Building OSGi Components

Carsten Ziegeler | cziegeler@apache.org

ApacheCon NA 2014

About
cziegeler@apache.org @cziegeler

- RnD Team at Adobe Research Switzerland
- Member of the Apache Software Foundation
  - Apache Felix and Apache Sling (PMC and committer)
  - And other Apache projects
- OSGi Core Platform and Enterprise Expert Groups
- Member of the OSGi Board
- Book / article author, technical reviewer, conference speaker

Agenda

- 1 OSGi Service Registry
- 2 Components
- 3 Declarative Services Today
- 4 Next version of Declarative Services

Component and Service

- Component
  - Piece of software managed by a (component) container
  - Java: instances created and managed by a container
  - Container provides configuration and used services

Component and Service

- Service
  - A component providing a service
  - Java:
    - Defined through an interface
    - A component implementing one or more interfaces (= services)
  - Usable by components and other services
    - Clients act on the service (interface)

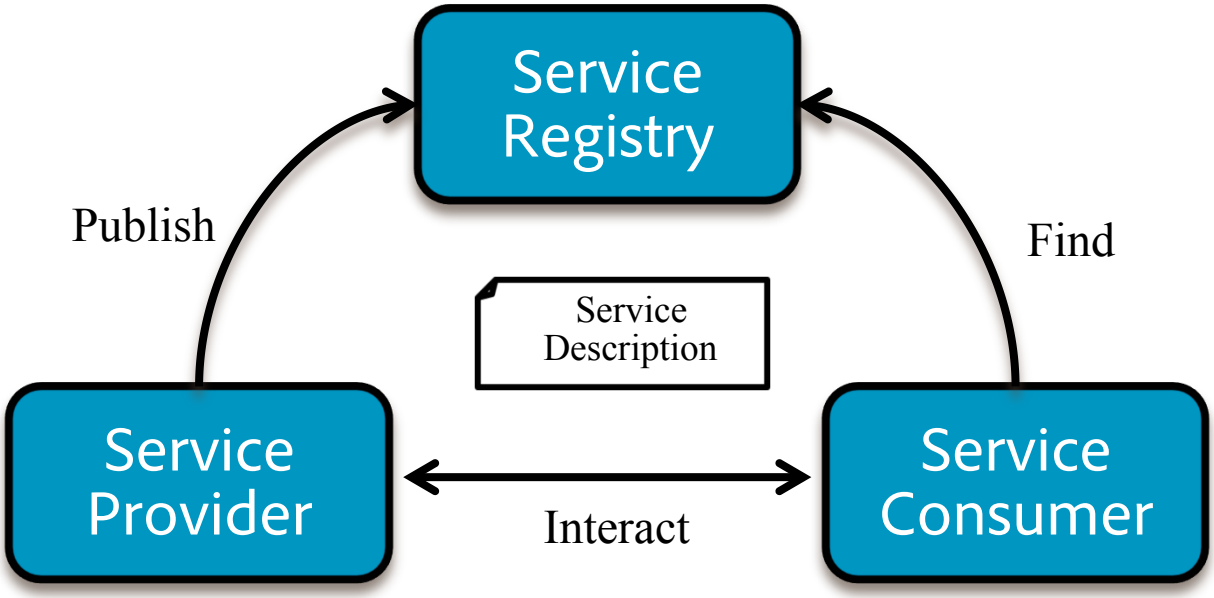
Foreword

- Many component frameworks for OSGi exist today
  - Difficulty of choosing
- For OSGi based component development it's more important to focus on the components than on the components framework
- Focus is on developing components
  - Developers choice
- Declarative Services is very good but it's not the only solution

1 Service Registry

OSGi Service Registry

- Service oriented architecture
  - Publish/find/bind

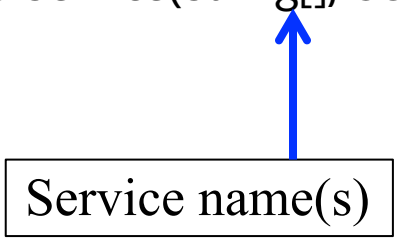


Registering a Service

- Each bundle has access to its bundle context object
  - Using bundle activator
- Bundle context:
  - registerService(String, Object, Dictionary)
  - registerService(String[], Object, Dictionary)

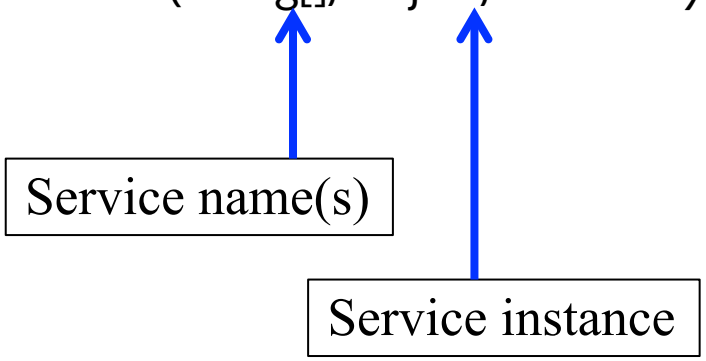
Registering a Service

- Each bundle has access to its bundle context object
  - Using bundle activator
- Bundle context:
  - registerService(String, Object, Dictionary)
  - registerService(String[], Object, Dictionary)



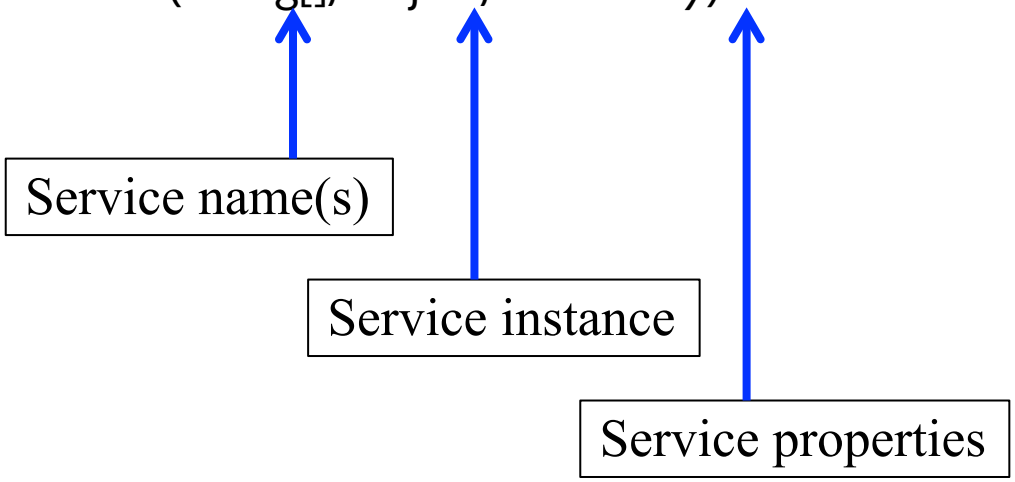
Registering a Service

- Each bundle has access to its bundle context object
  - Using bundle activator
- Bundle context:
  - registerService(String, Object, Dictionary)
  - registerService(String[], Object, Dictionary)



Registering a Service

- Each bundle has access to its bundle context object
  - Using bundle activator
- Bundle context:
  - registerService(String, Object, Dictionary)
  - registerService(String[], Object, Dictionary)



Registering a Service

- Each bundle has access to its bundle context object
  - Using bundle activator
- Bundle context:
  - registerService(String, Object, Dictionary)
  - registerService(String[], Object, Dictionary)

```
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceRegistration;

…
BundleContext bc = …;

final Dictionary<String, Object> props = new Hashtable<String, Object>();
props.put(Constants.SERVICE_DESCRIPTION, "Greatest Service on Earth");
props.put(Constants.SERVICE_VENDOR, "Adobe Systems Incorporated");

final Scheduler service = new MyScheduler();
this.bundleContext.registerService(
    new String[] {Scheduler.class.getName()},
    service, props);
```

Getting a Service from the Service Registry

```
BundleContext bundleContext = ...;

final ServiceReference sr = bundleContext.getServiceReference(
                                Scheduler.class.getName());

if ( sr != null ) {
    final Scheduler s = (Scheduler) bundleContext.getService(sr);

    if ( s != null ) {
        s.doSomething();
    }
    bundleContext.ungetService(sr);
}
```

Getting Service Properties

```
BundleContext bundleContext = ...;

final ServiceReference sr = bundleContext.getServiceReference(
                                Scheduler.class.getName());

if ( sr != null ) {
    // access properties
    final Object value = sr.getProperty(Constants.SERVICE_VENDOR);

    bundleContext.ungetService(sr);
}
```

Service Properties

```
import org.osgi.framework.Constants;

Constants.SERVICE_ID                    - set by the framework (long)
                                          id of the service
                                          increased for each registration
                                          dynamic - not persisted!

Constants.SERVICE_DESCRIPTION           - optional description (string)

Constants.SERVICE_VENDOR                - optional vendor (string)

Constants.SERVICE_PID                   - persistence identifier (string)
                                          optional, unique identifier

Constants.SERVICE_RANKING               - ordering of registrations
```

Multiple Registrations for a Service

```
BundleContext bundleContext = ...;
final ServiceReference[] refs = bundleContext.getServiceReferences(
                                Scheduler.class.getName(),
                                null);

if ( refs != null ) {
    // iterate over references, maybe sort by ranking etc.

}
```

Getting a Service from the Service Registry

```
BundleContext bundleContext = ...;

final ServiceReference sr = bundleContext.getServiceReference(
                                Scheduler.class.getName());

if ( sr != null ) {
    final Scheduler s = (Scheduler) bundleContext.getService(sr);

    if ( s != null ) {
        s.doSomething();
    }
    bundleContext.ungetService(sr);
}
```

Highest Ranking

Lazy Service Creation / Bundle Scope

- Register service factory instead of service
  - Framework calls factory once per client bundle

```
public interface org.osgi.framework.ServiceFactory {

    Object getService(Bundle bundle,
                      ServiceRegistration registration);

    void ungetService(Bundle bundle,
                      ServiceRegistration registration,
                       service);
}
```

Registering a Service Factory

```
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceRegistration;

…
BundleContext bc = …;

final Dictionary<String, Object> props = new Hashtable<String, Object>();
props.put(Constants.SERVICE_DESCRIPTION, "Greatest service on Earth");
props.put(Constants.SERVICE_VENDOR, "Adobe Systems Incorporated");

final ServiceFactory factory = new MySchedulerFactory();
this.bundleContext.registerService(
    new String[] {Scheduler.class.getName()},
    factory, props);
```

Service Event Listener

- Notification of registration / unregistrations
- Registered to the bundle context
  - Filter for service name, properties etc.

```
package org.osgi.framework;
public interface ServiceListener extends EventListener {
    void serviceChanged(ServiceEvent event);
}
```

OSGi Service Registry

- Lightweight services
  - Lookup is based on interface name
  - Direct method invocation
  - Scopes: singleton, bundle, prototype (R6)
- Good design practice
  - Separates interface from implementation
  - Separates registration from usage
  - Enables reuse, substitutability, loose coupling, and late binding

2 Components

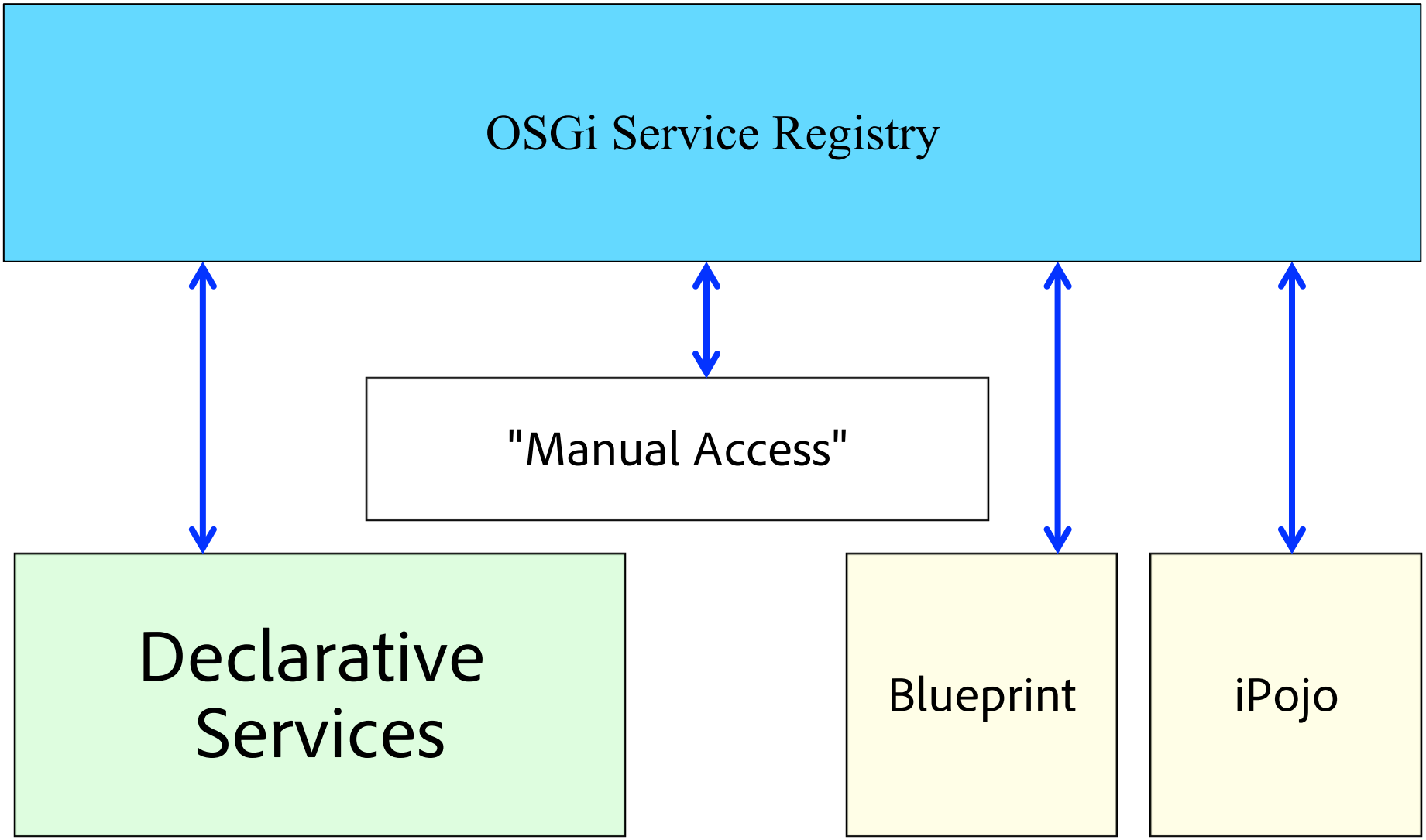
OSGi Service Registry

- Powerful but "complicated" to use directly
- Requires a different way of thinking
- Dynamic
  - Packages/Bundles might come and go
  - Services might appear/disappear
- Manually resolve and track services
- Doable, but requires "work"

Components and Services with OSGi

- Service interface
  - Public (if exported for other bundles)
  - Versioned through package version (Semantic versioning)
  - Private for internal services (sometimes useful)
- Component / service implementation
  - Always private

Component Container Interaction



Advanced OSGi Development Solutions

- Service Tracker
  - Still somewhat of a manual approach
- Declarative Services, Blueprint, *iPOJO*
  - Declarative
  - Sophisticated service oriented component frameworks
  - Automated dependency injection and more
  - More modern, POJO oriented approaches
- Straight forward with Declarative Services, Annotations, Maven/Ant/Bndtools...

3 Declarative Services I

Component Development with Declarative Services

- Declarative Services (OSGi Compendium Spec)
  - Defines Service Component Runtime (SCR)
  - Apache Felix SCR Annotations (DS annotations)
  - Available tooling: Maven/Ant/Bndtools...
- Some advantages (in combination with the tooling)
  - POJO style
  - Declarative
  - Single source: just the Java code, no XML etc.
  - "Integration" with Configuration Admin and Metatype Service

My First Component

```
package com.adobe.osgitraining.impl;

import org.apache.felix.scr.annotations.Component;

@Component
public class MyComponent {

}
```

Component Lifecycle

```java
package com.adobe.osgitraining.impl;

import org.apache.felix.scr.annotations.Activate;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Deactivate;

@Component
public class MyComponent {

    @Activate
    protected void activate() {
        // do something
    }

    @Deactivate
    protected void deactivate() {
        // do something
    }
}
```

Providing a Service

```
package com.adobe.osgitraining.impl;

import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Service;
import org.osgi.service.event.EventHandler;

@Component
@Service(value=EventHandler.class)
public class MyComponent implements EventHandler {

        …
```

Providing Several Services

```
package com.adobe.osgitraining.impl;

import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Service;
import org.osgi.service.event.EventHandler;

@Component
@Service(value={EventHandler.class, Runnable.class})
public class MyComponent implements EventHandler, Runnable {

        …
```

Using a Service

```
package com.adobe.osgitraining.impl;

import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Service;
import org.osgi.service.event.EventHandler;

@Component
@Service(value=EventHandler.class)
public class MyComponent implements EventHandler {

    @Reference
    private ThreadPool threadPool;

        …
```

Using an optional Service

```
package com.adobe.osgitraining.impl;

import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Service;
import org.osgi.service.event.EventHandler;

@Component
@Service(value=EventHandler.class)
public class MyComponent implements EventHandler {

    @Reference(cardinality=ReferenceCardinality.OPTIONAL_UNARY,
               policy=ReferencePolicy.DYNAMIC)
    private ThreadPool threadPool;

    @Reference(cardinality=ReferenceCardinality.MANDATORY_UNARY)
    private Distributor distributor;
```

Component Properties -> Service Properties

```
import org.apache.sling.commons.osgi.PropertiesUtil;

@Component
@Service(value=EventHandler.class)
@Properties({
    @Property(name="service.vendor", value="Who?”),
    @Property(name="service.ranking", intValue=500)
})
public class DistributingEventHandler
    implements EventHandler {
```

Configuration Admin

- OSGi Configuration Admin
  - "The" solution to handle configurations
  - Configuration Manager
  - Persistence storage
  - **Service API** to retrieve/update/remove configuration
- Integration with Declarative Services
  - Configuration changes are propagated to the components
  - Configurations are stored using the **PID**

Configuration – Supports Configuration Admin

```
import org.apache.sling.commons.osgi.PropertiesUtil;

@Component
@Service(value=EventHandler.class)
@Properties({
    @Property(name="event.topics", value="*", propertyPrivate=true),
    @Property(name="event.filter", value="(event.distribute=*)",
              propertyPrivate=true)
})
public class DistributingEventHandler
    implements EventHandler {

    private static final int DEFAULT_CLEANUP_PERIOD = 15;

    @Property(intValue=DEFAULT_CLEANUP_PERIOD)
    private static final String PROP_CLEANUP_PERIOD ="cleanup.period";

    private int cleanupPeriod;

    @Activate
    protected void activate(final Map<String, Object> props) {
        this.cleanupPeriod =
                PropertiesUtil.toInteger(props.get(PROP_CLEANUP_PERIOD));
    }
```

Configuration Update

```
import org.apache.sling.commons.osgi.OsgiUtil;

public class DistributingEventHandler
    implements EventHandler {

    …

    @Modified
    protected void update(final Map<String, Object> props) {
        this.cleanupPeriod =
                PropertiesUtil.toInteger(props.get(PROP_CLEANUP_PERIOD));
    }
```

Without update:
Component is restarted on
config change!

Configuration – Supports Configuration Admin

- Provided map contains
  - Configuration properties from Configuration Admin
  - Defined component properties

```
@Activate
protected void activate(final Map<String, Object> props) {
    …
}
```

Metatype and Web Console

- OSGi Metatype Service
  - Description of bundle metadata
  - Description of service configurations
  - Property type, name, and description
- Apache Felix Web Console
  - Great solution to configure the system
  - Especially component configurations
  - Uses metatype description

Configuration – Supports Metatype

```
import org.apache.sling.commons.osgi.PropertiesUtil;

@Component(metatype=true, label="Distributing Event Handler",
           description="This handler is awesome.")

@Properties({
    @Property(name="event.topics", value="*", propertyPrivate=true)
})
public class DistributingEventHandler
    implements EventHandler {

    private static final int DEFAULT_CLEANUP_PERIOD = 15;

    @Property(intValue=DEFAULT_CLEANUP_PERIOD,
              label="Cleanup Period",
              description="This is the cleanup period in seconds.")
    private static final String PROP_CLEANUP_PERIOD ="cleanup.period";
```

Lifecycle Methods

- Signatures for activate and deactivate:

```
protected void activate();

protected void activate(final Map<String, Object> properties);

protected void activate(final ComponentContext cc);

protected void activate(final BundleContext cc);

protected void activate(final Map<String, Object> properties,
                        final ComponentContext cc);

protected void activate(final Map<String, Object> properties,
                        final BundleContext cc);
```

Declarative Services

- A service is by default only started if someone else uses it!
  - Lazy is always good and usually sufficient!
  - Immediate flag on @Component forces a service start (use with care!)
- References are always bound through methods
  - SCR Plugin generates methods for unary references at built time

Unary References - Revisited

```
package com.adobe.osgitraining.impl;

import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.Service;
import org.osgi.service.event.EventHandler;

@Component
@Service(value=EventHandler.class)
public class MyComponent implements EventHandler {

    @Reference
    private Distributor distributor;

    protected void bindDistributor(Distributor d) {
        this.distributor = d;
    }

    protected void unbindDistributor(Distributor d) {
        if ( this.distributor == d ) {
            this.distributor = null;
        }
    }
```

Generated

Generated

References to Multiple Services

- Create bind / unbind methods

```
@Reference(name="AdapterFactory",
           referenceInterface=AdapterFactory.class,
           cardinality=ReferenceCardinality.OPTIONAL_MULTIPLE,
           policy=ReferencePolicy.DYNAMIC)
public class AdapterManagerImpl implements AdapterManager

    protected void bindAdapterFactory(ServiceReference reference) {
        // use component context to get the service
    }

    protected void bindAdapterFactory(AdapterFactory factory) {
    }

    protected void bindAdapterFactory(AdapterFactory factory,
                                      Map<String, Object> serviceProps) {
    }
```

Apache Felix SCR Tooling

- Combines everything (DS, Configuration Admin, Metatype, Maven/Ant)
- Annotation-based
- Single-source development = only java code
- Annotate components
  - Properties with default values and metatype info
  - Provided services
  - Services references (policy and cardinality)
- Generates DS XML
- Generates Metatype descriptors
- Generates Java code (for reference handling)
- Extensible by “annotation plugins”

Component Specification

- XML Configuration
  - Contained in bundle
  - Manifest entry pointing to config(s)
- Publishing services (through OSGi registry)
- Consuming services
- Reference policy (static,dynamic),
- Reference cardinality (0..1, 1..1, 0..n)
- Default configuration
- Service lifecycle management

Declarative Services

- Reads XML configs on bundle start
- Registers services (service factories)
- Keeps track of dependencies
  - Starts/stops services
- Invokes optional activation and deactivation method
  - Provides access to configuration
- Leverages OSGi service registry
  - Plays well with other component management approaches!

4 Declarative Services II

Configuring A Component

- Today's problems
  - Property definitions are lengthy…
  - ..and scattered across the code…
  - Conversion of configuration values
  - A lot of boilerplate code

Complex Sample

```
@Component
@Property(name="service.ranking", intValue=15)
public class MyComponent {

    private static final boolean DEFAULT_ENABLED = true;
    @Property(boolValue=DEFAULT_ENABLED)
    private static final String PROP_ENABLED = "enabled";

    @Property(value = {"topicA", "topicB"})
    private static final String PROP_TOPIC = "enabled";

    @Property
    private static final String PROP_USERNAME = "userName";

    String userName;

    String[] topics;

    @Activate
    protected void activate(final Map<String, Object> config) {
         final boolean enabled = PropertiesUtil.toBoolean(config.get(PROP_ENABLED),
                                        DEFAULT_ENABLED);
         if ( enabled ) {
             this.userName = PropertiesUtil.toString(config.get(PROP_USERNAME), null);
             this.topics = PropertiesUtil.toStringArray(config.get(PROP_TOPIC));
         }
    }
}
```

Define Configuration Annotation....

```
@interface MyConfig {

        boolean enabled() default true;
        String[] topic() default {"topicA", "topicB"};
        String userName();
        int service_ranking() default 15;

}
```

..and use in lifecycle method

```
@Component
public class MyComponent {

    String userName;

    String[] topics;

    @Activate
    protected void activate(final MyConfig config) {
        // note: annotation MyConfig used as interface
        if ( config.enabled() ) {
            this.userName = config.userName();
            this.topics = config.topic();
        }
    }
```

Or even simpler…

```
@Component
public class MyComponent {

    private MyConfig configuration;

    @Activate
    protected void activate(final MyConfig config) {
         // note: annotation MyConfig used as interface
         if ( config.enabled() ) {
            this.configuration = config;
         }
    }
}
```

In the works: Metatype Support (RFC 208)

```
@ObjectClassDefinition(label="My Component",
                       description="Coolest component in the world.")
@interface MyConfig {

    @AttributeDefinition(label="Enabled",
                         description="Topic and user name are used if enabled")
    boolean enabled() default true;

    @AttributeDefinition(...)
    String[] topic() default {"topicA", "topicB"};

    @AttributeDefinition(...)
    String userName();

    int service_ranking() default 15; // maps to service.ranking
}
```

Declarative Service Enhancements (RFC 190)

- Annotation Configuration Support
- Support for service scopes (prototypes)
- Introspection API

QnA