OSGi
Declarative Services

An Overview, How-to,
and Handy Reference

OSGi is great!

OSGi is great because it enables:

- Decoupling: API-centric scoped wiring
  - Package import/export per bundle
  - Wiring done by the framework (or hooks)-- 3rd party
  - Substitutable exports..

- Service-oriented composition:
  - Centralized service registry for registration and discovery of services

- Dynamism: bundle/service lifecycle

Using (base) OSGi Services

(great! And also a pain in the neck)

- OSGi Services are registered using classes and properties:

```
Dictionary<String, ?> map;
map.put("service.vendor", "IBM");

ServiceRegistration reg = bundleContext.registerService(String,Object,Dictionary)
ServiceRegistration reg = bundleContext.registerService(String[],Object,Dictionary)
```

  - New generic interfaces for registering services in 4.3

```
ServiceRegistration reg = bundleContext.registerService(Class,S,Dictionary)
```

  - Use the **ServiceRegistration** object to change a service's active properties.

- **ServiceReference**: representation of service in service registry
  - Retrieve service references from the bundle context:
    - `getServiceReference(String), getServiceReference(Class)`
    - `getServiceReferences(String,String filter), getServiceReferences(Class,String filter)`
  - Obtain the service from the bundle context via the service reference:
    - `getService(ref), ungetService(ref)`
  - Manually manage dynamism and ref counting
    - `getService()` can return null
    - `getService()` must be followed by `ungetService()`

ServiceTracker

(improvement)

- Compendium service (provided by the framework)
- Tracks services matching conditions
  - Builds on the filters already present for finding service references
  - Extensible: special action when service is added/removed
    - `ServiceTrackerCustomizer` or direct extension of `ServiceTracker`
- Manages dynamism and ref counting
  - Get current service: `getService()`
  - Uses cached reference to requested reference (while valid)

But: eager!

Tracked service(s) that match the filter will be eagerly instantiated

ServiceTracker Example

Given this in a BundleActivator:

```
ServiceTracker<Widget, Widget> tracker;
BundleContext context;
```

You would do something like this in the `start(BundleContext)` method:

```
tracker = new ServiceTracker<Widget, Widget>(context, Widget.class, null);
tracker.open();
```

As soon as the open method is called, the service tracker eagerly tracks all registered instances of the service. Services are eagerly tracked until the service is closed:

```
tracker.close();
```

When an instance of the service is needed:

```
Widget instance = tracker.getService();
Widget[] instances = tracker.getServices();
```

An instance retrieved this way should be held for the least possible time:

- get the reference close to where you need it, and
- don't cache the return value (the tracker caches the value internally).

Declarative Services

(are we there yet?)

- Declarative model for service publish/find/bind
- Injects dependencies according to declared requirements
  - Simplified programming model
  - Event-based push via bind/unbind
  - Pull-when-needed via `ComponentContext.locateService()`
- ***Supports lazy initialization of services***
  - Unused classes/services not loaded/created
  - Improved startup
  - Reduced memory footprint

Service Component

- Normal Java class in a bundle: No required API
- Methods found/invoked via reflection
- XML component definition (generated via bnd)
- MANIFEST.MF header – Service-Component

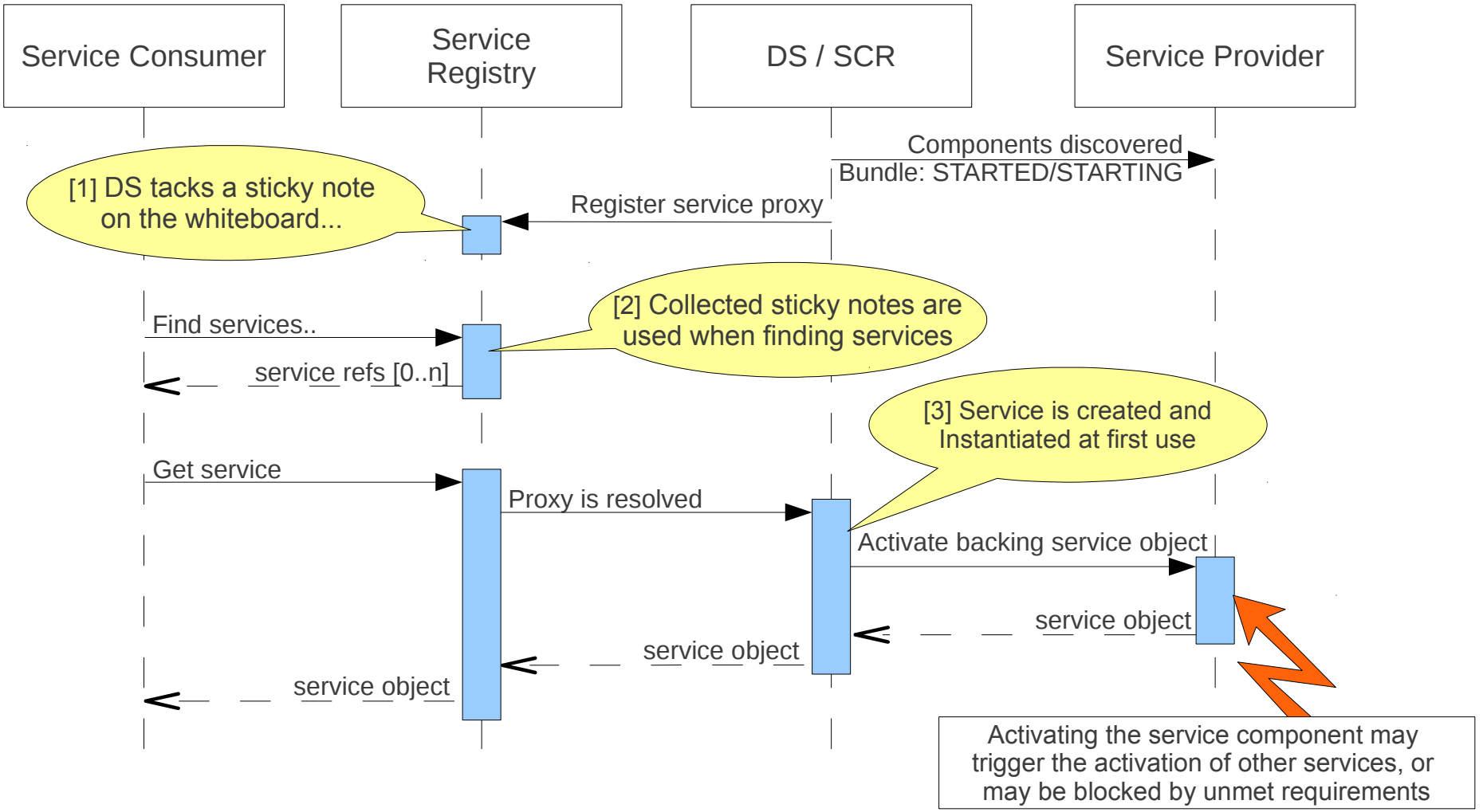
Lifecycle	activate, deactivate modified
Services	bind, unbind optional, multiple, dynamic
Activation Policy	Immediate vs. delayed factory
Configuration Policy	optional, require, ignore

Service Component Lifecycle

Component lifecycle is bounded by, but independent of, the host bundle lifecycle

1. Bundle (lazily) started
2. DS reads component definition (XML)
3. If the component provides a service, DS registers a proxy
4. Component lifecycle bound by component requirements while bundle is running
5. Bundle is stopped: DS deactivates active components

Service Component Activation



Providing Services

(Activation policy: immediate vs. delayed)

- Service Components can provide 1 or more services
- By default, a component that provides services is delayed
  - **DS registers a proxy into the service registry**
  - Real component is not created/activated until another service needs it
  - Must override the activation policy to force immediate instantiation

  Additional trivia:
  - The provided service does not have to be an interface
  - The provided service does not have to be in a package exported from the bundle (depending on expected consumer..)

Creating a Component

- Name: Unique within bundle
  - If used with ConfigAdmin, must be a valid PID (unique to framework)
    - Best practice: PIDs should be fully qualified and consumable
  - Qualify bundle-specific names in some way
    - <bundle symbolic name>.ds.<shortName>
    - While not official “external”, components are viewable via standard OSGi tools.

- Implementation: fully qualified class name
  - Best practice: bundle-private class (package protected or not exported)

Component properties

Properties defined in service definition can be merged (in order of precedence):

1. Dictionary passed to `ComponentFactory.newInstance()` (factory only)
2. ConfigAdmin-provided Configuration object
3. Properties from component definition
   (those specified later override those specified earlier)

All properties propagated to service properties except for .dotNames

Example: Widget

Bnd generates the MANIFEST.MF and packages bundles
using more powerful/readable directives (remember what MANIFESTs really look like)

Bnd shorthand

```
Service-Component=test.sample.widget;\
  implementation:=test.sample.internal.WidgetImpl;\
  provide:=test.sample.Widget;\
  configuration-policy:=require;\
  properties:="service.vendor=IBM,color=red"
```

component.xml

```
<?xml version='1.0' encoding='utf-8' ?>
 <scr:component name='test.sample.widget' ... configuration-policy='require'>
   <implementation class='test.sample.internal.WidgetImpl'>
   <service>
     <provides interface='test.sample.Widget'/>
   </service>
   <property name='service.vendor'>IBM</property>
   <property name='color'>red</property>
 </component>
```

Example: Widget – code

Super simple:

```
package test.sample.internal;

import ...

public class WidgetImpl implements Widget {

    protected void activate(Map<String, Object> props) {
        System.out.println("Pretty colors! " + props);
    }

    protected void deactivate(int reason) {
        System.out.println("Booo. " + reason);
    }
}
```

Use the Map signature for activate/deactivate to avoid dealing with Dictionary

Lifecycle methods invoked via reflection can not be private, should not be public, either!

If you need to know why the Component was stopped, use a signature variant that will tell you.

Component Activation

If we started a bundle containing the Widget component as defined, it would do nothing. Why?

- Instantiated and activated IFF conditions are satisfied activate()
  - Component is enabled
  - All required services are present
  - Configuration policy: Required Configuration available from ConfigurationAdmin
  - Component is "immediate", or is needed by another component
- Deactivated if/when conditions become unsatisfied deactivate()
  - Component is disabled, or bundle is stopped
  - Required service is stopped
  - Configuration policy: Required Configuration deleted
  - Service provided by component is no longer needed
  - Note: Object is deactivated before required services are unbound

Configuration Policy

- Not all components are configurable (have customizable attributes)
- ConfigurationAdmin is linked to the component lifecycle by `configuration-policy`

require	Configuration required for component activation
optional	Configuration will be used if available (default)
ignore	Configuration not expected and will be ignored

- A declared modified method prevents the component from being recycled when its configuration changes

Example: Widget – Config

Where does the widget's required config come from?
(remember merging component properties?)

- Component name is test.sample.widget
- DS looks for Configuration(s) using the component name
  - service.factoryPid → creates a unique component instance for each configuration
  - service.pid → applies the (merged) configuration to a single component instance
- Configuration objects for services are merged config
  - Some properties from component definition (like service.vendor)
  - Some from metadata definition: pid vs. factoryPid
  - The rest from the user or defaults

- Note: the config is required but we have no defined modified method.
  - If the config is changed, the component will be stopped and restarted.

Example: Widget – Metatype

metatype.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<metatype:MetaData xmlns:metatype="http://www.osgi.org/xmlns/metatype/v1.1.0"
                   localization="OSGI-INF/l10n/metatype">

    <!-- The OCD id must match the Designate's Object ocdref below -->
    <OCD id="test.sample.widget.factory.metatype" description="%widget.desc" name="%widget.name" >

        <!-- config.alias defines a short-form name to be used in server.xml -->
        <AD name="%config.alias" description="%config.alias.desc"
            id="config.alias" required="false" type="String" default="widget" />

        <!-- required widget color property, default is true -->
        <AD name="%color.name" description="%color.desc"
            id="color" type="String" required="true" default="blue" />

    </OCD>

    <!-- For factory pattern (many configurations, each creating a component instance),
         only the factoryPid should be specified, and it should match the pid specified
         in the DS component declaration in the bnd file...  -->
    <Designate factoryPid="test.server.widget">
        <Object ocdref="test.sample.widget.factory.metatype" />
    </Designate>

</metatype:MetaData>
```

Component Activation – still no dice

If we started a bundle containing the Widget component as defined, it would still do nothing. Why?

- Instantiated and activated IFF conditions are satisfied activate()
  - Component is enabled
  - All required services are present
  - Config policy: Required configuration available from CA
  - Component is "immediate", or is needed by another component
- Deactivated if/when conditions become unsatisfied deactivate()
  - Component is disabled, or bundle is stopped
  - Required service is stopped
  - Config policy: Required configuration deleted
  - Service provided by component is no longer needed
  - Note: Object is deactivated before required services are unbound

Example: NeedsOneWidget

This component provides no services, it is immediate by default.

Bnd shorthand

```
Service-Component=test.sample.refs.NeedsOneWidget;\
  implementation:=test.sample.refs.internal.NeedsOneWidget;\
  widget=test.sample.Widget;\
  properties:="service.vendor=IBM"
```

component.xml

```
<?xml version='1.0' encoding='utf-8' ?>
 <scr:component name='test.sample.refs.NeedsOneWidget' xmlns:='...'>
   <implementation class='test.sample.refs.internal.NeedsOneWidget'/>
   <property name='service.vendor'>IBM</property>
   <reference name='widget'
              interface='test.sample.widget'
              bind='setWidget' unbind='unsetWidget' />
 </component>
```

Reference Policy

- Static policy: don't (handle dynamism)
  - Component deactivated when bound service changes
  - A new instance may be created, however!
  - Might be expensive:
    - if target services change often,
    - or initialization is expensive...
  - *Static policy not useful with multiple cardinality*
- Dynamic policy (dynamic reference):
  - Bound service(s) can be changed while component is active
  - Beware: efficient, but concurrent..

Reference Cardinality

More examples at the end...

- Declared references have cardinality
  - Unary: use highest service.ranking, lowest service.id
    - 0..1 : optional
    - 1..1 : (at most 1) required for activation (default)
  - Multiple: bind called once for each bound service
    - 0..n : optional
    - 1..n : at least 1 required for activation

References

ServiceReference X for component Y
Assume we start with X1 for required services

- Bind and unbind methods can occur at different times and in different orders depending on how references are used.
- Static policy for service X:
  - setx(X1) called before activate() if service is required
  - Y.deactivate() method is called before unsetx(X1) if X1 was bound
- Dynamic
  - Required service
    - setx(X1) called before activate()
    - If X1 is going to be deactivated, but X2 is available, then
      - DS will call y.setx(X2) before y.unsetx(X1)
      - This allows the required/dynamic reference to remain satisfied.
    - Otherwise, as above for required reference.
  - Optional service
    - Set and unset can happen at any time, however: any still-bound services will be unbound after deactivate

Custom reference modifiers

Bnd shorthand

```
Service-Component:\
 test.sample.refs.NeedsOneWidget;\
  implementation:=test.sample.refs.internal.NeedsOneWidget;\
  widget=test.sample.Widget;\
  properties:="service.vendor=IBM"
scReferenceModifiers:
 test.sample.refs.NeedsOneWidget;\
  widget='bind=addWidget,unbind=removeWidget,udpated=widgetUpdated'
```

component.xml

```
<?xml version='1.0' encoding='utf-8' ?>
  <scr:component name='test.sample.refs.NeedsOneWidget' xmlns:='..felix..'>
    <implementation class='test.sample.refs.internal.NeedsOneWidget'/>
    <property name='service.vendor'>IBM</property>
    <reference name='widget'
               interface='test.sample.widget'
               bind='addWidget' unbind='removeWidget' updated='widgetUpdated'/>
</component>
```

- scReferenceModifiers header allows tweaking of the reference elements
  - Can choose bind/unbind names (e.g. want to call an existing tWAS method on unbind that doesn't have an unsetX name - happens in transactions)
  - Can add felix-ds specific "updated" method
    - If the test.sample.Widget service changes, then the method will be called on the test.sample.refs.internal.NeedsOneWidget class
    - Signature: protected void widgetUpdated(test.sample.Widget, Map<?,?> config)
- No validation at present, so as before... Always check the generated XML

Dynamic Reference Utilities

- Kernel service bundle utilities:
  - Reference validity tied to component lifecycle
  - Find/bind/cache service reference(s) lazily
- AtomicServiceReference<Service>
- ConcurrentServiceReferenceSet<Service>
- ConcurrentServiceReferenceMap<String, Service>
- ConcurrentServiceReferenceSetMap<String, ConcurrentServiceReferenceSet<Service>>

Rules of the road

- Avoid lifecycle confusion
  - Mixing and matching between DS & others works
  - BUT, consider expectations for validity of reference
    - Reference obtained via DS is as valid as the component is
    - References obtained from the BundleContext have a different lifecycle, with different (and sometimes conflicting) management cost
- DS is preferred to ServiceTracker wherever feasible
- Investigate the Bundle-ActivationPolicy header for your bundle for real laziness...

Service Component Definition

(bnd → XML)

- Bnd constructs the bundle manifest.
- Syntax for lists in the OSGi Bundle manifest:
  - Lists are comma separated, with semicolons delimiting an element and optional attributes
  - Element1;attribute=value, element2;version=[2.0,3.0), …
- Service-Component is a specified OSGi manifest header
  - *Service-Component: <comma-separated list of component definitions>*
  - Bnd short-hand component definitions are elements in that list
  - Bnd parses a short-hand element, generates the component xml file in the OSGI-INF directory, and replaces the element in the list with the component xml file name (rinse and repeat...)
- Other syntax rules:
  - Use \ for line continuation (no trailing space)
  - Only use = for assigning references, everything else should use :=
  - Bnd has shorthand for optional/multiple/dynamic (*, +, ?) – I avoid as too obscure
- Good advice:
  - Always check the generated XML when you're twiddling with the component definition.
  - Double check package and class names.
  - Triple check package and class names.

Lifecycle methods

Activate

- void <method-name>(<arguments>);
  - ComponentContext
  - BundleContext
  - Map
- void <method-name>(ComponentContext);
- void <method-name>(BundleContext);
- void <method-name>(Map);
- void <method-name>(two or more from list);
  - Implementor's discretion to pick one if multiple methods match the rule
- void <method-name>();

Deactivate

- void <method-name>(<arguments>);
  - ComponentContext
  - BundleContext
  - Map
  - Integer or int (reason for deactivation)
- void <method-name>(ComponentContext);
- void <method-name>(BundleContext);
- void <method-name>(Map);
- void <method-name>(int);
- void <method-name>(Integer);
- void <method-name>(two or more from list);
  - Implementor's discretion if multiple methods match the rule
- void <method-name>();

More Lifecycle methods

bind and unbind

- void <method-name>(ServiceReference);
  - ServiceReference to the bound service
  - Use ServiceReference with locateService(String,ServiceReference);
    - String is the “name” of the reference in component.xml
- void <method-name>(<parameter-type>);
  - Activated service object
  - Type of service object is verified
- void <method-name>(<parameter-type>, Map);
  - Activated service object AND service properties
  - Type of service object is verified

modified

- Identical to activate
- Method will be called when component configuration changes

Notes on deactivation (from the spec):

Deactivating a component configuration consists of the following steps:
1 Call the deactivate method, if present.
2 Unbind any bound services.
3 Release all references to the component instance and component context.
A component instance must complete activation or modification before it can be deactivated. A component configuration can be deactivated for a variety of reasons. The deactivation reason can be received by the deactivate method. The following reason values are defined:
0 – Unspecified.
1 – The component was disabled.
2 – A reference became unsatisfied.
3 – A configuration was changed.
4 – A configuration was deleted.
5 – The component was disposed.
6 – The bundle was stopped.
Once the component configuration is deactivated, SCR must discard all references to the component instance and component context associated with the activation.

Example: DynamicNeedsOneWidget

Bnd shorthand

```
Service-Component=test.sample.refs.DynamicNeedsOneWidget;\
 implementation:=test.sample.refs.internal.DynamicNeedsOneWidget;\
 widget=test.sample.Widget;\
 dynamic:='widget';\
 properties:="service.vendor=IBM"
```

component.xml

```
<?xml version='1.0' encoding='utf-8' ?>
 <scr:component name='test.sample.refs.DynamicNeedsOneWidget' … >
  <implementation
              class='test.sample.refs.internal.DynamicNeedsOneWidget'/>
   <property name='service.vendor'>IBM</property>
   <reference name='widget'
              interface='test.sample.widget'
              policy='dynamic'
              bind='setWidget' unbind='unsetWidget' />
 </component>
```

Example: MaybeOneWidget

Bnd shorthand

```
Service-Component=test.sample.refs.MaybeOneWidget;\
 implementation:=test.sample.refs.internal.MaybeOneWidget;\
 widget=test.sample.Widget;\
 optional:='widget';\
 properties:="service.vendor=IBM"
```

component.xml

```
<?xml version='1.0' encoding='utf-8' ?>
 <scr:component name='test.sample.refs.MaybeOneWidget' xmlns:='...'>
   <implementation class='test.sample.refs.internal.MaybeOneWidget'/>
   <property name='service.vendor'>IBM</property>
   <reference name='widget'
              interface='test.sample.widget'
              cardinality='0..1'
              bind='setWidget' unbind='unsetWidget' />
 </component>
```

Example: DynamicMaybeOneWidget

Bnd shorthand

```
Service-Component=test.sample.refs.DynamicNeedsOneWidget;\
 implementation:=test.sample.refs.internal.DynamicNeedsOneWidget;\
 widget=test.sample.Widget;\
 optional:='widget';\
 dynamic:='widget';\
 properties:="service.vendor=IBM"
```

component.xml

```
<?xml version='1.0' encoding='utf-8' ?>
 <scr:component name='test.sample.refs.DynamicNeedsOneWidget' … >
   <implementation
              class='test.sample.refs.internal.DynamicNeedsOneWidget'/>
   <property name='service.vendor'>IBM</property>
   <reference name='widget'
              interface='test.sample.widget'
              cardinality='0..1'
              policy='dynamic'
              bind='setWidget' unbind='unsetWidget' />
 </component>
```

Example: AtLeastOneWidget

Bnd shorthand

```
Service-Component=test.sample.refs.AtLeastOneWidget;\
 implementation:=test.sample.refs.internal.AtLeastOneWidget;\
 widget=test.sample.Widget;\
 multiple:='widget';\
 dynamic:='widget';\
 properties:="service.vendor=IBM"
```

component.xml

```
<?xml version='1.0' encoding='utf-8' ?>
 <scr:component name='test.sample.refs.AtLeastOneWidget' … >
   <implementation class='test.sample.refs.internal.AtLeastOneWidget'/>
   <property name='service.vendor'>IBM</property>
   <reference name='widget'
              interface='test.sample.widget'
              cardinality='1..n'
              policy='dynamic'
              bind='setWidget' unbind='unsetWidget' />
 </component>
```

Example: MaybeSomeWidgets

Bnd shorthand

```
Service-Component=test.sample.refs.MaybeSomeWidgets;\
 implementation:=test.sample.refs.internal.MaybeSomeWidgets;\
 widget=test.sample.Widget;\
 multiple:='widget';\
 optional:='widget';\
 dynamic:='widget';\
 properties:="service.vendor=IBM"
```

component.xml

```
<?xml version='1.0' encoding='utf-8' ?>
 <scr:component name='test.sample.refs.MaybeSomeWidgets' … >
   <implementation class='test.sample.refs.internal.MaybeSomeWidgets'/>
   <property name='service.vendor'>IBM</property>
   <reference name='widget'
              interface='test.sample.widget'
              cardinality='0..n'
              policy='dynamic'
              bind='setWidget' unbind='unsetWidget' />
 </component>
```