RFC 212 - Field Injection for Declarative Services

Draft

14 Pages

Abstract

The component model defined by Declarative Services is using a method based approach for injecting referenced services into the component. Compared to other component models this requires the developer to write the same boiler plate code for each and every reference. This RFC aims to provide a technical design to add field injection to Declarative Services..

This RFC focuses on field injection for Declarative Services.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

0.2 Trademarks

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

```
Source code is shown in this typeface.
```

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	04.07.14	Initial proposal – replace strategy Carsten Ziegeler (Adobe Systems Incorporated)
Update	17.07.14	Update Different strategies – event and replace Avoid type evaluation at runtime Carsten Ziegeler (Adobe Systems Incorporated)

1 Introduction

The component model defined by Declarative Services is using a method based approach for injecting referenced services into the component. Compared to other component models this requires the developer to write the same boiler plate code for each and every reference. This RFC aims to provide a technical design to add field injection to Declarative Services..

This RFC focuses on field injection for Declarative Services.

2 Application Domain

Declarative Services (chapter 112 in the OSGi specifications) defines a POJO programming model for OSGi services. While RFC 190 and RFC 208 aim at making component development with DS easier and try to reduce the amount of code to write, DS is using an event strategy based on method injection and therefore still requires the developer to implement bind/unbind/update methods for each and every reference. In most cases the code of these methods is always the same and usually simply updates a field in the component holding the referenced service. While the method provides a notification mechanism, too, this is rarely used.

The Apache Felix SCR Annotations and tooling based on these annotations provide an annotation to be used on a field holding a unary reference. The tooling generates byte code for a class holding such an annotation and adds the bind/unbind methods automatically, reducing the boiler plate code to be written by a component developer.

In other component models, like Apache Felix iPojo, CDI or the Spring Framework, field injection is very popular and field injection missing in DS has always been a larger criticism against DS.

DS supports four reference cardinality modes. In addition to supporting more than one reference, a reference can be optional or mandatory. That is, a reference can be satisfied with zero or one bound service. In addition, RFC 190 introduces the minimum cardinality property which allows to raise the specified minimum value to a higer number.

2.1 Terminology + Abbreviations

DS Declarative Services

POJO Plain old Java Object; term use for objects not implementing and framework specific plumbing such as Servlet API, Spring API, or OSGi API.

SCR Service Components Runtime; generally the implementation of the Declarative Services Specification; also the name of the Apache Felix implementation (Apache Felix SCR).

3 Problem Description

The current DS component model for handling references supports two different ways, the lookup strategy and the event strategy. When using the lookup strategy, a service is lookup through the `ComponentContext` each time it is used. The event strategy is based on implementing bind/unbind/update methods. The model describes when and in which order these methods are invoked. This depends on the cardinality of the reference (unary or multiple), whether the reference is mandatory and whether the reference is dynamic or static.

Field injection can be added to the model in two ways:

- By just defining a new annotation which is processed by tooling and the tooling enhances the class with corresponding method implementations. This is the approach the Apache Felix SCR tooling has taken and requires no changes to the DS specification.
- Adding field annotation as a first class citizen to the component model. This requires changes/additions to the DS spec, the XML schema, and the implementation. In addition an annotation needs to be defined. The benefit of this solution is that it does not depend on any specific tooling.

In contrast to method based injection, field injection moves (at least part of) the burden of proper synchronizing the access to the field to the implementation of field injection (either the DS implementation or the generated byte code). With method based injection, the burden lies solely on the component developer. Therefore field injection should make the life of the developer easier within the limitations of field injection.

4 Requirements

FID001 – The solution MUST provide a way to define field injection when developing DS components.

FID002 – The solution MUST support the same functionality as the reference handling through methods.

FID003 – The solution SHOULD outline the implications for the component developer with respect to thread safety concerns for accessing the value of the injected field.

FID004 – The solution SHOULD not be tied to Java 5+. It should be usable with lower Java versions.

5 Technical Solution

The technical solution proposes changes in DS, enhancing the XML schema and a new annotation for field injection.

As field injection provides the same functionality as method injection, most of the concepts from method injection can be reused as is, this includes defining the policy, the policy-option and the target filter. The solution for field injection provides the same options for the cardinality of a reference as method injection (0..1, 1..1, 0..n, 1..n) including raising the minimum cardinality as outlined by RFC190. The policy can either be dynamic or static and the policy option is either greedy or reluctant.

5.1 Supported Field Types

If a field references a service of type *SE* and *IN* is a type that is assignable from *SE*, the following types are supported for a field of cardinality unary:

- *IN*
- org.osgi.framework.ServiceReference
- org.osgi.framework.ServiceObjects
- java.util.Map for injecting the service properties (Map<String, Object>)
- java.util.Map.Entry<Map, IN> - the map contains the service properties (Map<String, Object>)
- java.util.Map.Entry<ServiceReferece, IN>

For a field reference of cardinality multiple different aggregate types of one of the unary types as defined above are supported:

- java.util.Collection
- java.util.List
- java.util.Set
- java.util.SortedSet
- any subtype of the above aggregate types if the event strategy is used for this field (see below)

Other field types are not supported. If a component is using an unsupported type, the component is not activated and the error situtation must be logged. Tooling might already detect the situation at build time and can issue an error to the developer.

At runtime, the DS implementation reads the component XML (see 5.4) and therefore gets the cardinality of the field and the type of the reference *SE*. If the cardinality is unary, the DS implementation can detect the type of the field through reflection. The XML also contains an information about the key type if Map.Entry is used as the type. For the rare case that the referenced service is one of ServiceReference, ServiceObjects, java.util.Map or java.util.Map.Entry and the same type is used for the field, only the service itself will be injected into the field.

For cardinality multiple the aggregate type is always a subtype of java.util.Collection, the XML contains the information about the aggregated type .

5.2 Field Injection Strategies

For fields of cardinality unary the `replace` strategy is always used. With this strategy the value of the field is replaced whenever changes regarding the referenced service occur. The field is set by DS in the same way and order as DS would call the methods for method injection:

- Instead of calling the bind method, the field is set to a value according to the used type
- Instead of calling the unbind method:
  - If the field has the same value as the unbound service or the value of the field is associated with the unbound service, the field is set to null
  - In all other cases the field is left untouched
- Instead of calling the updated method, the value of the field is updated if it contains the service properties either directly or through a `Map.Entry.` In the other cases, the field does not need to be updated as the value of the field does not change.

If the unary field reference has the policy dynamic, it must be declared as volatile. Otherwise other threads than the thread setting the field might never see an update of the field. If a component is using a non-volatile field for injection a dynamic unary reference, the component is not activate. This error must be logged. In addition, the tooling processing an annotated field should already signal an error for this situation.

For fields of type multiple two different strategies can be used: the `replace` and the `event` strategy. In the case of the event strategy for references of cardinality multple, the field is set once to the corresponding aggregate implementation (see below) and whenever changes to the set of referenced services occur, the collection is directly modified:

- If a developer is providing an implementation for the aggregate type, this needs to be done as part of the component object construction. If the field does not contain a value after constructing the object, it is set by the DS implementation right after the constructor is called. (More about the aggregate implementation in chapter 5.3.)
- Instead of calling the bind method, the add() method is called on the aggregate.
- Instead of calling the unbind method, the remove() method is called on the aggregate.
- ***TODO***: *how do we handle the update method, remove() followed by add() would be on option, optionally a method could be configured to be used?*

With the `replace` strategy, always a new immutable collection is created and set as the value of the field. The field is set by DS in the same way and order as DS would call the methods for method injection:

- Right after the component is constructed, the field is initialized with an empty collection. A value set by component code as part of construction the instance will be overwritten.
- Instead of calling the bind method, the field is set to a new collection including the new service.
- Instead of calling the unbind method, a new collection without the unbound service is set. If there is no matching service, an empty collection is set as the value.
- Instead of calling the updated method, for cardinality multiple the field is updated with a new collection if it is referencing the component properties.

If a field reference of cardinality multiple is using the replace strategy and has the policy dynamic, it must be declared as volatile. Otherwise other threads than the thread setting the field might never see an update of the field. If a component is using a non-volatile field for injection in this case, the component is not activate. This error must be logged. In addition, the tooling processing an annotated field should already signal an error for this situation.

5.3 Aggregate Types

For cardinality multiple the component developer can decide between providing an implementation for the aggregate type or lettting the DS implementation pick an implementation.

5.3.1 Replace Strategy

If the replace strategy is used, the DS implementation will pick the aggregate implementation. The aggregate type of the field must be one of

- `java.util.Collection`
- `java.util.List`
- `java.util.Set`
- `java.util.SortedSet`

In this case, other types are not supported. If a component is using a different type for this case, the component is not activated and an error must be logged. Tooling can already detect this error at build time and report it to the developer. The field must not be declared final. If it is declared as final, the component is not activated and an error must be logged. Tooling can already detect this error at build time and report it to the developer.
The collection is immutable and sorted by service ranking, highest ranking first. In the case of a clash, the services with a lower service ID are sorted before those with a higher one.

5.3.2 Event Strategy

If the event strategy is used, the component developer can decide between providing an implementation for the aggregate type or letting the DS implementation choose an implementation. If a developer is providing an implementation for the aggregate type, this needs to be done as part of the component object construction. If the field does not contain a value after constructing the object, it is set by the DS implementation right after the constructor is called.

If the DS implementation provides the implementation of the aggregate type, the provided collection is thread safe and can safely be used concurrently. The collection is immutable for the component code, however the collection is not sorted.The same restrictions apply for the type of the aggregate as with using the replace strategy. The field must not be declared final. If it is declared as final, the component is not activated and an error must be logged. Tooling can already detect this error at build time and report it to the developer.

If the component developers provides an implementation for the aggregate type, the field needs to be set during construction of the instance. The type of the field can be any of the supported types. In the case of the event strategy, the add/remove methods of the aggregate type are used to update the aggregate. A developer should not rely on `equals` or `hashCode` of the provided objects to detect which object to remove from the collection. It should rather be checked for identity of the object. The DS implementation ensures to pass the same object to the `remove` method as it passed to the `add` method. In addition, a thread safe aggregate implementation must be used or the access needs to be properly synchronized to avoid runtime errors like a concurrent modification exception.

5.4 XML Schema

For field injection a new element `field-reference` is added to the component XML schema with the attributes `policy`, `policy-option, cardinality, target` and `interface`. These attributes have the same values and meaning as those for the `reference` element. The attribute `field` contains the name of the field within the component class.

In addition the `strategy` attribute can either have the value `replace` or `event`. If it is not specified, `replace` is used as the default. For unary references specifying `event` is considered an error and the component is not activated. This case must be logged.

To handle one of the supported `Map.Entry` types, the attribute `keytype` can be used to specify the key type of the `Map.Entry`. The allowed values are `properties` and `reference`. It defaults to `properties`.

In the case of references with cardinality multiple, the runtime needs to have information about the aggregated type. The attribute `valuetype` can be used to specify the type. Allowed values are `service`, `properties`, `reference`, `serviceobjects`, or `tuple`. If not specified it defaults to `service`.

5.5 Annotation

A new annotation `@FieldReference` is added that can be used to annotate a field. The attributes `policy`, `policy-option`, `cardinality`, `target` and `interface` have the same meaning as the equivalents for the `@Reference` annotation and are mapped in the same way to the counterparts in the XML. The only exception is the reference policy, if not specified it defaults to `greedy` (and not `reluctant`).

The values of the different XML attributes for the field reference are tried to be deduced by the tooling depending on the type of the annotated field.

If the cardinality is not specified as part of the annotation, the cardinality is detected depending on the type of the field. If the type of the field is one of `java.util.Collection`, `java.util.List`, `java.util.Set`, or `java.util.SortedSet`, the cardinality is set to optional multiple (0..n), otherwise it is set to unary mandatory (1..1).
If the type of the field or the aggregated type for multiple collections is not SE (the service type) but a type that is assignable from SE, the annotation attribute interface must be set to the service type SE. By default, the type of the field, the aggregated type or the generic type (for ServiceReference, ServiceObjects) are used as the service type.
The values for `valuetype` and `keytype` in the component XML are deduced by the generic type information of the aggregate and the aggregated type.

5.6 Component Development

Field injection has some implications on the code written by the component developer:

- A field used for field injection must never be altered by client code. However, there is no way to check/ensure this from with the DS implementation. Therefore it's up to the component developer to follow this rule.
- Type safety can only be validated up to a certain point when using Java 5+. The DS implementation solely relies on the component XML to provide the correct type information, if a wrong type information is provided, a ClassCastException might occur at runtime. However the annotation tooling should try to check for wrongly used types and report this to the developer. If no generic information is available, the tooling should at least issue a warning.
- Static fields can't be used for field reference. If a component is trying to use such a field for field injection, the component is not activated. This error should be logged. In addition, tooling can already report this as an error at build time.
- Final fields can only be used for a field reference if the component developer provides the aggregate implementation in the case of a reference of cardinality multiple. If a component is trying to use such a field for field injection in other cases, the component is not activated. This error should be logged. In addition, tooling can already report this as an error at build time.
- In the case of the event strategy, the add/remove methods of the aggregate type are used to update the aggregate. A developer should not rely on equals or hashCode or the provided objects to detect which object to remove from the collection. It should rather be checked for identity of the object.

5.7 Examples

Example for unary reference:

```
@FieldReference(policy=ReferencePolicy.DYNAMIC)
private volatile MyService service;

public void doIt() {
    final MyService localService = this.service;
    if ( localService != null ) {
        // use service
    } else {
        // do something without service
    }
}
```

Example for multiple reference with replace strategy:

```
@FieldReference(policy=ReferencePolicy.DYNAMIC)
private volatile List<MyService> serviceList;

public void doItList() {
    final List<MyService> localList = this.serviceList;
    if ( !localList.isEmpty() ) {
        for(final MyService ms : localList) {
            // do something with ms
        }
    } else {
        // no service available, do something else
    }
}
```

Example for multiple reference with event strategy:

```
@FieldReference(policy=ReferencePolicy.DYNAMIC,
                strategy=ReferenceStrategy.EVENT)
private volatile List<MyService> serviceList;

public void doItList() {
    for(final MyService ms : serviceList) {
      // do something with ms
    }
}
```

5.8 Notes/Alternatives (Will be moved to section 8)

5.8.1 Byte Code Generation

The above approach could also be implemented using byte code generation. The byte code generation would generate complex methods dealing with all the cases. However this solution would depend on specific tooling.

5.8.2 Support for Collections in Methods

As described in this RFC, the DS implementation does already the heavy work of creating the collections for field injection, support for new method signatures for the bind method could be added to DS:

- protected void bindMyService(Collection<MyService> serviceCollection)

This is not part of this proposal.

6 Data Transfer Objects

A new DTO for field injection is required which is similar to the reference DTO with the difference that it points to a field and not to methods.

7 Javadoc

TODO

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

8.1 Byte Code Generation

A similar solution could also be implemented using byte code generation. The byte code generation would generate complex methods dealing with all the cases. However this solution would depend on specific tooling.

8.2 Volatile vs AtomicXXX

In order to keep the spec simple, AtomicXXX as an alternative to making a field volatile are not supported. Both concepts basically provide the same functionality, therefore limiting it to just volatile.

8.3 Support for Collections in Methods

As described in this RFC, the DS implementation does already the heavy work of creating the collections for field injection, support for new method signatures for the bind method could be added to DS:

- protected void bindMyService(Collection<MyService> serviceCollection)

This is not part of this proposal.

9 Security Considerations

No change from the Declarative Services specification as updated through RFC 190.

10 Document Support

10.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

10.2 Author's Address

Name	Carsten Ziegeler
Company	Adobe Systems Incorporated
Address	
Voice	
e-mail	cziegele@adobe.com

Name	
Company	
Address	
Voice	
e-mail	

10.3 Acronyms and Abbreviations

10.4 End of Document