)
 - Life-cycle Layer
 - **Service Layer**
 - Security Layer

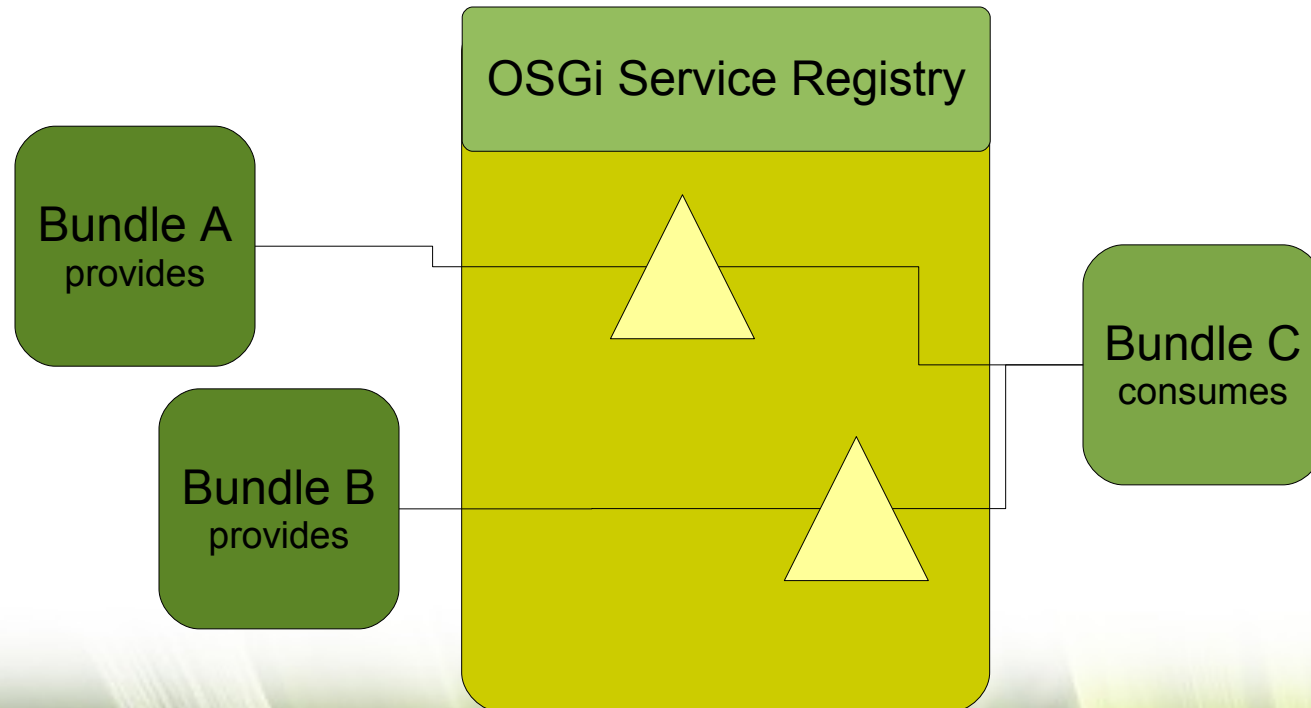
- Service Layer manages integration points identified as OSGi services and exposed by a Bundle
- Works closely with Module Layer
- Defined as Publish/Find/Bind model
 - **Collaborative** – provides mechanism for bundles to publish, find and bind to each service
 - **Dynamic** – addresses dynamic changes to life cycle of services (i.e., start, stop etc...)
 - **Versioned** – addresses evolution of bundles and services

OSGi Services:

- Identified by the following core entities:
 - Service Provider
 - Service Consumer
 - Service Registry
- Defined as POJO
- Advertise one or more POJI
- Registered with OSGi Service Registry
- Interaction with OSGi Services via BundleContext
- Packaged in OSGi Bundles

OSGi Services

- Provider Bundles – register OSGi Services
- Consumer Bundles – consume OSGi Services



ServiceReference

- Encapsulates meta-information about the service it represents
- Avoids dynamic dependency on the actual service object (similar to bean definition representing a bean)
- Returned when Framework is queried for a particular service
 - `BundleContext.getServiceReference(String name)`
 - `ServiceRegistration.getReference()`
- Then used to bind to the actual service object
- Valid for as long as the service object is registered

Publish Service

```
Bar serviceInstance = new Bar();  
ServiceRegistration reg =  
    bundleContext.registerService(Bar.class.getName(),  
                                   serviceInstance, null);  
...  
//ServiceReference ref = reg.getReference();  
//reg.unregister();
```

Find Service

```
ServiceReference ref =  
    bundleContext.getServiceReference(Bar.class.getName());
```

Service version can be passed as a filter expression

Bind to Service

```
Bar bar = (Bar) bundleContext.getService(ref);  
...  
bundleContext.ungetService(ref);  
// “bar” should no longer be used here
```

- Service Properties
 - Provide information about the service
 - Can be used in conjunction with Filter interface to uniquely identify a service
- Service Filters
 - Allow to limit the visibility of a service based on its meta data during service discovery
 - Syntax is based on LDAP search filters
 - (&(key1=someValue)(key2=someOtherValue))
 - Created by `BundleContext.createFilter(String)` method - where String represents filter expression

- Register Service with Property

```
Bar bar = new Bar();  
Dictionary filters = new Hashtable();  
filters.put("property.name", "Hello OSGi");  
bundleContext.registerService(Bar.class.getName(),  
                               bar, filters);
```

- Search for services with Filters

```
ServiceReference[] sr =  
    bundleContext.getServiceReferences(Bar.class.getName(),  
                                       "(property.name=Hello OSGi)");  
... 
```


- OSGi components can also get Services info by participating in OSGi Event infrastructure
- Individual bundles can register service listener to subscribe to service events
 - **ServiceEvent** - reports various changes to the Service (i.e., registering, unregistering, property changes etc.)
 - **ServiceListener** - notified synchronously about occurrence of ServiceEvent(s)
- Since OSGi services are dynamic, listening for service events is important to be aware of changes to individual services

- Creating and Registering ServiceListener

```
...
bundleContext.addServiceListener(new ServiceListener(){
    public void serviceChanged(ServiceEvent serviceEvent) {
        switch (serviceEvent.getType()) {
            case ServiceEvent.REGISTERED:
                break;
            case ServiceEvent.UNREGISTERING:
                break;
            ....
        }
    }
});
```

Service Dynamics

- Individual services owned and managed by registering bundle
- When bundle is stopped, its services are automatically unregistered
- Presents issues, such as [Stale References](#) - a reference to an unregistered service or a stopped bundle
 - Could affect garbage collection and create memory leaks
 - Proper usage of OSGi notification mechanism and Service Tracker will allow avoiding these issues

- OSGi Layers
 - Module Layer
 - “USES” directive (Package Constraint)
 - Life-cycle Layer
 - Service Layer
 - **Security Layer**

- Optional layer of OSGi platform
- Based on Java 2 security architecture
 - Uses SecurityManager

Start OSGi platform by providing:

```
-Djava.security.manager  
-Djava.security.policy=policyFileURL
```

Permissions are specified in the policy file:

```
grant codeBase "file:bundle/shellplugin.jar" {  
    permission java.security.AllPermission;  
};
```

- OSGi Service Platform is based on layered Architecture
- OSGi allows for more granular approach when it comes to resource sharing and dependency/constraint resolution
- OSGi Services are dynamic (can come and go at any time)
 - EventListener infrastructure provided to address dynamics and potential stale references
 - Service Properties and Filters can further define criteria for matching services

Lab

OSGi-1

Spring-DM Intro

Introduction to Spring Dynamic Modules
for OSGi Service Platforms

Topics in this Session

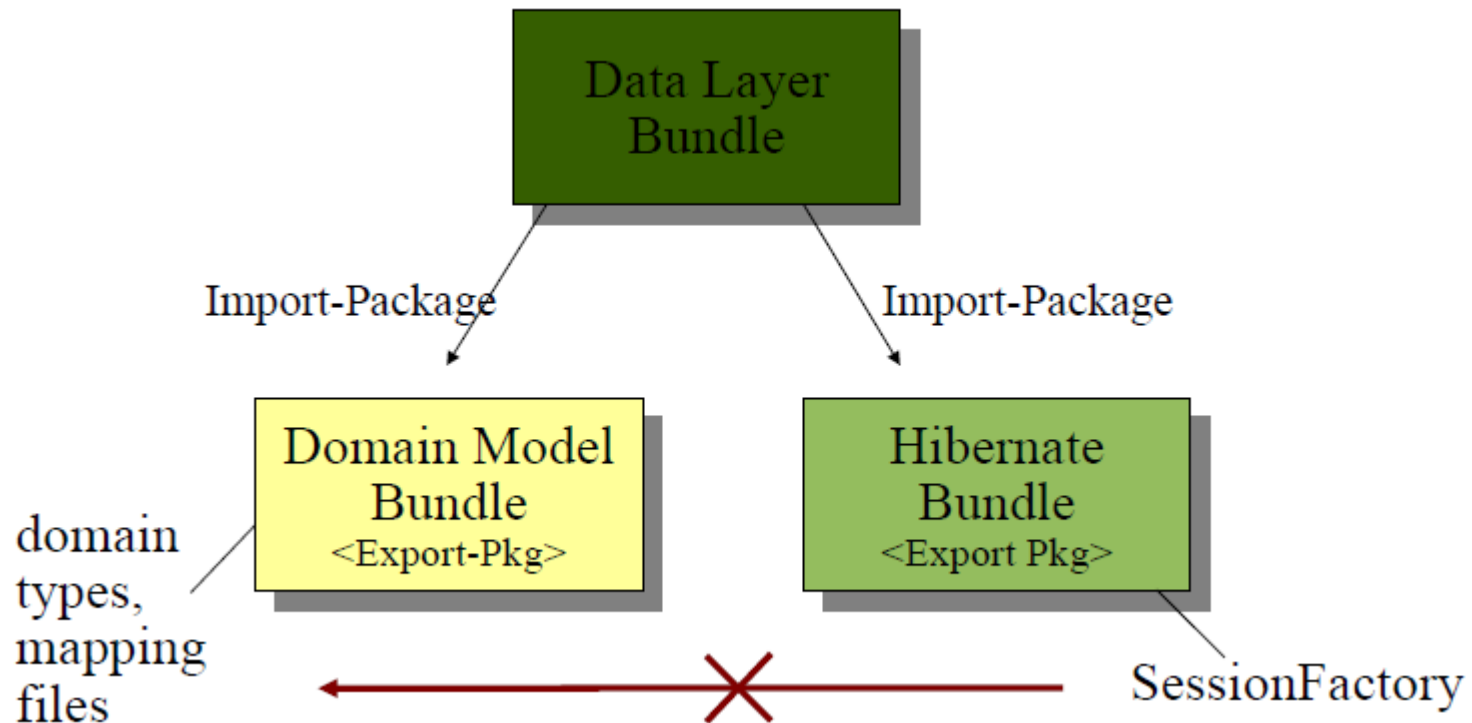


- Why Spring-DM?
 - OSGi Challenges
 - Spring-DM objectives
- Bundles and Application Context
- Extender Bundle
- Packaging and Deploying
- Service Dynamics
- RFC-124

- Service dynamics (come and go at any time)
 - Requires a lot of plumbing code to manage
 - OSGi code gets mixed up with business code (concern/code tangling)
- OSGi Integration Testing
- Concurrency and thread management
- Class and Resource loading issues
 - Class visibility
 - Context Class Loader
 - Resources in META-INF
 - Taglibs

Why Spring-DM?

- Class Visibility issues



- **Why Spring-DM?**
 - OSGi Challenges
 - **Spring-DM objectives**
- Bundles and Application Context
- Extender Bundle
- Packaging and Deploying
- Service Dynamics
- RFC-124

- Bring benefits of Spring to OSGi
- Address some of the OSGi challenges
 - Integration Testing
 - Web Support
 - Verboseness and plumbing of OSGi
 - Bring the best of Spring to OSGi
 - Define transparency between Application Context and OSGi Bundle
 - Use Spring for Configuration

Spring Dynamic Modules



- Open source project in the Spring portfolio
- Lead by SpringSource
- Includes committers from Oracle and BEA (Oracle)
- Many community contributors

Topics in this Session



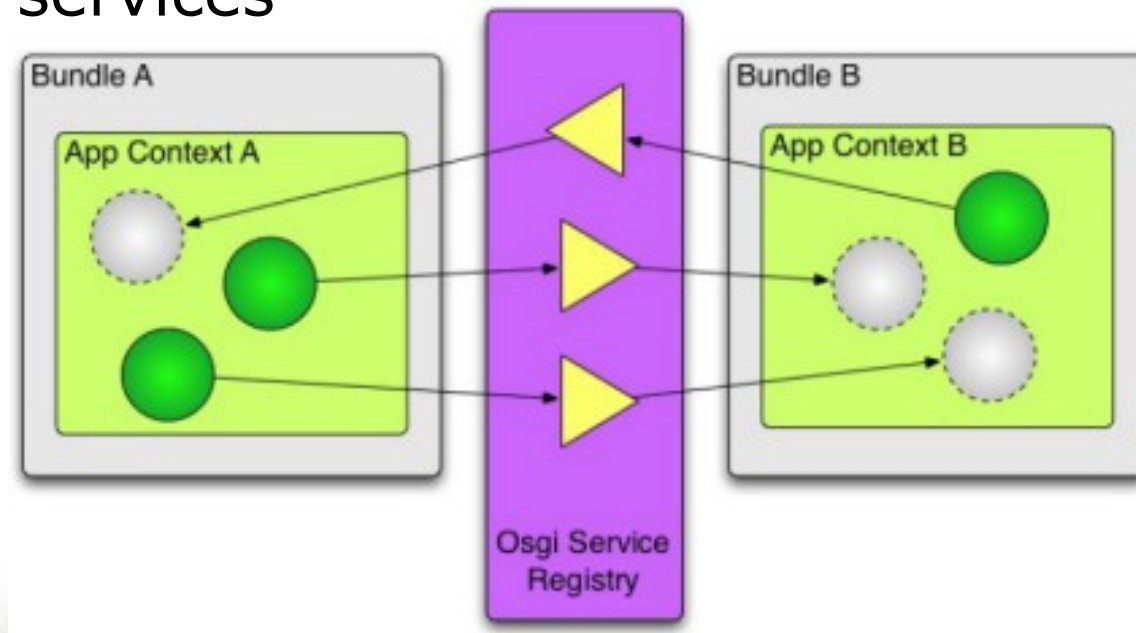
- Why Spring-DM?
 - OSGi Challenges
 - Spring-DM objectives
- **Bundles and Application Context**
- Extender Bundle
- Packaging and Deploying
- Service Dynamics
- RFC-124

Bundles and Application Context



- Spring Application Context (module context) per bundle (module)
- Created automatically for you by Spring
- Bootstrapped by the Extender bundle
- No need to depend on any OSGi APIs

- Application constructed as a set of bundles
 - each with their own module context
- Spring Beans are published and consumed as OSGi services



Topics in this Session



- Why Spring-DM?
 - OSGi Challenges
 - Spring-DM objectives
- Bundles and Application Context
- **Extender Bundle**
- Packaging and Deploying
- Service Dynamics
- RFC-124

- Spring-DM provides special OSGi Bundle to bootstrap Spring Application Context
 - org.springframework.osgi.bundle.extender
 - like ContextLoaderListener in Spring-based web apps
- Looks for Spring-powered Bundles in ACTIVE state and creates ApplicationContext
 - Uses [META-INF/spring/*.xml files](#) by default
 - [Spring-Context](#) header in [MANIFEST.MF](#) to override
- Listens for Bundle starting events and checks if bundle is Spring-powered

- Asynchronous by default
 - Ensures quick startup
 - No deadlock for bundles with inter-dependencies
- Bundle transitions to ACTIVE state before its Application Context is created
- Synchronous creation is possible
 - using Spring-DM headers in MANIFEST.MF
 - `Spring-Context: *;create-asynchronously:=false`

- Typically no need for OSGi APIs with Spring-DM
 - Allows proper Separation of Concerns
- Implement BundleContextAware to access the OSGi BundleContext object in your Spring beans
 - Requires ApplicationContext to be Spring-DM powered

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

Topics in this Session



- Why Spring-DM?
 - OSGi Challenges
 - Spring-DM objectives
- Bundles and Application Context
- Extender Bundle
- **Packaging and Deploying**
- Service Dynamics
- RFC-124

- Define Application Context configuration files in predefined location ([META-INF/spring](#))
- OR*
- Specify [Spring-Context:](#) header in MANIFEST.MF pointing to the location of Application Context configuration files
 - Spring-DM will take care of the rest

- Any Spring Bean can be exported as an OSGi Service!
 - Define your bean as usual
 - Use **osgi:service** to export it as an OSGi Service
 - Provide the interface(s) for the service

```
<beans>
  <bean id="accountService" class="foo.bar.AccountServiceImpl">
    <constructor-arg ref="accountRepository" />
  </bean>

  <osgi:service ref="accountService" interface="foo.bar.AccountService"/>

</beans>
```


- Import OSGi services as Spring beans
 - Use **osgi:reference** to import OSGi Service and bind it to a proxy bean
 - Can be dependency injected like any other bean
 - Spring-DM locates the best matching service
 - Spring-DM handles Service dynamics

```
<beans>
  <osgi:reference id="accountService" interface="foo.bar.AccountService"/>

  <bean id="accountManager" class="foo.bar.AccountManager">
    <constructor-arg ref="accountService" />
  </bean>
</beans>
```

Topics in this Session



- Why Spring-DM?
 - OSGi Challenges
 - Spring-DM objectives
- Bundles and Application Context
- Extender Bundle
- Packaging and Deploying
- **Service Dynamics**
- RFC-124

- Application Context creation blocks until all *mandatory* Service References are satisfied
 - Use “**timeout**” attribute of the reference element to specify how long to wait for Service Registration
 - Defaults to 5 minutes
 - If exported service goes away, dependent service will be unregistered
- Bean created immediately for *optional* reference
 - regardless of whether or not there is currently a matching service

What happens when a service goes away?

- `<osgi:reference cardinality="0..1" .../>`
 - Track replacement and retarget proxy when suitable target found
 - `ServiceUnavailableException` after timeout if invoked
- `<osgi:reference cardinality="1..1" .../>`
 - Same as above, plus:
 - Unregister any exported services that depend on the unsatisfied reference

What happens when a service goes away?

- `<osgi:set/list cardinality="0..n" .../>`
 - Service is removed from the set
 - Iterator contract is honored
- `<osgi:set/list cardinality="1..n" .../>`
 - Same as above, plus:
 - Unregister any exported services that depend on the unsatisfied reference

Service Listeners:

- You work with constant reference
 - Proxy, Set or List
- Spring-DM manages backing Service(s) for you
- *Consumers* can listen to *bind/unbind* events
 - When proxy-backing service comes or goes
- *Publishers* can listen to *register/unregister* events
 - When the service is (un)registered
 - e.g. because its dependencies are no longer satisfied

Bind/Unbind – reference listener:

- Use `osgi:listener` element
- Define listener as POJO, specifying bind-/unbind-method attributes

```
<beans>

<osgi:reference id="accountService" interface="foo.bar.AccountService">
  <osgi:listener bind-method="onBind" unbind-method="onUnbind">
    <bean class="foo.bar.AccountServiceListener"/>
  </osgi:listener>
</osgi:reference>

</beans>
```

Bind/Unbind – reference listener (cont. . .)

- Service Listener implementation is just a POJO

```
public class AccountServiceListener {  
  
    public void onBind(AccountService as, Map serviceProperties) { . . . }  
  
    public void onUnbind(AccountService as, Map serviceProperties) { . . . }  
  
}
```


Registration listener:

- Use `osgi:registration-listener` element
- Define listener as POJO, specifying registration-/unregistration-method attributes

```
<beans>

<osgi:service ref="accountService" interface="foo.bar.AccountService">
  <osgi:registration-listener registration-method="register"
                             unregistration-method="unregister">
    <bean class="foo.bar.AccountServiceRegistrationListener"/>
  </osgi:registration-listener>
</osgi:service>

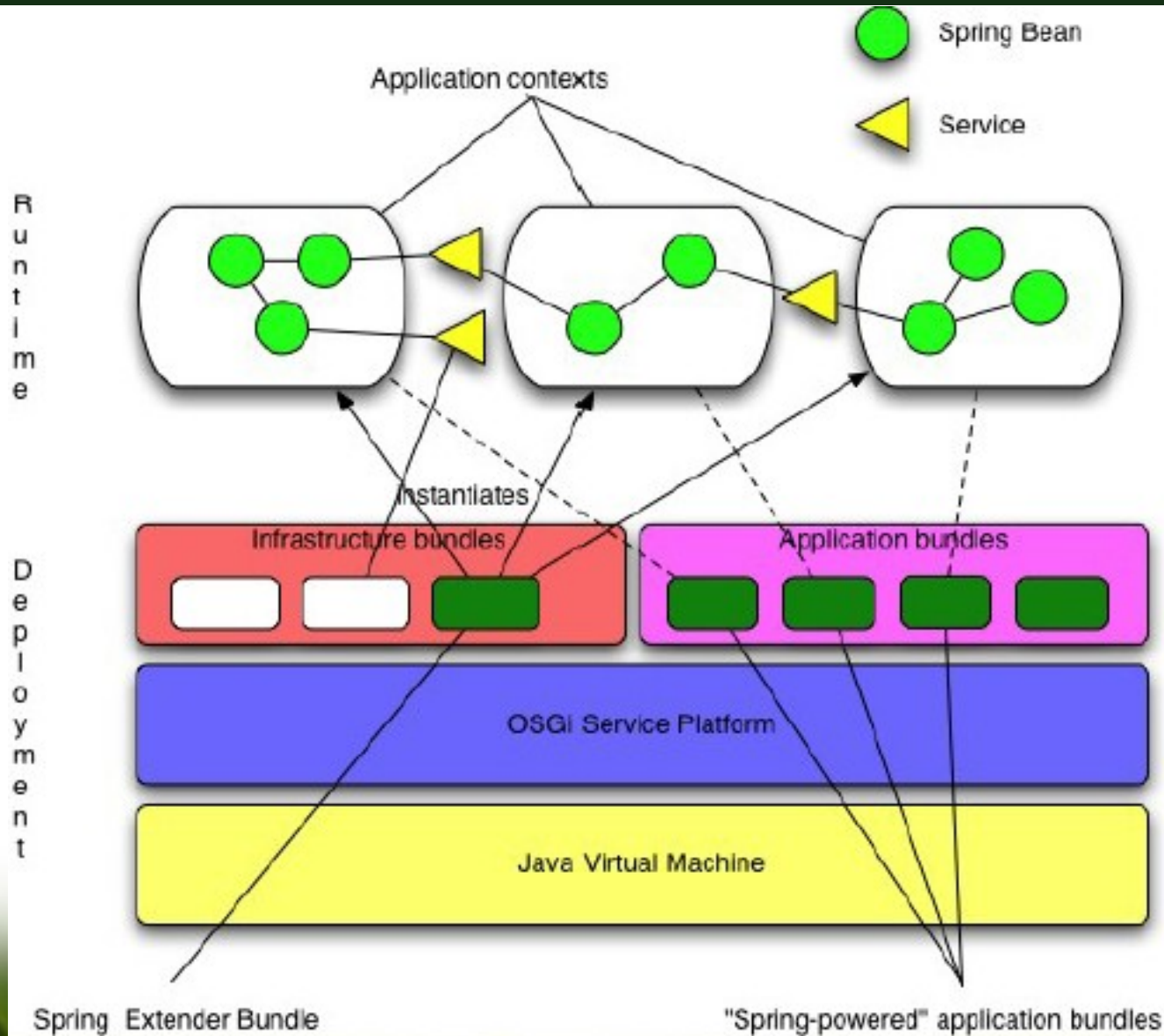
</beans>
```

Registration listener (cont. . .)

- Registration Listener implementation is just a POJO

```
public class AccountServiceRegistrationListener {  
  
    public void register(AccountService as, Map serviceProperties) { . . . }  
  
    public void unregister(AccountService as, Map serviceProperties) { . . . }  
  
}
```

Spring Dynamic Modules



Topics in this Session



- Why Spring-DM?
 - OSGi Challenges
 - Spring-DM objectives
- Bundles and Application Context
- Extender Bundle
- Packaging and Deploying
- Service Dynamics
- **RFC-124**

- Standardization effort lead by SpringSource around OSGi component model
 - Started in 2006
 - Identifies close relationship between component model of Spring Framework and modularity/versioning of OSGi
 - Resulted in inception of Spring Dynamic Modules project
 - Key contributors from SpringSource, Oracle, BEA Apache Felix, Knopflerfish etc. . .

Why another component model?

- Current OSGi specification is lacking the following areas
 - No component model (i.e., <bean>)
 - Configuration and assembly support is very basic comparative to DI frameworks such as Spring
 - Dependency on OSGi API at the component level which violates separation of concerns
- Dynamics addressed by Spring-DM

- Spring-DM based component:
`<bean id="account" class="org.bar.Account">`
`<osgi:service ref="account" interface="org.bar.AccountStrategy"/>`
- Standards-based definition of a component:
`<component id="account" class="org.bar.Account">`
`<service ref="account" interface="org.bar.AccountStrategy"/>`

- RFC 124 – based on learning experiences of Spring-Dynamic Modules and contributions of other members of Enterprise Expert Group within OSGi Alliance
- Standardizes Spring-DM concepts in OSGi R4.2
- Spring-DM version 2 will be RI of RFC-124
 - Once specification is final

- Spring-DM brings the familiar Spring model to the OSGi platform
- Allows you to develop OSGi based applications without writing a single line of OSGi code
- RFC-124 – standardization effort around component model of Spring and OSGi

Lab

osgi-springdm-intro

Introduction to the SpringSource dm Server

A short overview of what's in the box

Topics in this Session

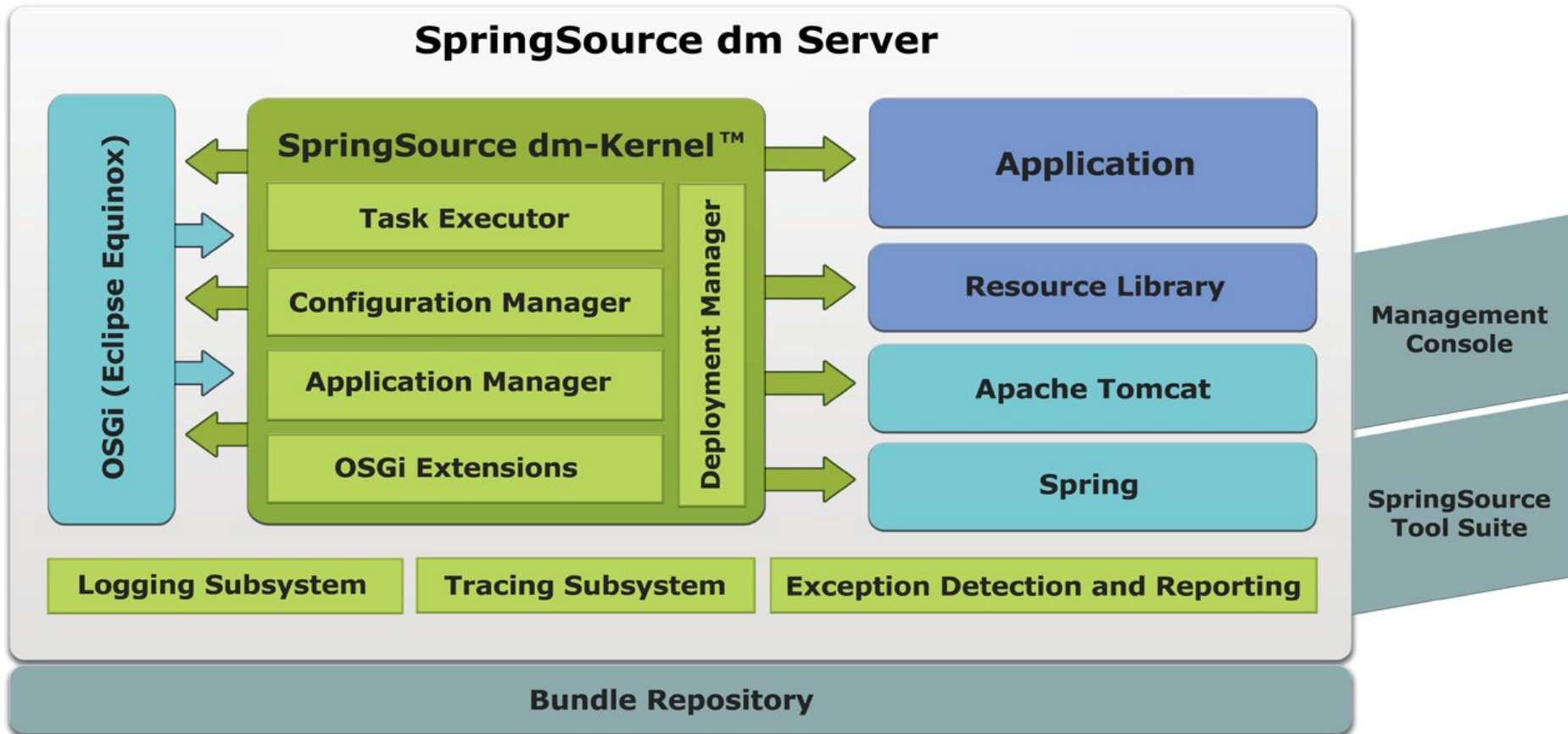


- **Overview**
- Features
- Support for web applications
- Installing the server
- Running the server
- Eclipse tooling

- OSGi-based application server
- Based on Tomcat, Equinox, Spring and Spring Dynamic Modules (Spring-DM)
- Requires Sun Java 5 or 6
- Allows you to build dynamic, modular enterprise Java applications that leverage OSGi

- Modular, lightweight architecture
 - Use just what you need
- OSGi not just 'under the covers' but exposed through the programming model
 - Using Spring-DM
 - Adds many features for 'enterprise OSGi'
- Not a full-blown Java EE Server
 - Similar to Tomcat 6 in web-application support
 - Considering Java EE 6 compliance for web profile

High-level Architecture



Topics in this Session



- Overview
- **Features**
- Support for web applications
- Installing the server
- Running the server
- Eclipse tooling

- Supports both WAR files and OSGi bundles
 - WARs can use OSGi
 - Also offers Web Module bundle format
 - Other personalities planned
- Advanced logging and tracing support
- First Failure Data Capture (FFDC)
 - Find the root cause of an error

- Loads your dependencies on demand
 - Provisioning system for OSGi bundles
- Use different versions of a library in a single application
- Dynamically update modules in your application
 - No full restart of the application needed

Ease the use of enterprise technologies in an OSGi environment:

- Deploy and manage related bundles as a single application
- Avoid endless Import-Package headers
- Support for load-time weaving
 - Needed for most JPA providers

Topics in this Session



- Overview
- Features
- **Support for web applications**
- Installing the server
- Running the server
- Eclipse tooling

- Supports existing Java EE 5 WAR files
 - Transformed into OSGi bundle during deployment
- Adds options for using OSGi in your WARs
 - To use bundles instead of jars in WEB-INF/lib
 - To use shared services from OSGi service registry
- Adds Web Module for Spring MVC
 - Bundle instead of WAR
 - Auto-deploys a DispatcherServlet for you

- Uses Tomcat 6 with custom OSGi integration
 - dm Server does NOT use Spring-DM's web support!
- Supports load balancing and session clustering
 - Exactly like Tomcat
- Does not yet support JNDI, custom security realms or admin-defined DataSources
 - Planned for later releases

Topics in this Session



- Overview
- Features
- Support for web applications
- **Installing the server**
- Running the server
- Eclipse tooling

Installing the dm Server



- Download the binary distribution
 - Or build your own from the GPL-ed source
- Extract to a directory
 - On Windows, avoid long path names
- Define JAVA_HOME environment variable
- That's it!

dm Server Directory Layout



- bin: scripts
- config: configuration files
- docs: documentation
- lib: libraries for internal use
- licenses: text of various licenses
- pickup: drop applications here to auto-deploy them
- repository: contains bundles and library definitions
- serviceability: holds log and trace files and generated dumps
- work: contains the expanded bundles

Topics in this Session



- Overview
- Features
- Support for web applications
- Installing the server
- **Running the server**
- Eclipse tooling

Running the server (1)



- Use provided startup script under 'bin' dir
- Optional parameters:
 - debug enable debugging
 - suspend wait for debugger to attach
 - clean clean work dir
 - configDir use different config directory
 - jmxport, -jmxusers, -keystore,
 - keystorePassword JMX connector configuration
- Secured remote JMX connector always started since 1.0.1
- Optionally provide port number for debug

Running the server (2)



- Shutdown scripts also provided
- Optional parameters:
 - immediate** force immediate shutdown
 - truststore**, like **keystore** at startup
 - truststorePassword**
- For production use, you'll probably use a Windows service or Unix init.d script
- Currently not included, but easy to roll your own
 - e.g. using Java Service Wrapper

Successful server start



```
C:\WINDOWS\system32\cmd.exe - startup.bat


C:\springsource-dm-server-1.0.1.RELEASE\bin>startup.bat
[2008-12-01 21:48:35.156] main          <SPKB0001I> Server starting.
[2008-12-01 21:48:36.218] main          <SPOF0001I> OSGi telnet console available on port 2401.
[2008-12-01 21:48:40.093] main          <SPKE0000I> Boot subsystems installed.
[2008-12-01 21:48:41.453] main          <SPKE0001I> Base subsystems installed.
[2008-12-01 21:48:43.296] server-dm-3 <SPPM0000I> Installing profile 'web'.
[2008-12-01 21:48:47.078] server-dm-3 <SPPM0001I> Installed profile 'web'.
[2008-12-01 21:48:47.156] server-dm-8 <SPSC0001I> Creating HTTP/1.1 connector with scheme http on port 8080
-
[2008-12-01 21:48:47.250] server-dm-8 <SPSC0001I> Creating HTTP/1.1 connector with scheme https on port 844
3.
[2008-12-01 21:48:47.281] server-dm-8 <SPSC0001I> Creating AJP/1.3 connector with scheme http on port 8009.
-
[2008-12-01 21:48:47.312] server-dm-8 <SPSC0000I> Starting ServletContainer.
[2008-12-01 21:48:48.687] server-dm-14 <SPPM0002I> Server open for business with profile 'web'.
[2008-12-01 21:48:48.718] fs-watcher <SPDE0048I> Processing 'INITIAL' event for file 'server.admin.web-1.0
1.0.1.RELEASE.jar'.
[2008-12-01 21:48:50.265] fs-watcher <SPSC1000I> Creating web application '/admin'.
[2008-12-01 21:48:50.828] async-delivery-thread-1 <SPSC1001I> Starting web application '/admin'.
[2008-12-01 21:48:51.578] fs-watcher <SPDE0010I> Deployment of 'com.springsource.server.servlet.admin' ver
sion '1.0.1.RELEASE' completed.
[2008-12-01 21:48:51.578] fs-watcher <SPDE0048I> Processing 'INITIAL' event for file 'server.admin.splash-
1.0.1.RELEASE.jar'.
[2008-12-01 21:48:52.375] fs-watcher <SPSC1000I> Creating web application '/'.
[2008-12-01 21:48:52.515] async-delivery-thread-1 <SPSC1001I> Starting web application '/'.
[2008-12-01 21:48:52.890] fs-watcher <SPDE0010I> Deployment of 'com.springsource.server.servlet.splash' ve
rsion '1.0.1.RELEASE' completed.
-
```

What's next?



- Connect to the web admin console
 - <http://localhost:8080/admin>
 - User: admin Password: springsource
- Deploy apps through pickup dir or console
 - WAR files
 - OSGi bundles
 - Platform archives
- Use the Equinox console
 - `telnet localhost 2401`

Web Admin Console

SpringSource dm Server™

Applications

Admin Console

Result of the last operation: 'Applications Listed'.

Deployed Applications

Name	Version	Origin	Date	Undeploy
com.springsource.server.servlet.admin	1.0.1.RELEASE	Hot Deployed	1-dec-2008 21:48:50 CET	undeploy
Associated Modules:				
com.springsource.server.servlet.admin	(type: Web)	/admin		
com.springsource.server.servlet.splash	1.0.1.RELEASE	Hot Deployed	1-dec-2008 21:48:52 CET	undeploy
Associated Modules:				
com.springsource.server.servlet.splash	(type: Web)	/		

Deploy an Application

Select an application or bundle to upload and deploy to the server. Valid file formats: *jar, war, par*.

Application Location

Bladeren...

Upload

Information

Server Properties

Name	Value
Default Time Zone	Europe/Berlin

- Currently only for deploy and undeploy
- Will be expanded in future releases

Equinox Telnet Console




```
osgi> help
---Controlling the OSGi framework---
    launch - start the OSGi Framework
    shutdown - shutdown the OSGi Framework
    close - shutdown and exit
    exit - exit immediately (System.exit)
    init - uninstall all bundles
    setprop <key>=<value> - set the OSGi property
---Controlling Bundles---
    install - install and optionally start bundle
    uninstall - uninstall the specified bundle(s)
    start - start the specified bundle(s)
    stop - stop the specified bundle(s)
    refresh - refresh the packages of the specified bundle(s)
    update - update the specified bundle(s)
---Displaying Status---
    status [-s [<comma separated list of bundle states>]] - display status of bundles
    ss [-s [<comma separated list of bundle states>]] - display status of bundles
    services [filter] - display registered service packages
    packages [<pkgname>|<id>|<location>] - display package details
    bundles [-s [<comma separated list of bundle states>]] - display bundle details
    bundle (<id>|<location>) - display details for bundle
    headers (<id>|<location>) - print bundle headers
    log (<id>|<location>) - display log entries
---Extras---
```

- Low-level tool
- Good for debugging problems with bundles or visibility issues
- Warning:
exit kills the server!
 - Use **disconnect**

Topics in this Session



- Overview
- Features
- Support for web applications
- Installing the server
- Running the server
- **Eclipse tooling**

- Free plugin provided for use with Eclipse
- Depends on Spring-IDE
- Already included in SpringSource Tool Suite
- Install through update site for plain Eclipse:
`http://static.springsource.com/projects/sts-dm-server/update/` 
– Local update site also available

Update: combined update site now also available:
`http://www.springsource.org/update/e3.4`

- Allows deployment to local server
- Dedicated editor for bundle manifests
 - Syntax checking & highlighting, code assist, etc.
- Manages project classpath based on manifest headers and target server
- Wizards for new dm Server projects
- Updates individual bundles of running application
 - Speeds up development: no full restarts anymore!

- SpringSource dm Server makes OSGi available to enterprise applications
- Supports WAR files and OSGi bundles
- Very lightweight
- Allows for truly dynamic and modular applications

Lab

dms.intro

Deploying to the SpringSource dm Server

The Various Deployment Options

Topics in this Session



- What and how can you deploy
- See what's happening
- Inspect the end result
- Controlling the context path

What You Can Deploy



- Java EE WAR files
 - May use some OSGi
- Plain OSGi bundles
 - We'll discuss OSGi in more detail in the next module
- PAR files
 - Platform Archive, basically a jar of bundles
 - We'll see on day 2

How You Can Deploy



- Web Admin Console
- pickup dir
- Eclipse tooling
- Deployer MBean
- Bundles only: repository/bundles/usr dir
 - But that's not really deploying as we'll see

- (Un)deploy of all supported formats
- Enables remote uploads
 - No local access to server or filesystem needed
- Ordering of deployed artifacts is preserved
 - On restart, will start in the same order
 - Important for dependency resolving
- Server must be running
 - Access through ***servername:8080/admin***
 - Default login/pwd: admin/springsource

Web Admin Console

SpringSource dm Server™



Applications

Admin Console

Result of the last operation: 'Application undeployed'.

Deployed Applications

Name	Version	Origin	Date	Undeploy
com.springsource.server.servlet.admin	1.0.0.RELEASE	Hot Deployed	15-okt-2008 9:03:44 CEST	undeploy
Associated Modules: com.springsource.server.servlet.admin (type: Web) /admin				
com.springsource.server.servlet.splash	0	Hot Deployed	15-okt-2008 9:03:47 CEST	undeploy
Associated Modules: com.springsource.server.servlet.splash (type: WAR) /				

Deploy an Application

Select an application or bundle to upload and deploy to the server. Valid file formats: *jar*, *war*, *par*.

Application Location

Information

How You Can Deploy



- Web Admin Console
- **pickup dir**
- Eclipse tooling
- Deployer MBean
- repository/bundles/usr dir

'pickup' Directory

- Drop file in the 'pickup' directory to deploy
- Requires access to local filesystem on server
- Ordering of deployed artifacts is preserved
 - On restart, will start in the same order
 - No ordering guarantees for sets of files:
deploy one at a time!
- Remove file from 'pickup' directory to undeploy
- Server doesn't have to be running

How You Can Deploy

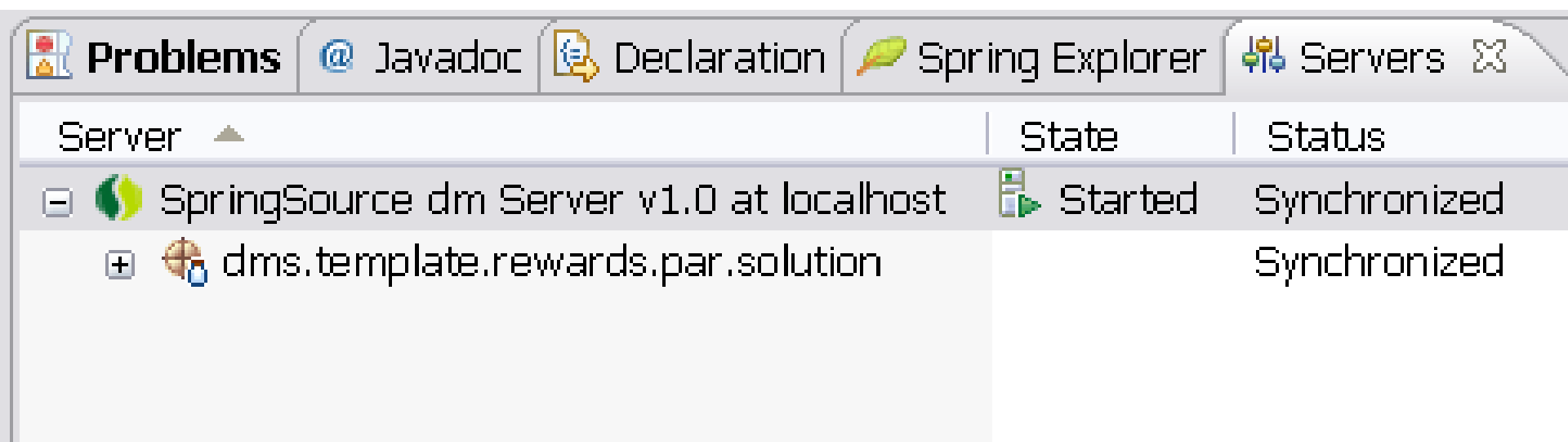


- Web Admin Console
- pickup dir
- **Eclipse tooling**
- Deployer MBean
- repository/bundles/usr dir

Eclipse Tooling



- (Un)deploy to local server instance
- Use "Add and Remove Projects", "Run on Server" or drag-and-drop
- Server doesn't have to be running
- Similar to WTP's Tomcat support



How You Can Deploy



- Web Admin Console
- pickup dir
- Eclipse tooling
- **Deployer MBean**
- repository/bundles/usr dir

- dm Server registers several JMX MBeans, incl. Deployer
- Mostly intended for internal use
- (Un)deploy artifacts from server's local filesystem
- Access locally or remotely through JSR-160 remote JMX connector
 - e.g. using JConsole

Deployer MBean in JConsole



Java Monitoring & Management Console - pid: 1620 com.springsource.server.kerne...

Connection Window Help

Overview Memory Threads Classes VM Summary MBeans

JMImplementation
com.springsource.server
Deployer
Operations
Notifications
ProvisioningRepository
RecoveryMonitor
Shutdown
SystemDumpControl
java.lang
java.util.logging
springsource.server.catalina

Operation invocation

void refresh (p1 String , p2 String)

CompositeData deploy (p1 String , p2 true)

CompositeData deploy (p1 String)

URI

void undeploy (p1 String , p2 String)

void updateResource (p1 app name , p2 app version , p3 String , p4 String)

void updateResource (p1 String , p2 String , p3 String , p4 String)

void deleteResource (p1 String , p2 String , p3 String , p4 String)

void deleteResource (p1 String , p2 String , p3 String)

How You Can Deploy



- Web Admin Console
- pickup dir
- Eclipse tooling
- Deployer MBean
- **repository/bundles/usr dir**

- Can also place bundles in repository/bundles/usr
- Not an actual deployment
 - Bundle won't be started automatically
- Only makes bundle available for *provisioning*
 - More about that later
- Typically used for 3rd party libraries
 - Have to be OSGi bundles!

Topics in this Session

- What and how can you deploy
- **See what's happening**
- Inspect the end result
- Controlling the context path

See what's happening

- dm Server writes important messages to console and to log file
 - serviceability/logs/logging.log
- Complete trace of events is also kept
 - serviceability/trace/trace.log
 - Can become huge
- These are important for troubleshooting

Sample: Successful Deploy



- The default applications are splash and web
- Output to console & log (timestamps removed):

```
fs-watcher          <SPDE0048I> Processing 'INITIAL' event for file 'server.admin.web-1.0.1.RELEASE.jar'.
fs-watcher          <SPSC1000I> Creating web application '/admin'.
async-delivery-thread-1 <SPSC1001I> Starting web application '/admin'.
fs-watcher          <SPDE0010I> Deployment of 'com.springsource.server.servlet.admin' version '1.0.1.RELEASE'
  completed.
fs-watcher          <SPDE0048I> Processing 'INITIAL' event for file 'server.admin.splash-1.0.1.RELEASE.jar'.
fs-watcher          <SPSC1000I> Creating web application '/'.
async-delivery-thread-1 <SPSC1001I> Starting web application '/'.
fs-watcher          <SPDE0010I> Deployment of 'com.springsource.server.servlet.splash' version '1.0.1.RELEASE'
  completed.
```

- Log format contains timestamp, logger, code and message
- Code looks like <SPSC1000I>
 - Subsystem, number, level
 - Documented in user guide
 - Level is I for info, W for warning, E for error
- After startup, trace.log is already > 15.000 lines
 - Best to only inspect it when errors occur

Unsuccessful Deploy

- Failure to deploy often caused by missing dependencies
- Log will report failure
- Sometimes requires trace analysis to find root cause

Topics in this Session

- What and how can you deploy
- See what's happening
- **Inspect the end result**
- Controlling the context path

Inspect The End Result

- Applications show up in admin console
- Web application now accessible
 - Context path used shown in log and admin console
- Bundles will now be in started state
 - We can check in Equinox console

Result in Web Admin Console



SpringSource dm Server™



Applications

Admin Console

Result of the last operation: 'Applications Listed'.

Depends on deploy method used

Deployed Applications

Name	Version	Origin	Date	Undeploy
com.springsource.server.servlet.admin	1.0.1.RELEASE	Hot Deployed	1-dec-2008 21:48:50 CET	undeploy
Associated Modules:				
com.springsource.server.servlet.admin	(type: Web)	/admin		
com.springsource.server.servlet.splash	1.0.1.RELEASE	Hot Deployed	1-dec-2008 21:48:52 CET	undeploy
Associated Modules:				
com.springsource.server.servlet.splash	(type: Web)	/		

Deploy an Application

Select an application or bundle to upload and deploy to the server. Valid file formats: *jar*, *war*, *par*.

Application Location

During deployment, artifacts can be transformed:

- WARs become bundles
- **Import-Bundle** and **Import-Library** manifest headers expand to regular OSGi **Import-Package** headers
- Extra Import-Package statements can be added
- And there's more (we'll see on Day 2)

Deployment Pipeline (2)



- This is called the Deployment Pipeline
- End result is visible under 'work' directory
 - Under 'work/com.springsource.server.deployer/Module/...' to be exact
- NB: All this does not apply to repository jars!
 - They're left untouched after being copied into repository/bundles/usr

Provides useful commands to verify deployment:

- **ss**
 - Shows list of installed bundles
 - Including id and status (RESOLVED, ACTIVE, etc.)
- **diag <bundle-id>**
 - Lists unresolved constraints for given bundle
- **headers <bundle-id>**
 - Lists manifest headers for given bundle
 - Will often be rewritten by dm Server during deployment!

Topics in this Session



- What and how can you deploy
- See what's happening
- Inspect the end result
- **Controlling the context path**

- For WARs, file name determines context path
 - MyWebApp.war available under /MyWebApp
- To override, use Web-ContextPath header in META-INF/MANIFEST.MF

Web-ContextPath: myapp

- Currently the only way
- Existing JIRA for setting during deployment
 - Will be implemented in future release

- We can deploy WARs, bundles and PARs
- Using admin console, pickup dir or JMX
- Results are written to log and trace files
- After deploying, everything is a bundle
- Context path controlled through manifest

Lab

dms.deploy

The dm Server's Provisioning Repository

Managing shared bundles

Topics in this Session



- Overview & Outline
- Adding bundles and libraries
- SpringSource Enterprise Bundle Repository
- Configuring the repository
- Provisioning at runtime
- Tooling support

- OSGi allows application modularization in bundles
- Some bundles will have your application code
- Others are shared dependencies
 - 3rd party OSS libraries
 - Reusable components within your organization

- Don't want to package common bundles with our application
 - Prevent 'library bloat'
 - Should be loaded only once, not for each application
- Where to keep the shared dependencies?
- How to make them available to your application?

Provisioning Repository (1)



- All done by the Provisioning Repository
- Contains bundles and *libraries*
 - Some prepackaged with the dm Server
 - You can add others
- Libraries have not been discussed yet
 - For now, just think of them as lists of related bundles
 - e.g. Spring library has core, container, beans, etc.

Provisioning Repository (2)



- Repository provides dependencies
 - No need to package bundles with your application
- Bundles are installed on demand
 - Unused bundles do not consume resources
- Bundles are used as-is
 - Do not go through deployment pipeline like bundles installed using admin console or pickup dir
 - So no transformations applied
- This process is called *provisioning*

Repository Outline



[-] springsource-dm-server-1.0.0



...

[-] repository

[-] bundles



ext

[+] subsystems



usr



installed

[-] libraries



ext



usr

- bundles/ext:
preinstalled bundles
- bundles/usr:
user-installed bundles
- libraries/ext:
preinstalled libraries
- libraries/usr:
user-installed libraries
- installed &
bundles/subsystems:
for internal use only

- Extras go under bundles/usr and libraries/usr
 - Just copy the files in
 - No need for admin console or other tools
- Picked up at runtime
 - No need to restart after adding new bundles
- File names have to be unique
 - Typically consist of Bundle Symbolic Name and version
 - e.g. `org.springframework.core-2.5.6.A.jar`

- Overview & Outline
- **SpringSource Enterprise Bundle Repository**
- Configuring the repository
- Provisioning at runtime
- Tooling support

It's All About Bundles

- All jars in repository must be OSGi bundles
- Many existing OSS libraries are not
 - Though more and more are bundles out of the box
- How do you obtain a bundle version?
 - i.e., one containing a manifest with proper headers

- Ask the developers to provide manifest headers
 - They (should) know best
 - But may not care about OSGi
- Do it yourself
 - Analyze dependencies and provided packages
 - Use tools such as bnd to make life easier
 - Necessary for home-grown or commercial libraries
- Let SpringSource do it!
 - Enter the SpringSource Enterprise Repository...

- Online repository with OSGi-ified versions of many OSS libraries
 - Dependency analysis done by SpringSource
 - Provided for free
- Contains only proper OSGi bundles
 - No Import-Bundle or Import-Library headers
 - No Require-Bundle either
 - Usable by any OSGi container, not just dm Server
- Also holds dm Server library definitions

- Can be used as a web-app, through Eclipse tooling, or as a Maven or Ivy repository
- Searchable contents
 - by name, symbolic name, class, package or resource
- Documents all dependencies of a bundle
- Also holds source jars for use within IDE
- Includes Maven, Ivy, and OSGi code snippets

<http://www.springsource.com/repository>

Demo

Browsing the SpringSource Enterprise Repository

- Maven configuration for bundles:

```
<repository>
  <id>com.springsource.repository.bundles.release</id>
  <name>S2EBR - SpringSource Bundle Releases</name>
  <url>http://repository.springsource.com/maven/bundles/release</url>
</repository>

<repository>
  <id>com.springsource.repository.bundles.external</id>
  <name>S2EBR – External Bundle Releases</name>
  <url>http://repository.springsource.com/maven/bundles/external</url>
</repository>
```

- For libraries, replace bundles with libraries

- Ivy configuration for bundles:

```
<url name="com.springsource.repository.bundles.release">
  <ivy pattern="http://repository.springsource.com/ivy/bundles/release/
    [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
  <artifact pattern="http://repository.springsource.com/ivy/bundles/release/
    [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
</url>

<url name="com.springsource.repository.bundles.external">
  <ivy pattern="http://repository.springsource.com/ivy/bundles/external/
    [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
  <artifact pattern="http://repository.springsource.com/ivy/bundles/external/
    [organisation]/[module]/[revision]/[artifact]-[revision].[ext]" />
</url>
```

- For libraries, replace bundles with libraries

Topics in this Session



- Overview & Outline
- SpringSource Enterprise Bundle Repository
- **Configuring the repository**
- Provisioning at runtime
- Tooling support

- Search path configured in config/server.config file under provisioning
 - Uses default configuration if none configured
- Change defaults to:
 - Share directories between dm Server installations
 - Use Maven/Ivy repositories directly within dm Server

Default configuration (used if not specified):

```
"provisioning" : {  
  "searchPaths": [  
    "repository/bundles/subsystems/{name}/{bundle}.jar",  
    "repository/bundles/ext/{bundle}",  
    "repository/bundles/usr/{bundle}",  
    "repository/libraries/ext/{library}",  
    "repository/libraries/usr/{library}"  
  ]  
}
```

Bold entries are always required!

- Paths can contain wildcards
- {foo} is wildcard for a single file or directory
 - Name is not significant
 - Match can be narrowed: {bundle}.jar only matches jars
- Can also use Ant-style * and **
 - * is like {foo}
 - ** includes subdirectories
- Can also use system properties: \${user.home}

- Example of using Maven repository:

```
"provisioning" : {  
  "searchPaths": [  
    "repository/bundles/subsystems/{name}/{bundle}.jar",  
    "repository/bundles/ext/{bundle}",  
    "${user.home}/.maven/repository/**/{bundle}.jar",  
    "repository/libraries/ext/{library}",  
    "repository/libraries/usr/{library}"  
  ]  
}
```

- Can also used shared directory
 - Windows mounted share, Unix NFS, etc.

Topics in this Session



- Overview & Outline
- SpringSource Enterprise Bundle Repository
- Configuring the repository
- **Provisioning at runtime**
- Tooling support

- Done when Import-Package/Require-Bundle cannot be resolved against installed bundles
- If resolving succeeds:
 - Repository bundle is started
 - Your application deploys
- If resolving fails:
 - Your app/bundle won't deploy
 - Just add dependency to repository and redeploy
 - Typically no Server restart needed

Optional Transitive Dependencies



- Some bundles have *optional* dependencies
 - Only loaded when available
 - e.g. spring-orm optionally depends on iBatis
- When such a bundle is provisioned and optional dependency is missing, it's *dropped*
 - Dependency no longer exists
- Means dependency won't be resolved after adding the missing transitive dependency
 - Applications that failed to deploy will still fail
 - e.g. spring-orm won't resolve to iBatis bundle anymore
- In this case a server restart *is* required

Topics in this Session



- Overview & Outline
- SpringSource Enterprise Bundle Repository
- Configuring the repository
- Provisioning at runtime
- **Tooling support**

- dm Server Eclipse tools offer repository support
- Working with local repository
 - Resolving during development
- Working with SpringSource Enterprise Bundle Repository
 - Search without leaving your workspace
 - Download to local repository
 - Download missing sources

Demo

Using the dm Server Eclipse tools for
working with repositories

- dm Server provides repository for provisioning
- Dependencies not packaged with application
- Bundles loaded on demand
- SpringSource Enterprise Bundle Repository provides OSGi compliant versions of many common libraries

Lab

dms-repo

Bundle Development for the dm Server

Topics in this Session



- Import-Bundle and Import-Library
- Using the Eclipse tooling
- Logging and tracing
- Working with DataSources
- `<service>` and `<reference>` revisited

- We've seen Import-Package and Require-Bundle
 - Standard OSGi headers
- dm Server also supports two others
 - Import-Bundle
 - Import-Library
- Both act as macros, expanded during deployment
 - Result is plain OSGi bundle using only Import-Package

Importing Entire Bundles

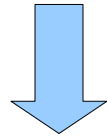
- **Import-Package** is sometimes too fine-grained
 - Really want to refer to an entire bundle
 - Don't want to list all exported packages
- **Require-Bundle** doesn't really work for that
 - Can cause issues with split packages
- **Import-Bundle** provides the middle ground

- Refer to a single bundle
 - Not all its exported packages
- Resolves to Import-Package header with list of packages instead of a Require-Bundle header
 - Including BSN, bundle-version and version attributes
 - Expansion performed during deployment
- Use like Import-Package
 - Supports **version** and **resolution** attributes

Import-Bundle Sample

Import-Bundle:

`com.springsource.org.apache.taglibs.standard;version="[1.1.2,1.2.0)"`



```
Import-Package: org.apache.taglibs.standard.tag.common.core;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.resources;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.tag.rt.sql;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.lang.jstl.parser;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.tag.el.xml;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.lang.jstl;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.extra.spath;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.functions;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.tag.rt.xml;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-version="1.1.2";version="1.1.2",org.apache.taglibs.standard.tag.rt.fmt;bundle-symbolic-name="com.springsource.org.apache.taglibs.standard";bundle-
```

- Even bundles are sometimes too fine-grained
 - Think of Spring: 13 bundles for 1 framework
 - aop, aspects, beans, context, core, jdbc, jms, etc.
 - Apps use Spring 2.5.6: *that's* the real dependency
- dm Server introduces concept of a *library*
 - Named and versioned list of bundles to import
- Defined in .libd file under repository/libraries/*

Library Definition Sample

Library-SymbolicName: org.springframework.spring

Library-Version: 2.5.6.A

Library-Name: Spring Framework

Import-Bundle:

```
org.springframework.aop;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.aspects;version="[2.5.6.A, 2.5.6.A]";import-scope:=application,  
org.springframework.beans;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.context;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.context.support;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.core;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.jdbc;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.jms;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.orm;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.transaction;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.web;version="[2.5.6.A, 2.5.6.A]",  
org.springframework.web.servlet;version="[2.5.6.A, 2.5.6.A]",  
com.springsource.org.aopalliance;version="[1.0.0, 1.0.0]"
```

- org.springframework.spring-library-2.5.6.A.libd
as found under repository/libraries/ext

- Use Import-Library to refer to a library
- Expanded during deployment like Import-Bundle
- Also supports **version** and **resolution** attributes

Import-Library: org.springframework.spring;version="2.5";resolution:=optional

- Results in Import-Package with huge list of packages
- Saves you from finding out low-level package dependencies yourself
 - Both compile-time and run-time!

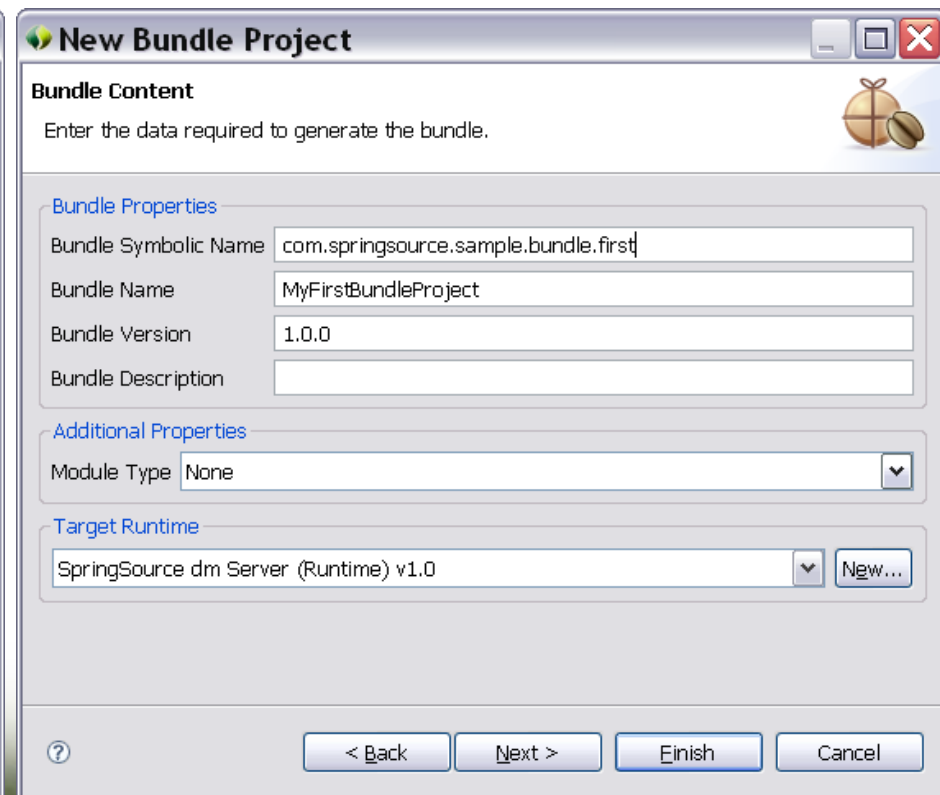
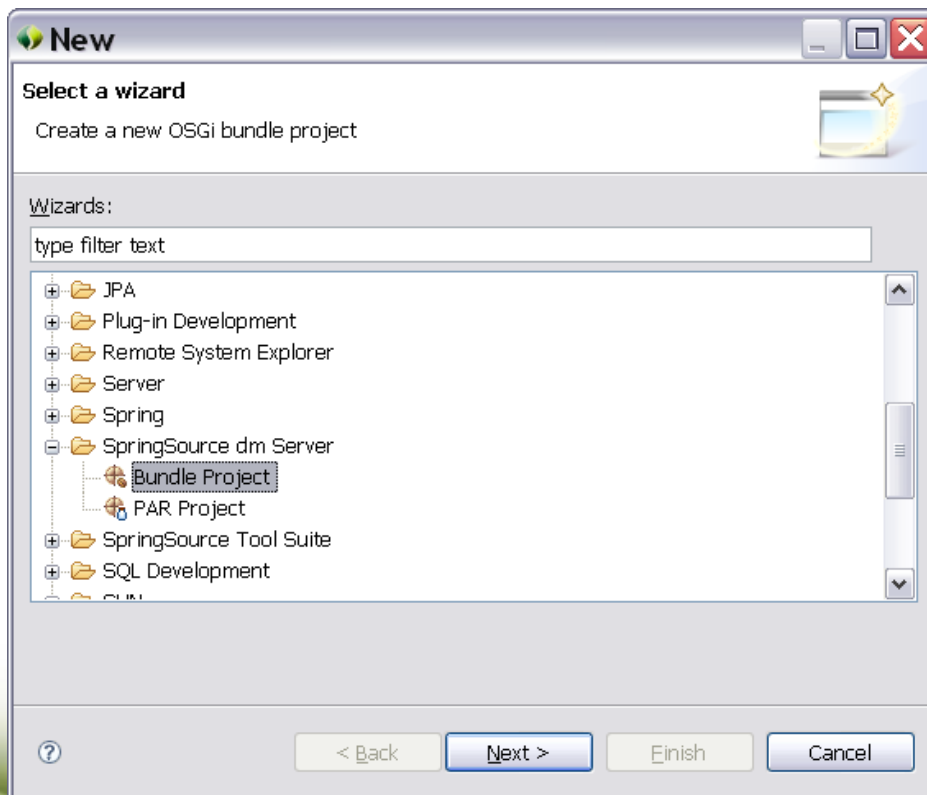
- Keep in mind that Import-Bundle/-Library are resolved during deployment
- Resulting bundle is a plain OSGi bundle
- We don't change the runtime semantics of OSGi!
- But make working with enterprise libraries a *lot* more pleasant

- Import-Bundle and Import-Library
- **Using the Eclipse tooling**
- Logging and tracing
- Working with DataSources
- `<service>` and `<reference>` revisited

- Free dm Server tooling plugin for Eclipse
 - Requires Eclipse 3.3 or 3.4 and Spring-IDE
 - Built-in with SpringSource Tool Suite
- We've seen some features already
 - Deploy to local server
- Let's look at some development features
 - New Project Wizard
 - Manifest editing

Creating A New Bundle Project

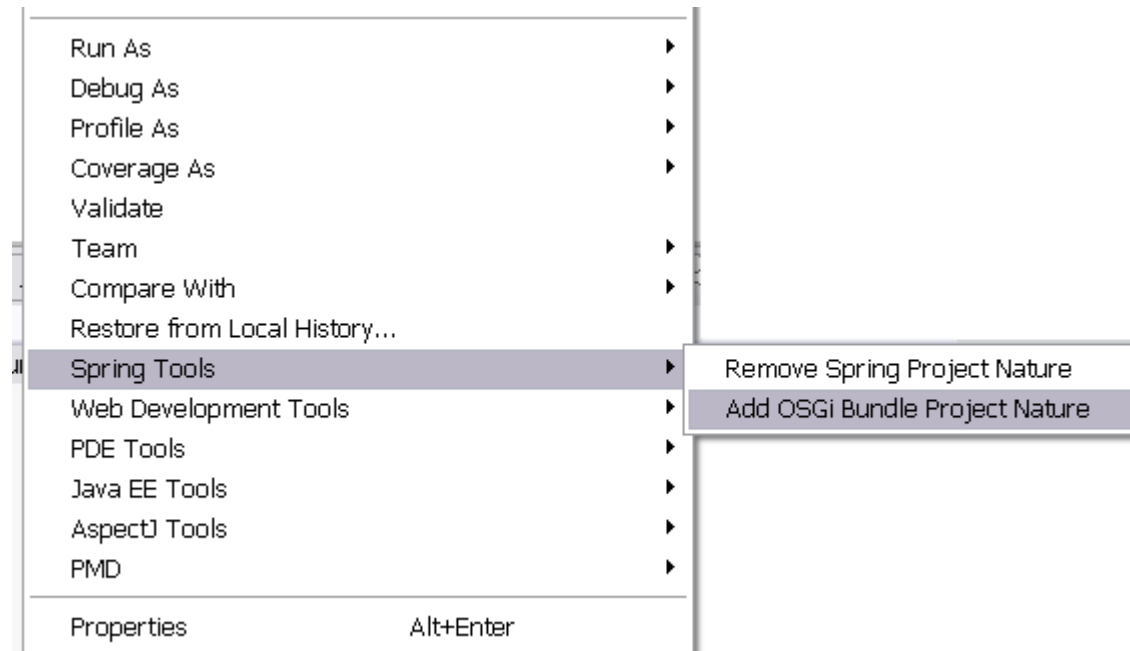
- Step 1 is to create a new bundle project
- There is a wizard to help you do that



- dm Server bundle *project nature*
 - in .project
- Manifest *classpath container*
 - in .classpath
- dm Server bundle *facet*
 - in .settings/org.eclipse.wst.common.project.facet.core.xml
- Enables tight integration with Eclipse and WTP

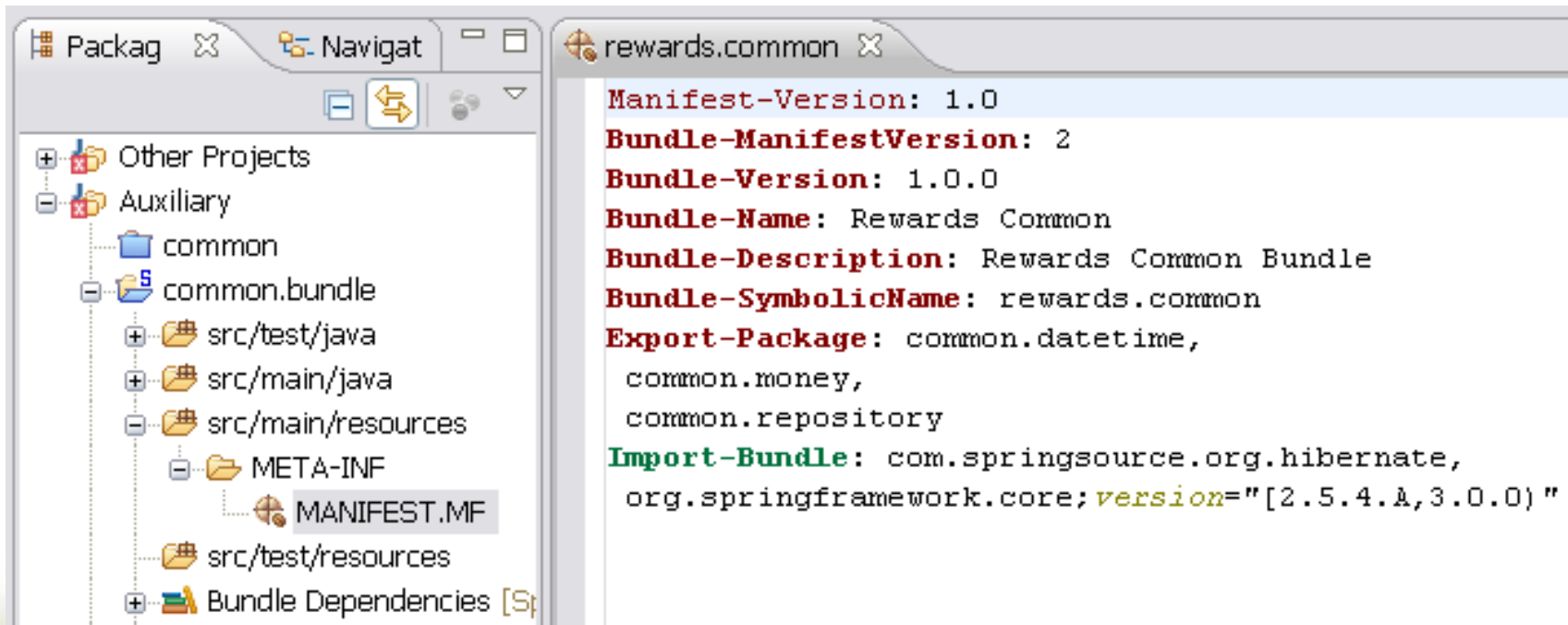
Convert Existing Projects

- Can also be added to existing projects
 - Like Dynamic Web Projects



Editing MANIFEST.MF

- Dedicated editor for bundle manifests
 - Different from PDE support
 - NB: META-INF must be placed in source folder!



Code Assist For Manifests

```
Import-Bundle: com.springsource.org.hibe
```

Bundle names

- com.springsource.org.hibernate
- com.springsource.org.hibernate.annotations
- com.springsource.org.hibernate.annotations.comm
- com.springsource.org.hibernate.ejb

```
export-p
```

Export-Package

Headers

```
Export-Package: common.datetime,  
common.
```

```
Import-B
```

- common.repository
- common.money

```
Import-Bundle: com.springsource.org.hibernate,  
org.springframework.core;version="
```

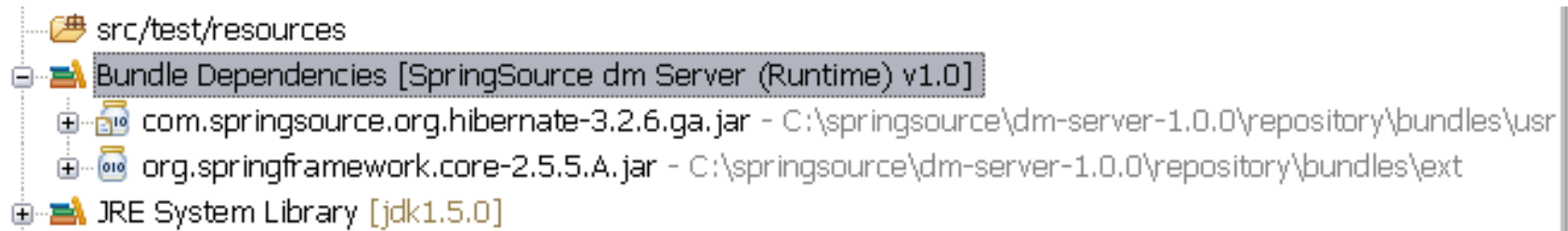
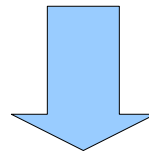
Version ranges

- "2.5.5.A"
- "[2.5.5.A,2.6.0)"
- "[2.5.5.A,2.5.5.A]"
- "[2.5.5.A,3.0.0)"

Package names

- Tooling manages your project's classpath based on imports in the manifest
 - No need to add jars and other projects manually

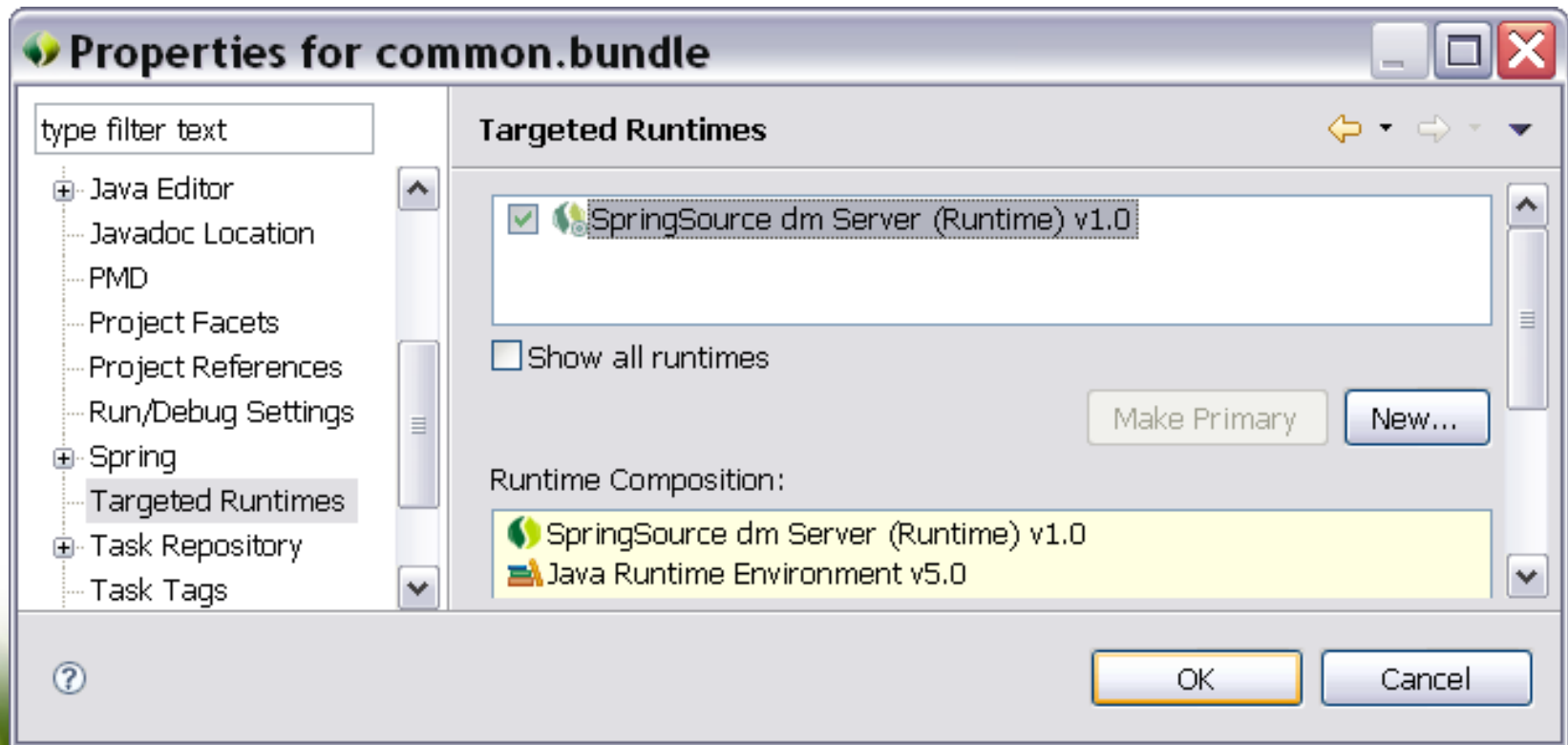
```
Import-Bundle: com.springsource.org.hibernate,  
org.springframework.core;version="[2.5.4.A,3.0.0)"
```



- Very convenient during development
 - But doesn't help with Ant or Maven builds
 - Have to manage dependencies manually at build-time
 - Integrated build support using Bundlor planned Q2 2009
- You have to take some steps to make this work
 - 1) For resolving against server's repository
 - 2) For resolving against other workspace projects

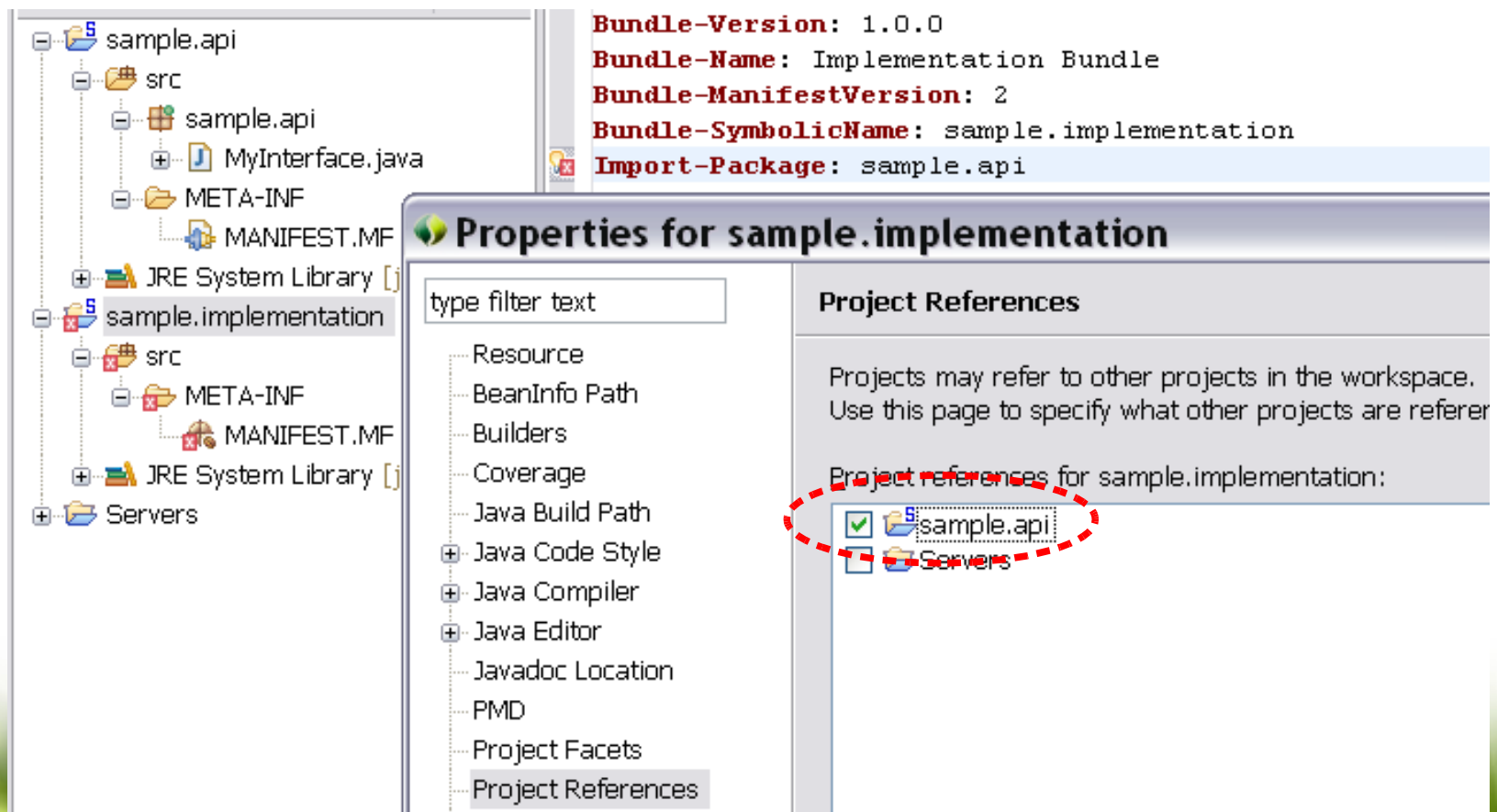
Setting Targeted Runtimes

- Point Targeted Runtimes to dm Server instance
 - Imports resolved against repository of local server
 - Can be set using New Bundle Project wizard



Adding Project References

- For inter-project dependencies, add a Project Reference for imports to be resolved



The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer displays a project structure with 'sample.api' and 'sample.implementation' projects. 'sample.implementation' has a 'src' folder and a 'META-INF' folder. The 'Properties' dialog for 'sample.implementation' is open, showing the 'Project References' tab. The 'Project References' tab lists 'sample.api' and 'Servers' as project references. 'sample.api' is checked, indicating it is a project reference. A red dashed circle highlights the 'sample.api' entry in the list.

Bundle-Version: 1.0.0
Bundle-Name: Implementation Bundle
Bundle-ManifestVersion: 2
Bundle-SymbolicName: sample.implementation
Import-Package: sample.api

Properties for sample.implementation

type filter text

- Resource
- BeanInfo Path
- Builders
- Coverage
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- PMD
- Project Facets
- Project References

Project References

Projects may refer to other projects in the workspace. Use this page to specify what other projects are referred to.

Project references for sample.implementation:

- ☒ sample.api
- ☐ Servers

Topics in this Session



- Using the Eclipse tooling
- **Logging and tracing**
- Working with DataSources
- `<service>` and `<reference>` revisited

- dm Server has lots of support for logging and tracing
 - Part of 'serviceability' support
- High level messages go to console and log file
- Detailed messages and application output go to trace file
- Each bundle also has its own trace file

Logging vs. Tracing



- Note that in dm Server, *tracing* refers to what most people call *logging* in applications
- In dm Server, logging is only done by the server
- Each log message has its own code
 - e.g. **<SPSC1000I>**
 - Specified in the manual

Logging from a bundle



- Logging typically done using Log4J or commons-logging
- Just add Import-Package headers to manifest
- Output will end up in trace file
 - As well as System.out and System.err
 - Your logging configuration is completely ignored
 - But you'll always know where to look
- How does this work?

- dm Server uses SLF4J for logging
- Your Import-Package headers will be resolved by the Server's SLF4J
 - Used through the Log4J or commons-logging API
- This leads to two questions:
 - 1) How to configure this logging?
 - 2) How to use your own Log4J logging?

- Use Application-TraceLevels manifest header
 - Comma-separated key-value pairs
 - Keys: package and class names (can use wildcards)
 - Values: one of error, warn, info, debug, or verbose

Application-TraceLevels: *=info,com.myapp.*=verbose

- Most specific key will match
- Setting is read when starting the bundle
- Currently no way to change at runtime

- Configure log settings for dm Server in `config/server.config`
 - Called tracing in dm Server, remember?

```
"trace": {  
  "directory": "serviceability/trace",  
  "levels": {  
    "*" : "info",  
    "com.foo.*" : "verbose",  
    "com.foo.TheClass" : "debug",  
    "com.bar.AnotherClass" : "verbose"  
  }  
}
```

- When no match for a class, tracing is disabled

Configuring Global Logging



- Location of logs directory also configurable

```
"logs": {  
  "directory": "serviceability/logs"  
}
```

- Only used for dm Server log messages
 - What you also see in the console
- No levels to configure
- Application logging *always* goes to trace file

Using Native Log4J: Why?



- Central trace file through SLF4J has its benefits
- But sometimes you want real Log4J
 - Custom appenders for different output medium or log format
 - Change the log level at runtime
- Simply importing the Log4J packages doesn't work
 - You'll get SLF4J
- So how do you do this?

Using Native Log4J: How?



- Qualify the import with the providing bundle's BSN

Import-Package: org.apache.log4j;
bundle-symbolic-name="com.springsource.org.apache.log4j"

- This bundle is not included with the dm Server
 - Download it from the SpringSource Enterprise Bundle Repository

Using Native Log4J: Why Not?



This approach has a severe drawback:

- Log4J can only be initialized once *per server*
 - Uses static initialization and there's only one bundle
- Means single config file for *all* applications
 - Bundles cannot package their own
 - All but the first one used will be ignored
- Future dm Server versions will support *cloning* bundles
 - Will solve this issue, as well as the issue of *pinning*
 - Current workaround is to include Log4J bundle in your PAR file (explained later)

Topics in this Session



- Using the Eclipse tooling
- Logging and tracing
- **Working with DataSources**
- `<service>` and `<reference>` revisited

- No global DataSources defined in JNDI registry
- Define DataSource in a bundle
 - Do it yourself
 - Or server operator, so you don't have to know DB pwd
- Expose it as OSGi service
- Reference the service from client bundles

DataSource Configuration

- In some config of datasource bundle:

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      init-method="createDataSource" destroy-method="close">
  <property name="driverClassName" value="{jdbc.driver}" />
  <property name="url" value="{jdbc.url}" />
  <property name="username" value="{jdbc.user}" />
  <property name="password" value="{jdbc.password}" />
</bean>
```

- In osgi-context.xml of same bundle:

```
<service interface="javax.sql.DataSource" ref="dataSource" />
```

DataSource Consumption



- In osgi-context.xml of client bundle:

```
<reference interface="javax.sql.DataSource" id="dataSource" />
```

- In some config of same bundle:

```
<bean id="myDao" class="com.example.MyDaoImpl">  
  <property name="dataSource" value="dataSource" />  
</bean>
```

- Not all implementations work with OSGi
 - Often use DriverManager, causing visibility issues
- Best is to use Apache Commons DBCP
 - Version obtained from SpringSource Enterprise Bundle Repository
- No other connection pools supported for now
 - Will probably change in the future

- Noticed the `init-method="createDataSource"`?

```
<bean id="dataSource"  
      class="org.apache.commons.dbcp.BasicDataSource"  
      init-method="createDataSource"  
      destroy-method="close">  
...  

```

- That's there for a reason

- DBCP uses the *thread context classloader* when first Connection is created to load JDBC driver
 - Lazy initialization of pooled connections
- Likely to be classloader from another bundle
 - Which can't see the JDBC driver classes
 - You'll get a ClassNotFoundException
- Forcing eager initialization ensures correct classloader is used
 - i.e. the classloader from the bundle defining the DBCP DataSource

- With multiple applications, this scheme results in multiple DataSources
 - All available in the OSGi registry
- Means **<reference interface="javax.sql.DataSource"/>** doesn't work as-is
 - Might get wrong instance

Topics in this Session



- Using the Eclipse tooling
- Logging and tracing
- Working with DataSources
- **<service> and <reference> revisited**

- Service registry is global, so multiple services may exist implementing the same interface
 - e.g. `javax.sql.DataSource`
- How to reference a particular instance?
 - We need other distinguishing features than type

Picking Your Service



- OSGi and Spring-DM let you choose the service you need
- Filter by service properties; or
- Pick the 'best' service based on ranking

- Service properties are additional meta-data
- Can be defined when registering the service

```
<service interface="javax.sql.DataSource" ref="dataSource">
  <service-properties>
    <beans:entry key="db.name" value="rewards"/>
  </service-properties>
</service>
```

- Property `org.springframework.osgi.bean.name` is automatically set to name of target bean

Filtering By Properties

- Service reference can add additional filters

```
<reference id="dataSource" interface="javax.sql.DataSource"  
          filter="(db.name=rewards)" />
```

- Shortcut for filtering by bean name

```
<reference id="dataSource" interface="javax.sql.DataSource"  
          bean-name="someDataSource" />
```

- Allows for uniquely identifying desired service

- Based on LDAP
 - Supports boolean operator for combinations
 - Compare using =, ~=, >= and <= (~= is the 'like' operator)
- Examples:
 - (Bundle-SymbolicName=dms.bundle.provider.start)
 - (&(objectclass=org.osgi.service.http.HttpService)(port=80))
- objectclass set automatically by Spring-DM using the *interface* attribute
 - No need to specify it yourself in a filter

- Services can have a *ranking*
 - Default is 0
- With multiple matches, highest ranking service wins
 - Allows for a default and fallback instances

```
<service interface="javax.sql.DataSource" ref="dataSource"  
        ranking="9"/>
```

- dm Server makes OSGi easier for enterprise development
 - Custom headers
 - Dedicated tooling
 - Serviceability features
- Application development is different than for standard Java EE
 - Think in terms of bundles and services for everything
 - Use Commons-DBCP to define your DataSources

Lab

dms-bundle

Developing web applications for the SpringSource dm Server

- **Overview of Web Support**
- Various WAR formats supported:
 - Plain WAR
 - Shared Libraries WAR
 - Shared Services WAR
 - Web Modules

Overview of Web Support



- dm Server fully supports web applications
- Has embedded Tomcat 6.0 server
- You can deploy plain WAR files
- WAR files can make use of OSGi
 - Importing types and services
- Dedicated Web Module format for Spring-MVC
 - Bundle, not WAR file
 - Bootstraps DispatcherServlet for you

- WAR files converted to OSGi bundles
- Various imports added automatically
 - Mostly `javax.*` packages
 - What you expect to be there in Tomcat runtime
- Context Path determined from WAR name or by setting **Web-ContextPath** manifest header
- WAR name defines Bundle Symbolic Name
 - You can also provide your own BSN in the manifest
 - That is a best practice for shared services WARs

Topics in this Session

- Overview of Web Support
- Various WAR formats supported:
 - **Plain WAR**
 - Shared Libraries WAR
 - Shared Services WAR
 - Web Modules

- Standard WARs can be deployed as-is
 - Well, almost:
 - No support for JNDI yet
 - No support for J2EE security with your own realms yet
- Good as starting point for migrating to OSGi
- Doesn't benefit from OSGi strong points
 - All libraries need to be included in WEB-INF/lib
 - Code cannot be shared easily with other applications

Topics in this Session



- Overview of Web Support
- Various WAR formats supported:
 - Plain WAR
 - **Shared Libraries WAR**
 - Shared Services WAR
 - Web Modules

Shared Libraries WAR (1)



- First step to migrate to OSGi
- Use shared libraries through OSGi imports instead of WEB-INF/lib jars
- Simply add headers to MANIFEST.MF
 - Import-Package, Import-Bundle, Import-Library
- Prevents 'library bloat'
 - WARs don't have to include dozens of 3rd party jars
 - At runtime, jars are loaded only once per server instead of for each application

- Requires libraries to be present as bundles in bundle repository!
 - Download from SpringSource Enterprise Bundle Repository if these libraries are not OSGi bundles yet
- Using both WEB-INF/lib and imports works, but is not recommended
 - Bundles can't see classes from WEB-INF/lib jars
 - Only do it if there's no OSGi bundle for your jar and other bundles don't depend on it

Topics in this Session



- Overview of Web Support
- Various WAR formats supported:
 - Plain WAR
 - Shared Libraries WAR
 - **Shared Services WAR**
 - Web Modules

- Uses Spring-DM <reference> or <list> to access shared OSGi service(s)
 - These services can be upgraded or replaced at runtime
 - Often without restarting your application!
- Allows common code to be used by multiple applications
 - Not just classes, but actual service instances!
 - SOA within a single VM

- Need custom ApplicationContext to do this
 - Regular XmlWebApplicationContext not OSGi-aware
- Set through context param in web.xml
 - Used by the ContextLoaderListener

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
    com.springsource.server.web.dm.ServerOsgiBundleXmlWebApplicationContext
  </param-value>
</context-param>
```

- Only used for root context
 - DispatcherServlet context typically doesn't need it
 - For Spring-MVC there's a better option anyway

To work with Shared Services WARs in Eclipse:

- Create as WTP Dynamic Web Project
 - Like any WAR
- Add Spring Project Nature
 - For extra Spring IDE support
- Add OSGi Bundle Project Nature
 - For dm Server tooling support

Programmatic Access (1)



- ApplicationContext can be accessed from your web tier
- Use WebApplicationContextUtils like regular Spring web applications
- Example:

```
public class MyServlet extends HttpServlet {  
    private Service service;  
  
    @Override public void init() throws ServletException {  
        WebApplicationContext context =  
            WebApplicationContextUtils.getRequiredWebApplicationContext(  
                getServletContext());  
        this.service = (Service) context.getBean("someService");  
    }  
    //...  
}
```

Programmatic Access (2)

- Can also access BundleContext if required
- Lookup in ServletContext under constant name
- Example:

```
public class MyServlet extends HttpServlet {  
  
    @Override public void init() throws ServletException {  
        BundleContext context = (BundleContext) getServletContext().getAttribute(  
            ServerOsgiBundleXmlWebApplicationContext.BUNDLE_CONTEXT_ATTRIBUTE);  
        // use the BundleContext here  
    }  
    //...  
}
```


- Middle-tier code can't use that trick
- Bean classes can implement Spring-DM's *BundleContextAware* instead
 - only when using `ServerOsgiBundleXmlWebApplicationContext`

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

- Typically not needed in an application
- Ties your code to both Spring and OSGi

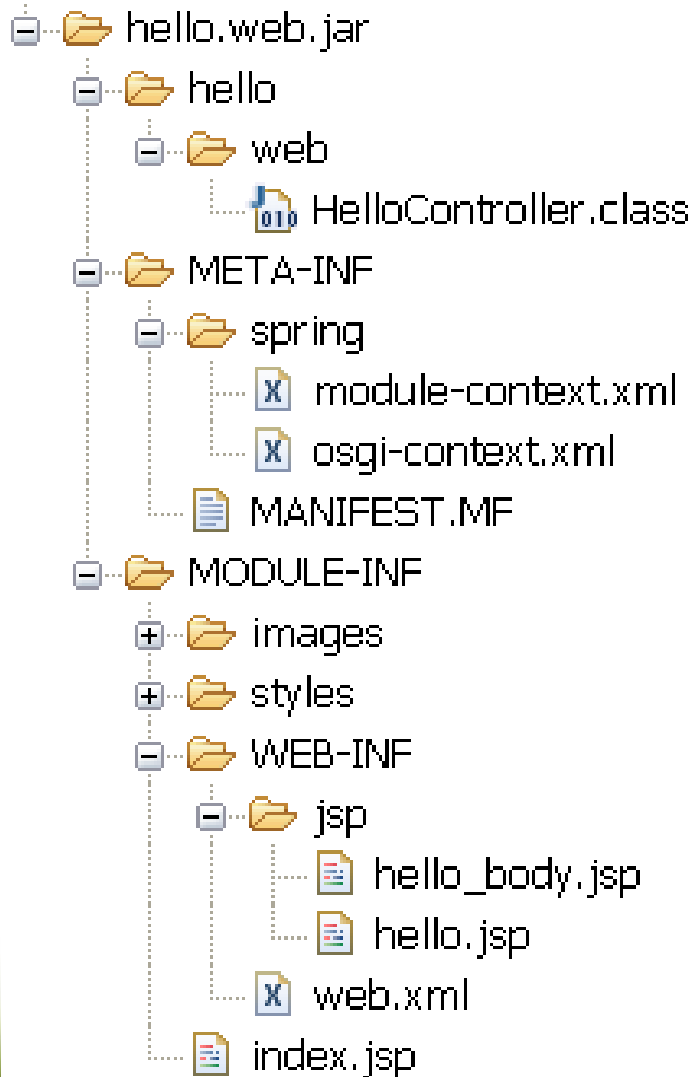
Topics in this Session



- Overview of Web Support
- Various WAR formats supported:
 - Plain WAR
 - Shared Libraries WAR
 - Shared Services WAR
 - **Web Modules**

- Dedicated format for Spring-MVC
- Bundle with extra manifest headers and MODULE-INF directory
- Creates MVC DispatcherServlet automatically
 - No <servlet> or <servletmapping> in web.xml needed
- Uses single WebApplicationContext
 - No root context, so no ContextLoaderListener needed
- Supports web.xml *fragment*
 - Merges given web.xml with own configuration

Web Module Layout



classpath is the root

regular META-INF goes in root

web content goes under MODULE-INF

No WEB-INF/classes
or WEB-INF/lib

- **Module-Type: Web**
 - Indicates bundle is a web module (required)
- **Web-DispatcherServletUrlPatterns: /web/***
 - The servlet mapping for the DispatcherServlet
 - Defaults to *.htm
 - Can use comma-separated list
- **Web-ContextPath: rewards**
 - Like with WARs: the context path to use
 - Defaults to file name minus extension
 - Use single '/' for root context

- **Web-FilterMappings: securityFilter;url-patterns:="*.htm,*.jsp"**
 - Registers a ServletFilter using a DelegatingFilterProxy
 - So name must be a bean name
 - Syntax:
**<name>; [targetFilterLifecycle:=<true|false>;]
url-patterns:="<patterns>"
[; dispatcher:="<dispatcher mappings>"]**
 - *targetFilterLifecycle* indicates if lifecycle events should be delegated to filter bean: defaults to true
 - *url-patterns* follows Servlet spec
 - *dispatcher* is comma-separated list of dispatcher mapping values (e.g., REQUEST, FORWARD, INCLUDE, ERROR)
 - Can have comma-separated list

- Other settings can be made in normal web.xml
 - Located under MODULE-INF/WEB-INF
- Contents will be merged with generated contents based on manifest headers
- Might not be needed for very simple applications
 - Then just leave out web.xml

- WARs get extra imports added during deploying
 - all packages exported by the OSGi system bundle, excluding any packages beginning with "org.eclipse" or "com.springsource"
- Means you might need more imports in your web module than in your WAR
 - e.g. javax.sql, org.w3c.dom
- Best practice:
always explicitly import known dependencies!
 - even if they're imported by default
 - everything not under java.*

- The dm Server fully supports web applications
- WAR files can be deployed as-is, use bundles instead of WEB-INF/lib and even use shared OSGi services
 - The latter requires custom WebApplicationContext
- For Spring-MVC you can use a Web Module instead of a WAR
 - Auto-defines DispatcherServlet and WebApplicationContext
 - Later dm Server releases will support web modules without Spring-MVC as well

Lab

dms.web

Platform Archives

An overview of the PAR file format

- **Why bundles are not enough**
- The PAR file
- Deploying
- Versioning
- Classloading and Load Time Weaving
- PAR files in Eclipse tooling

Bundles Are Not Enough

- No single unit for (un)deploying
- No single log or trace file
- Lack of application scoping
- Some things just don't work

No Single Unit For Deploying



- Single application consists of multiple bundles
- Deployment involves all those bundles
 - No single archive to deliver like WAR or EAR file
- Have to be installed in the right order
 - To satisfy dependencies
- After deployment nothing indicates bundles belong together
 - Just a bunch of bundles like all others
 - Makes uninstall problematic: is bundle shared or application-specific?

No Single Log Or Trace File



- Each bundle has its own trace logging
- Have to inspect trace files for all bundles for single application
- Hard to see what's happening
- Hard to do trace file maintenance

Lack Of Application Scoping



- No 'private' types or services can be defined
 - e.g. application DataSource
 - Private to what? There's no application concept
- Cannot deploy same application twice as different version
 - Exported and imported types and services conflict
- No global imports for all bundles in application
 - Means many bundles have to import e.g. Hibernate even if they have no compile-time dependency

Some Things Don't Work



- Load Time Weaving
 - Classloaders don't support instrumentation
 - Even if they do, what should be instrumented?
 - There's no application boundary
 - Might instrument classes from other applications as well
- Classpath scanning
 - So no `<context:component-scan/>`
- Context Classloader
 - Not defined in the OSGi spec
 - No sensible way to manage it with just bundles

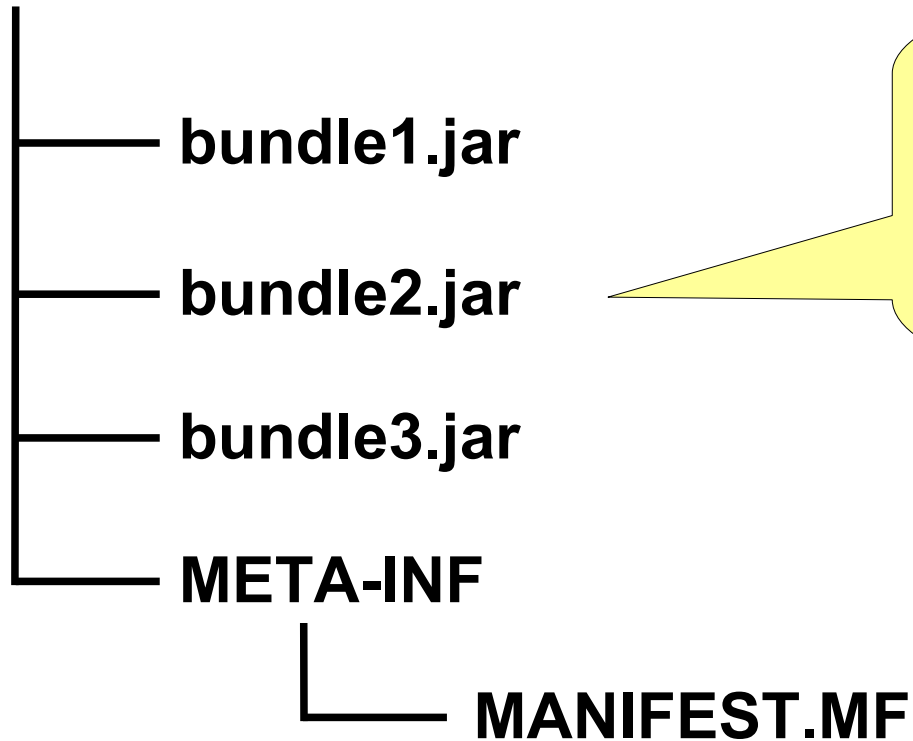
- Why bundles alone are not enough
- **The PAR file**
- Deploying
- Versioning
- Classloading and Load Time Weaving
- PAR files in Eclipse tooling

- Platform Archive
- Groups bundles into application
 - Jar of bundles
 - Extension is .par instead of .jar
- Has special manifest headers
- Single unit for (un)deployment

- Trace file per PAR
 - Not per bundle
- Supports scoping and versioning
- Enables certain techniques to be used
 - Load Time Weaving
 - Classpath Scanning
 - Context Classloading

PAR Layout

application.par



Can also
contain shared
services WARs

Manifest-Version: 1.0

Application-SymbolicName: rewards

Application-Version: 1.0.0

Application-Name: RewardsNetwork

Application-Description: Rewards Network for lab

- SymbolicName and Version required
- Version follows OSGi conventions
- Name and Description are optional

Topics in this Session



- Why bundles alone are not enough
- The PAR file
- **Deploying**
- Scoping and Versioning
- Classloading and Load Time Weaving
- PAR files in Eclipse tooling

Deploying a PAR File

- PAR is deployed as single unit
- Admin Console shows both PAR and contained bundles
- Order of bundles is not important anymore
 - Will all be installed first before they are started, so dependencies will resolve successfully
- Can be undeployed as a whole

Synthetic Bundles

- For each PAR a *synthetic bundle* is generated
- Imports exported packages of all PAR bundles
- Enables thread context class loading
 - e.g. using `Class.forName()`

dms.template.rewards.par.solution 1 file:/C:/springsource/dm-server-1.0.0/sta



Associated Modules:

dms.template.rewards.web.solution	(type: Web)	/template
data.source	(type: Bundle)	No personality identifier
dms.template.rewards.internal.solution	(type: Bundle)	No personality identifier
dms.template.rewards.par.solution-synthetic.context	(type: Bundle)	No personality identifier
dms.template.rewards.solution	(type: Bundle)	No personality identifier
rewards.common	(type: Bundle)	No personality identifier
rewards.data.source.initializer	(type: Bundle)	No personality identifier

- Bundles within the PAR cannot be updated by simply reinstalling them
 - Can only be done for regular, 'global' bundles
- Updates can be made using Deployer MBean
 - Eclipse tooling does this
 - Not an official SPI
 - Might be better supported in future versions
 - PARs typically redeployed as a whole anyway

- Each PAR has its own trace file
- Output from all bundles in PAR goes there
 - No more separate files per bundle
- Don't have to correlate log entries from different files anymore
 - Centralized overview per application
- Works through SLF4J
 - Just like the bundle trace logging

- Why bundles alone are not enough
- The PAR file
- Deploying
- **Scoping and Versioning**
- Classloading and Load Time Weaving
- PAR files in Eclipse tooling

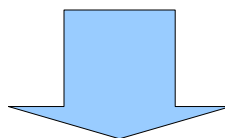
- PAR files are *versioned*:
applies to all of their bundles
 - Can deploy same application twice with different version and same bundles
- Types and services of bundles in PAR are *scoped to PAR*
 - Means they are not visible to other bundles
 - But PAR bundles can see global types and services

Scoping Types And Services



- Under the covers this works through plain OSGi
- Platform adds and applies properties and filters

```
<service interface="hello.service.HelloService" ref="helloService"/>  
  
<reference id="helloDao" interface="hello.dao.HelloDao"/>
```



```
<service interface="hello.service.HelloService" ref="helloService">  
  <service-properties>  
    <beans:entry key="com.springsource.server.app.name"  
                 value="hello.par"/>  
  </service-properties>  
</service>  
  
<reference filter="(com.springsource.server.app.name=hello.par)"  
           id="helloDao" interface="hello.dao.HelloDao"/>
```

- Why bundles alone are not enough
- The PAR file
- Deploying
- Scoping and Versioning
- **Classloading and Load Time Weaving**
- PAR files in Eclipse tooling

- PAR allows context classloader to be managed
- Third-party code can load your classes!
 - Without needing explicit Import-Package
 - e.g. Hibernate for generating CGLib proxies
- Also enables classpath scanning
 - For `<context:component-scan>` or JPA Entities
- Important features to make many enterprise libraries work in an OSGi environment

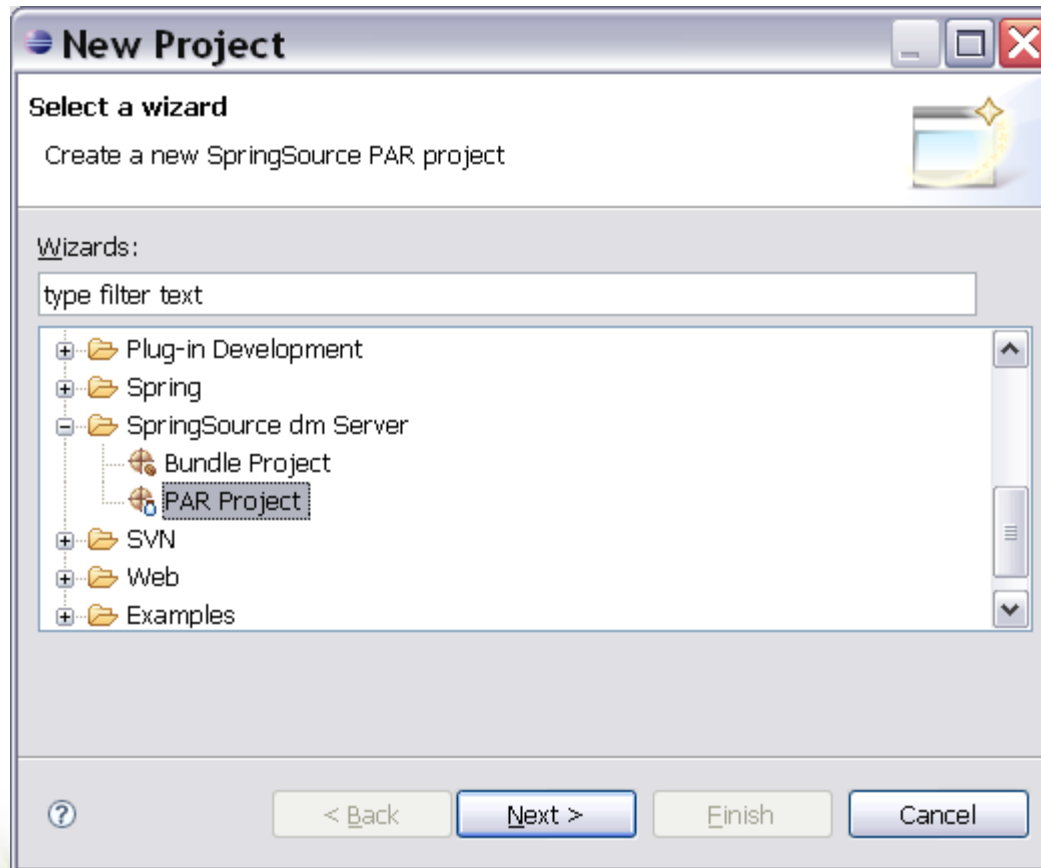
- Imports can also be scoped to application instead of single bundle now
- `Import-Bundle:`
`com.springframework.org.hibernate;`
`version="[3.2.6.ga, 3.3.0)";`
`import-scope:=application`
- Prevents all bundles that import domain entities from having to import Hibernate as well

- LTW works within a PAR file
 - Platform provides instrumentation-capable classloaders
 - PAR provides boundary for classes to instrument
- Needed for most JPA providers
- Can be used with AspectJ as well
 - `<context:load-time-weaver aspectj-weaving="on"/>`
- Requires **Import-Library** for weaving library in all bundles to be woven
 - import-scope works very nicely here

- Why bundles alone are not enough
- The PAR file
- Deploying
- Scoping and Versioning
- Classloading and Load Time Weaving
- **PAR files in Eclipse tooling**

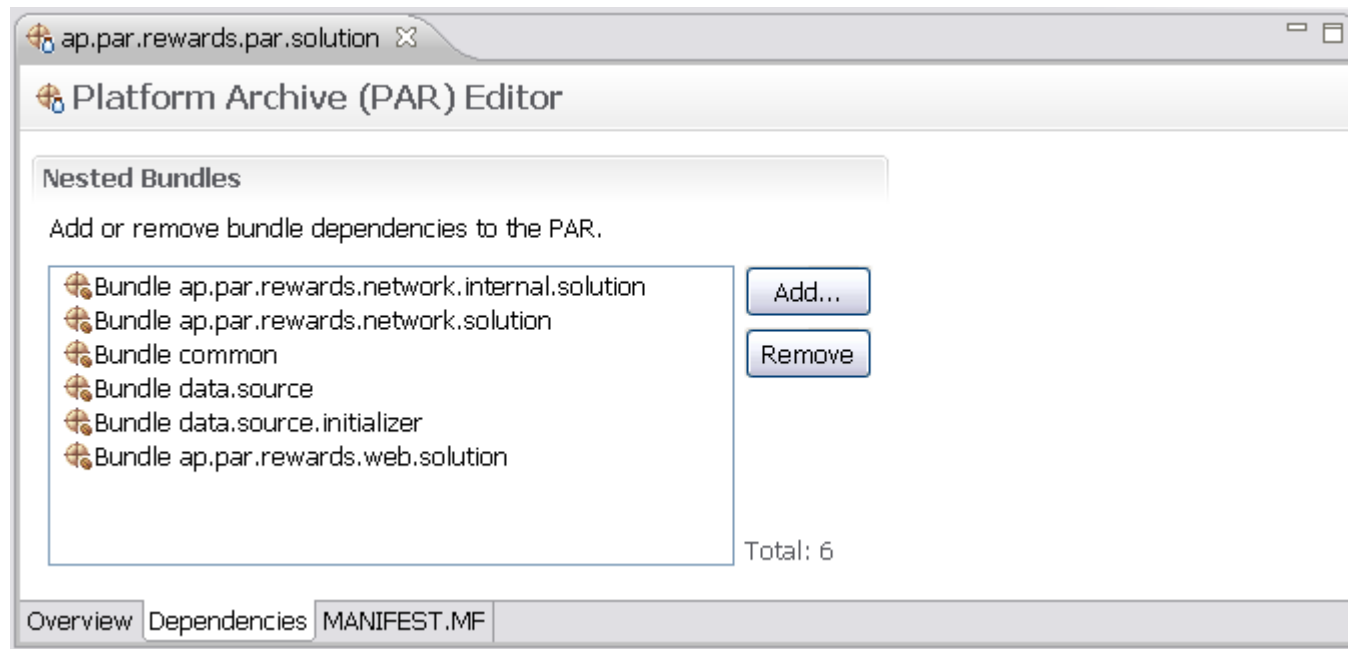
PAR Files In Eclipse (1)

- Use Wizard to create new PAR project



PAR Files In Eclipse (2)

- Use dedicated manifest editor to add bundle projects and edit custom headers



- Bundle jars can also be added to project root

PAR Files In Eclipse (3)

- Bundle projects that are in the same PAR can simply import each others exports
- No need to add an explicit Project Reference to the exporting Bundle Project like with non-PAR Bundle Projects
- Still require Project References to Bundle Projects *outside* the PAR

- PAR files are the preferred way of deploying
- Single unit to deliver and to manage
- Enable applications to use enterprise libraries
- Still give OSGi benefits of multiple bundles
- Scope types and services of contained bundles
- Can still use shared types and services deployed globally

Lab

dms.par

Best Practices for dm Server development

Some guidance in creating OSGi-
based applications for the dm Server

Topics in this Session



- Common Errors & Best Practices
- Modularizing your application
- Versioning
- Build and dependency management
- Operational aspects

- Obvious difference between 'normal' Enterprise Java and OSGi is how classloading works
- Cause of most errors in OSGi / dm Server applications
- Much existing code is not OSGi-compatible
 - Uses `Class.forName()`
 - Expects thread context classloader to be able to see classes in the same jar as the code that uses it
 - Splits classes in the same packages over multiple jars
 - Expects to get its own classloader for each application

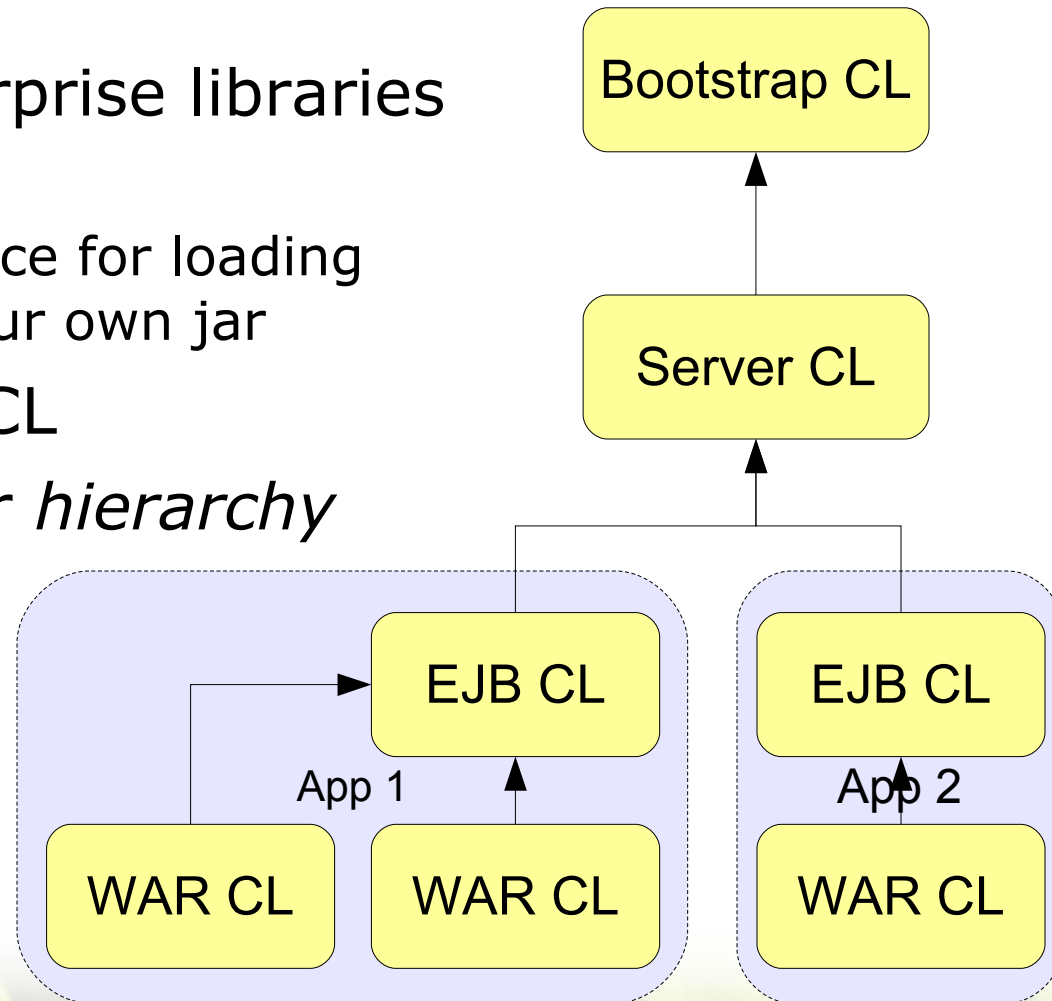
Using `Class.forName(String)`



- Many Enterprise libraries use `Class.forName(String)` to dynamically load a class
- Typically uses the wrong `ClassLoader` under OSGi
 - i.e. the bundle's own classloader
- Resulting in `ClassNotFoundException`s
- Solution: pass in the `ClassLoader` to use
 - `Class.forName(name, true, classLoader)`
 - `ClassLoader` gets passed in as parameter to calling code or is determined in the code itself

Thread Context ClassLoader

- Use of TCCL in enterprise libraries widely spread
 - Actually a best practice for loading resources outside your own jar
- Server manages TTCL
- Assumes classloader *hierarchy*
 - calling code at same level or lower in the hierarchy
 - So TCCL sees what's needed



- Doesn't work in OSGi in general
 - Spec doesn't even define what TCCL will be
- dm Server manages TCCL
 - Set to bundle that started call stack
- But that bundle can't see internals of bundle that has the call to TCCL
 - The whole point of OSGi
- Means relying on TCCL is dangerous
 - Will often fail, depending on who is calling the code

- Example: Hibernate
 - Uses TCCL for loading entity classes, loading hibernate classes from proxied DB connections, etc.
 - Means bundles need to import Hibernate even without a compile-time dependency
 - Remember the dms-par lab?
- Example: Commons-DBCP
 - Uses TCCL to load JDBC Driver classes
 - By default lazily, so TCCL doesn't see those classes
 - TCCL will be set to CL from another bundle
 - Solved by eagerly creating Connection on bundle startup

- PARs can help
 - TCCL tied to synthetic bundle
 - Imports all exports from other PAR bundles
 - Still doesn't help with global bundles outside the PAR
 - import-scope:=application
 - May introduce unnecessary imports
 - But at least at runtime, not at development time
- Again, explicit ClassLoader passing is best
 - But often this is third party code
 - In that case dm Server does the best it can do

- Many libraries store application state in statics
 - Example: Log4J
- Works when using classloader hierarchy where each application has its own copy of library
 - Each application has private copy of a class
- Breaks down in OSGi
 - Single bundle for all applications
 - So all bundles share the same class
 - Results in single configuration for entire server

- **Avoid statics if at all possible**
 - Bad practice for many reasons
- For future release, support for *cloning* bundles is planned in dm Server
 - Gives each application its own copy of a bundle
 - Solves both the static state issue and the *pinning* issue (hard-wired transitive dependencies for a bundle)
- Until then, you can also package bundles with your PARs to create a private copy

- In regular Java, multiple jars can define classes in the same package
 - e.g. `org.springframework.ws.server.endpoint.adapter` exists in `spring-ws-core-1.5.4.jar` and `spring-ws-core-tiger-1.5.4.jar`
- In OSGi, only one bundle can provide a package
- Also means a bundle shouldn't import a package it defines itself as well
 - Will not be able to see its own classes anymore
 - Even if it doesn't export that package!

- **Avoid split packages if you can!** If not:
- Use *fragment bundle* to add classes to an exported package of a host bundle
 - Used by Spring-WS version in Enterprise Bundle Repository
 - NB: Fragment bundles cannot be deployed, but only placed in the repository
- Use *Require-Bundle*
 - Can re-export import from another bundle
 - Should be considered as last resort only

- Interfaces for use by client code and implementation(s) often in the same package
- Plain Java doesn't define visibility of packages
- OSGi does: should only export your API!
 - Provides proper encapsulation of your module
- Means using dedicated packages for interfaces defining public API
 - Export only those packages (avoiding split packages)
 - All other code stays internal to bundle

Depending On Bundles

- OSGi defines Require-Bundle
- dm Server adds Import-Bundle
- Both tie your code to a specific bundle
 - Violates OSGi principle that packages should be substitutable
 - Causes more dependencies than needed

- In general, it's better to use Import-Package
 - At least for your own code and things like javax.*
 - Doesn't tie you to particular bundle
 - Causes minimal set of dependencies
- Import-Bundle for enterprise libraries is fine
 - You don't want to investigate all the packages you're depending on, esp. at runtime
 - Prefer it over Require-Bundle

Note: Enterprise Bundle Repository only uses Import-Package in all its bundles!

- Many applications use serialization
 - For storing state
 - For RPC, e.g. using RMI
- Can result in dependency on many concrete classes
 - Causing tight coupling and many extra required exports and imports
- Also, RMI doesn't work in OSGi
 - Classloader used by RMI Registry to deserialize objects can't see classes without all sorts of hacks

- Try to avoid serialization
 - Other representations of object's state may be more appropriate
- If you must, think about serialized form
 - Try making it insensitive to bundle's implementation details
- Don't use RMI
 - Other remoting mechanisms like Spring HTTP remoting and Hessian or Burlap work just fine

Topics in this Session



- Common Errors & Best Practices
- **Modularizing your application**
- Versioning
- Build and dependency management
- Operational aspects

- How to split up your application in bundles is not an easy question to answer
- Question is really how to partition and what granularity to use

Partitioning can be done in different ways:

- **Vertical:** *functional* partitioning
 - For example orders, warehouse, billing and CRM
- **Horizontal:** *technical* partitioning
 - Web, services, repositories, infrastructure
- A combination of the two

- Bundles represent functional modules
- Preferred approach for big enough applications
 - Single module assignable to a team of developers
 - Encapsulates internals like repositories
 - OrderRepository only needed in 'order' module
 - Minimizes module's 'surface area'
 - Only needs to expose its business interfaces
- Might not work well for small applications
 - Might not need partitioning in the first place

- Bundles represent architectural layers
- Natural approach to many developers
 - Tend to think of layers as modules already
- Allows for replacing layers easily
 - For testing or during early development
 - Deploy stubbed repository bundle without changing services module
- Typically means more maintenance
 - Use cases spread across multiple bundles
 - So changes often span bundle boundaries

- Web Resources like JSPs cannot be split across multiple bundles
 - Not for single ServletContext and HttpSession at least
- Must use single web module/shared services WAR
 - Even when using vertical partitioning
- This limitation will be removed in future versions of the dm Server
 - Requires changes to the internal Tomcat server

- Most applications use shared infrastructure across functional, vertical slices
 - Same DataSource, transaction manager, JMS ConnectionFactory, etc.
- Creating infrastructure bundle(s) makes sense
 - Even when using vertical partitioning: there's no JNDI registry with globally defined resources!
 - Simply expose resources as OSGi services
 - By application developers or operations team

Bundle Granularity (1)

- How much stuff goes in one bundle
- Use same rules as for object orientation
 - Bundles need to have a clear responsibility
 - High cohesion within a bundle
 - Loose coupling between bundles
- Works well with vertical partitioning
 - Horizontal tends to increase dependencies between bundles

Bundle Granularity (2)

- Easy to make modules too fine-grained
 - Often seen in samples and labs
 - To show how OSGi works
 - Doesn't necessarily represent best practice!
 - Better to extract extra bundle later if desired
- Typically shouldn't create bundle if non-OSGi application wouldn't have dedicated jar for the same code

- Interfaces can be placed in a dedicated bundle
 - As opposed to just a dedicated package
- Clients depend only on interface bundle
 - Typically by using services provided by implementations in another bundle
- Means implementation bundles can be updated or replaced without client bundles also restarting
 - Web application keeps running while backend services are updated
 - Would cause rippled restart effect with single bundle

- Only works in straightforward cases
 - i.e. without using weaving or proxying classes
- When using e.g. JPA weaving, client bundles still need to be restarted
 - Dependency isn't just on service interfaces anymore
- Use this approach if:
 - 1) You don't use weaving or similar techniques
 - 2) It's important to avoid rippled bundle restarts
- Just adds to maintenance in other cases

- Pure vertical partitioning not a feasible option for most applications
 - Typically need web & infrastructure bundles
 - Can still be applied to other bundles for encapsulation
- Apply common sense in module design
 - Limit the number of modules
 - Think of what makes sense
- Best practices are still emerging

Topics in this Session



- Common Errors & Best Practices
- Modularizing your application
- **Versioning**
- Build and dependency management
- Operational aspects

- Several things can be versioned
 - Packages
 - Bundles
 - PARs
- How should you apply versioning in your applications?

- Packages are versioned on export
 - Do NOT inherit version of their exporting bundle!
 - If not specified, have version 0.0.0
- Import-Package can use single version or range
 - Single: [2.5.6.A,2.5.6.A]
 - Range: [1.1.0, 2.0.0) or just 1.1.0 for no upper bound
 - When not specified, uses 0.0.0 (anything goes)
- How to best apply?

As a rule of thumb you should:

- version your package exports
 - Esp. for global bundles used by multiple applications
 - Less important for bundles in a PAR (already scoped)
- use versions on Import-Package
 - With sensible version ranges
 - "[1.2.3, 1.2.3]" probably too strict:
what's wrong with 1.2.4?
 - "3.2.4" probably too loose:
will 3.3.0 work? What about 4.0.0?
 - Depends also on versioning policy used

- Bundle versions typically follow project guidelines
 - Have the same version as the product release
- PAR versions do the same
 - Remember bundles in a PAR are scoped automatically
 - Means you only have to increase application version
 - Might not bother with bundle versions if they're only internal to the PAR
 - Best practice is to version bundles with new releases if they've changed as well

Increases in versions indicate type of changes

- **Major** number:
 - Typically many new features and APIs
 - Often not backwards compatible
 - API changes, new configuration formats, etc.
- **Minor** and **micro** number:
 - Some new features, bug fixes, no major changes
 - Often backwards compatible (micro even forwards)
 - May have deprecated code removed
- **Qualifier**:
 - No standardized meaning (build label, platform, etc.)

Using Version Qualifiers



- Qualifier is sorted **alphabetically**
- Means 1.2.3.M3 and 1.2.3.RC1 come *after* 1.2.3
 - SpringSource releases use ".RELEASE" qualifier because of this
- Also means 1.0.0.10 comes before 1.0.0.9
- Don't use numbers as qualifiers!
 - Avoids any confusion and chance for mistakes

Topics in this Session



- Common Errors & Best Practices
- Modularizing your application
- Versioning
- **Build and dependency management**
- Operational aspects

Build And Dependency Management Issues



- dm Server Eclipse Tools manage classpath during development
 - Used in all labs
- Of no use during Ant or Maven build
 - Or during out-of-container integration tests, because transitive dependencies are missing
- Means manual management of dependencies

Manual Dependency Management



- Results in some duplication
 - Both Manifest and build have dependencies listed
- Will improve with later dm Server releases
 - Integrated dependency management is planned
 - Using tool called Bundlor: available in Q2 2009
 - Classpath container will also contain transitive dependencies
- But what to use *now*:
 - for (transitive) dependencies during build and test;
 - for managing manifests?

Dependency Management – Build Time



- Use Maven or Ivy for transitive dependencies
 - Download bundles from Enterprise Bundle Repository
 - Available during build, i.e. compilation and testing
 - Can configure dm Server to use local repository as-is
- Cannot be used yet to generate manifest
 - Doesn't work for Import-Package
 - Doesn't provide info on exported packages
- This will be supported by Bundlor

Dependency Management – Development Time



- Use tooling support for managed classpath during development if available
 - e.g. m2eclipse or Q4E for Maven with Eclipse
 - Requires removing Bundle Classpath Container!
- Otherwise, use dm Server tools or manage classpath yourself
 - dm Server tools give 1st level dependencies only
 - Doesn't work for integration tests
 - Will become available later this year
 - Tip for manual management:
prefer variables over hard-coded paths in Eclipse!

- Manifest can be edited manually
 - Doable for smaller bundles, esp. with tooling support
 - Becomes a burden with large ones
- Can use tooling to compute imports and exports
 - Bundlor will become our advised tool for this
 - Fully integrated with dm Server tools
 - Supports both Ant + Ivy and Maven
 - 'bnd' by Peter Kriens is another popular tool
- Be aware of runtime-only dependencies
 - Reflection requires manually adding extra imports

Topics in this Session



- Common Errors & Best Practices
- Modularizing your application
- Versioning
- Build and dependency management
- **Operational aspects**

Differs from 'normal' (non-OSGi) applications:

- Applications no longer self-contained
 - Don't package their own dependencies anymore
 - Might rely on services outside own application
- Without PARs, installation order usually matters
 - Dependencies between bundles
- Pinning issues prevent some applications from being deployed on the same server

- Release processes need to be in place
- Document dependencies
 - To ensure required bundles are in place
 - To ensure required services are available
 - Include transitive dependencies that might conflict with other applications because of pinning
- Document deployment ordering
- Good communication with ops team is crucial
 - Can't simply throw your application over the fence

- Ensure repository used in all stages is the same
 - Dev, test, acceptance, production
 - So you test against configuration used in production
- Might need to add bundles to local repository not used by your application
 - To reproduce production environment more closely
 - Ensures your code always uses the same bundles
- Good version ranges are important
 - Adding new bundles shouldn't break your application
 - Always provide upper bound for 3rd party libraries

- OSGi allows individual bundles to be installed, updated, replaced, uninstalled, etc. at runtime
 - Without restarting server
- Potential for upgrading without downtime
- Requires lots of testing though
 - Easy to introduce strange application state that's hard to reproduce
 - Remember difference between types and services!

- PARs typically upgraded as a whole
 - Individual bundles can be upgraded using JMX
 - Really intended for development only: use at own risk
- Some changes do require a restart
 - e.g. adding a missing transitive dependency
- Might be safer to restart after update
 - Diagnostic tools too low level (Equinox console) to analyze cause of errors resulting from hot deploy
 - Will improve with future releases

- OSGi changes how you work in many ways
- Important to realize what this requires
 - Must design and code for OSGi / dm-Server
- Not always clear-cut how to do that best
 - OSGi is a mature technology in itself
 - But its use in enterprise applications is cutting edge
- Both tooling and best practices are emerging
 - SpringSource dm Server is a leading product here!
 - SpringSource actively involved in providing feedback to OSGi community

Lab

dms-practices

SpringSource dm Server Roadmap

An overview of what future versions will
bring to the table

Topics in this Session



- Release schedule
- Features per release

Release Schedule

- Schedule as of January 2009:

Name	Release Date
Wight	current version
Jersey	May 2009
Orkney	October 2009
Hebrides	May 2010
Skye	Q3 2010

- Unified Provisioning
 - better performance at start-up and during application installation
 - optional transitive dependency resolution via live connection to SpringSource Enterprise Bundle Repository
- Optimized dependency management
 - **cloning**: single version of enterprise library can be wired to multiple versions of the libraries it depends on
 - solves both *pinning issue* and the use of *statics* (like in Log4J)
 - goes beyond what plain OSGi has to offer

- Web Modules
 - supports Spring Web 3.0 with "zero-conf" support
 - deploy without the need for any XML
 - web components automatically detected and registered based on annotations
 - web components automatically exposed via AMS and admin console
- Web Services Modules
 - Spring-WS Personality
 - web service components automatically exposed via AMS and admin console

- In-container Integration Testing
 - run code against same profile used by dm Server
- JNDI Support
 - Bridge between OSGi Service Registry and JNDI
- Global DataSource Configuration
 - Exposed through Service Registry and JNDI
- Exploded WAR file support
 - Like Tomcat
- Support for custom Tomcat Security Realms
- Basic distributed deployment and management functionality

- Pluggable Administration Console
 - centralized management console for administrators and operations staff
- "zero-conf" deployment options for many Spring application modules
- Improved diagnostics
 - esp. for modules that fail to successfully install
- New repository management tools
- Deployment of exploded applications

- Unified dependency management / build support
 - removes the need to duplicate dependency information across build files and application modules
- Improved Administration
 - full insight into Spring managed components inside applications
 - Spring beans, their wiring, configuration, and runtime metrics
 - easier to validate production configurations
 - easily start and stop any individual application or shared service without needing to uninstall

- Batch, web services, web, domain, and application services module support
- Batch:
 - schedule launch of batch jobs with operator access
 - batch components automatically exposed via AMS and admin console
- Full Clustering Support
 - easily deploy application to all nodes in a cluster
 - centralized management
- Grid, Cloud and Virtualization Support
 - feature set not final yet

-
- Additional support for operations, clustering and cloud
 - feature set not final yet

- SpringSource dm Server 1.0 provides stable foundation for future releases
- Many improvements and new features planned
 - Both for developers and administrators/operators
- Continuing the same programming model
 - Migration process will be painless

Securing the dm Server

How to secure your dm Server instance
and applications

Topics in this Session



- Security concerns
- Infrastructure
- JVM
- J2EE
- Server
- Application

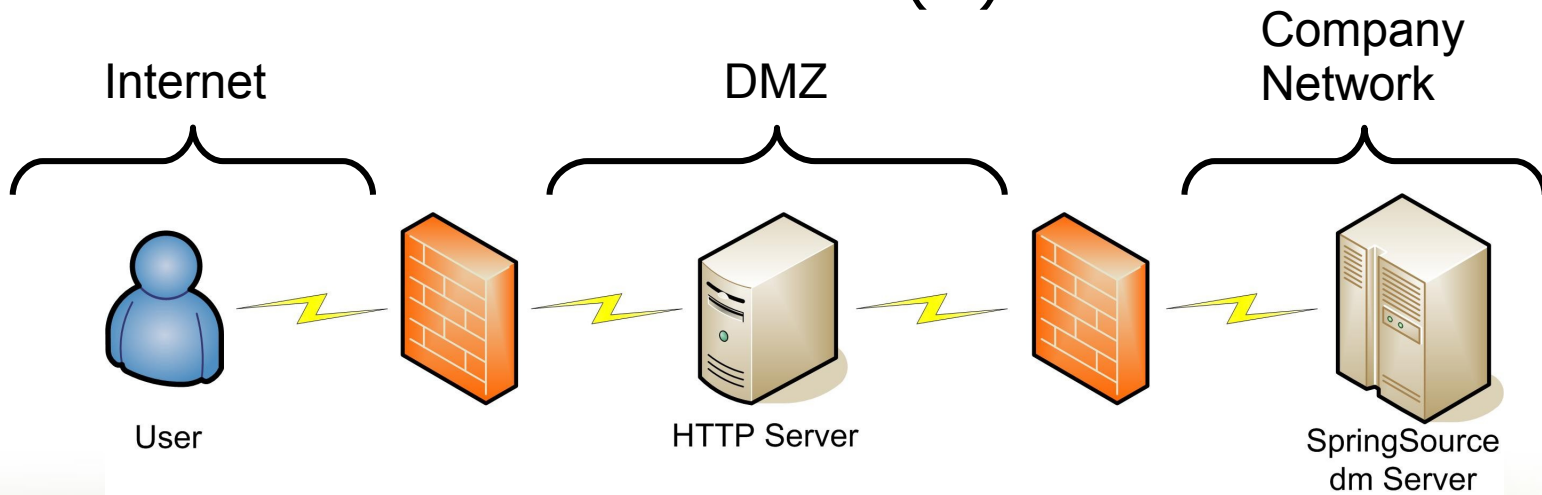
- Security covers many areas:
 - Infrastructure: firewalls, secure connections, etc.
 - JVM: J2SE Security Manager
 - J2EE: Role-based security as defined by Servlet spec
 - Server: Securing access to admin console, etc.
 - Application: Security managed by your application
- And that's only the technical aspects
 - Policies, procedures, contracts, etc. also crucial
- We'll go over technical areas in this presentation

Topics in this Session



- Security concerns
- **Infrastructure**
- JVM
- J2EE
- Server
- Application

- Firewalls and other network infrastructure are not dm Server specific
- dm Server typically placed in internal network behind firewall
- Communication over HTTP(S)



- What *is* configured in dm Server is the use of **SSL** for secure HTTP connections
- In config/servletContainer.config
- This is really just standard Tomcat Configuration
- Typically configured by administrator, not by developer

SSL Configuration Sample



```
{ "servletContainer": {  
  ...  
  "connectors": [  
    ... // regular HTTP and/or AJP Connector defined here  
    {  
      /* HTTPS Connector. Documentation at  
      * http://tomcat.apache.org/tomcat-6.0-doc/config/http.html  
      * and http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html */  
      "enabled": true,  
      "port": 8443,  
      "protocol": "HTTP/1.1",  
      "scheme": "https",  
      "connectionTimeout": 20000,  
      "maxThreads": 150,  
      "emptySessionPath": false,  
      "clientAuth": false,  
      "keystoreFile": "keystore",  
      "keystorePass": "changeit",  
      "secure": true,  
      "SSLEnabled": true,  
      "sslProtocol": "TLS"  
    }  
  ],  
  ...  
}
```

- dm Server ships with predefined keystore
 - Holds self-signed certificate
 - Password is *changeme*
 - For testing purposes only
- SSL Configuration out-of-scope for this course
 - For more info, check the following URL:
<http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html>

Topics in this Session



- Security concerns
- Infrastructure
- **JVM**
- J2EE
- Server
- Application

- J2SE defines a so-called security manager
- Can allow or deny code to do certain things
- Used by the standard OSGi Security specification
- dm Server 1.0 does not support running under a security manager
- So you cannot use OSGi security

- Intended to restrict bundles in what they can do
 - e.g., install new bundles or stop active bundles
- Esp. important in environments with untrusted code from external sources
 - Cellphones and other embedded devices
- Not that interesting in an application server
 - J2SE security hardly used in other servers as well
- So this is not a big issue

Topics in this Session



- Security concerns
- Infrastructure
- JVM
- **J2EE**
- Server
- Application

- The Enterprise Java specifications have security support built-in
- Role-based access of things like Servlets and EJBs
- Application declares roles
- Configuration done in server
 - e.g. mapping roles to LDAP groups
 - Non-portable, server-specific

- dm Server 1.0 only supports a single in-memory security realm
 - Configured in config/servlet/tomcat-users.xml
 - Cannot configure custom realms as in Tomcat yet
- Means Java EE security cannot really be used
 - Only for extremely simple use-cases
- Let application handle its own security
 - Typically using Spring Security
 - Has everything Java EE has to offer and more

Topics in this Session

- Security concerns
- Infrastructure
- JVM
- J2EE
- **Server**
- Application
- Conclusion

Securing The dm Server



- The default dm Server installation is insecure
 - Default password for the admin user
 - Equinox console enabled on port 2401
 - Starts remote JMX connector with default security settings
- Good for development, not for production
- So you'll have to *harden* the production server
- Let's see how

Change The Admin Password



- Configured in config/servlet/tomcat-users.xml
- Change 'springsource' to secure password
- Can change username as well
 - Keep the rolename
 - Prevents guessing password for known user

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="admin"/>
  <user username="admin" password="springsource" roles="admin"/>
</tomcat-users>
```

Disable Equinox Console

- Configured in config/server.config
- Simply change enabled value to false
- Alternative is to use firewall to restrict access to localhost only
 - Then use something like SSH to connect to server

```
{  
  ...  
  "osgiConsole": {  
    "enabled": true,  
    "port": 2401  
  }  
}
```

- Secured JSR-160 Remote JMX Connector always started by default startup scripts
 - Since dm Server 1.0.1
- But with default SSL certificate and user/password
 - See files in config/control directory
- Everyone can connect to and manage your server!

Configure secured connector yourself:

- Use `-jmxusers`, `-keystore` and `-keystorePassword` args
 - Allow for custom username/password and/or SSL certificates
- Or create your own startup script
 - using standard JVM System Properties
 - Use existing scripts as example
 - Consult JVM documentation for details

Topics in this Session



- Security concerns
- Infrastructure
- JVM
- J2EE
- Server
- **Application**
- Conclusion

- Security is the concern of your application
- Doesn't mean you have to code it yourself
- We advice to use *Spring Security*
 - Declarative security allows for separation of concerns
 - Secures URLs and method access
 - Supports Access Control Lists
 - Supports several mechanisms for authentication and authorization
 - Very flexible, mature framework

- dm Server 1.0 does not support J2SE Security managers and therefore OSGi security
- dm Server 1.0 only supports a single in-memory security realm for Java EE
- dm Server needs to be hardened before taken into production
- Security at the application level is best handled using Spring Security