RFC 212 - Field Injection for Declarative Services

Draft

11 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

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

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	Carsten Ziegeler (Adobe Systems Incorporated) initial version

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 121 in the OSGi specifications) defines a POJO programming model for OSGi services. While RFC 190 (and also RFC 208) aim at making component development with DS easier and reducing the amunt of code to write, DS 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.

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, 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.

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 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. 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.

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 synchronizing access to 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.

Handling of the cardinality information can be simplified for field injection as compared to method injection. While the cardinality information for method injection differentiates four cases (optional unary, mandatory unary, optional multiple and mandatory multiple), the cardinality part (unary or multiple) can be deduced by the type of the field (either the type is a service interface or a collection). Therefore the developer needs only to specify whether the reference is mandatory or optional.

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.

A new annotation `@FieldReference` is added which has the same attributes as the new XML element except for `field` and `cardinality`. As the annotation is annoting the field, this information is already available.

The cardinality (unary or multiple) for the annotation is detected depending on the type of the field. If a field references a service of type SERV, the type of the field must either be SERV for an unary reference, or for a multiple reference one of SERV[], java.util.Collection<SERV>, java.util.List<SERV>, java.util.Map<ServiceReference<SERV>, SERV> or java.util.SortedMap<ServiceReference<SERV>, SERV>. Other field types are not supported and a component using a different type is not activated. This error should be logged. Together with the boolean annotation attribute `mandatory` that defaults to true, the cardinality element of the XML description can be calculated correctly.

For references of cardinality multiple, always a new collection is created and set as the value of the field. The collection is immutable (of course, the values of an array could be changed). The collection is sorted by service ranking, highest ranking first. Therefore DS injects a sortable map in the case of type java.util.Map and a sortable collection in the case of a collection type.

If the field reference should reference a different service type than the type of the field, the annotation attribute `interface` must contain the service type which must be a subtype of the field's type.

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 service (unary) or to a new collection including the new service (multiple)
- Instead of calling the unbind method:
  - If the field is of cardinality unary and has the same value as the unbound service, the field is set to null
  - If the field is of cardinality unary but has a different value, it's left untouched
  - If the field is of cardinality multpile, 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.
- In addition, before the component gets activated and before any field injection is performed, the field is initialized with an empty collection for cardinality multiple.

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

- If a reference is dynamic the field 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 reference, the component is not activate. This error should be logged.
- 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. If client code is altering the field, the result is undefined.
- It is advisable to create a local copy of a field during method invocation. This ensures that the state is consistent within this method invocation. The field might be altered by DS during a method invocation. This is only required if the reference is dynamic. (The same problem exists with using method injection and updating a field from within the methods).

5.1.1 Examples

Example for unary reference:

```
@Reference(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:

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

    public void doItList() {
        final List<MyService> localList = this.serviceList;
        if ( localList.size() > 0 ) {
            // get first service
            final MyService s = localList.get(0);
            // do something with s
        } else {
            // no service available, do something else
        }
    }
```

5.2 Notes/Alternatives

5.2.1 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.

5.2.2 Support for More Collection Types

The spec is limiting the collection types to be used to a specifc set and does not allow for others. For one this keeps the spec simple. On the other hand allowing arbitrary collection types would require DS to know how to construct and fill them. If a component implementation needs a specific collection type, it can still use it by using method injection.

5.2.3 Direct Manipulation of Collections

An alternative to inject a new collection into the field for references of multiple cardinality would be to let DS modify the collection directory and only initially inject a collection instance once.

However, this would have at least these drawbacks:

- The component developer would need to take care of synchronizing access to the field. Simply declaring the field as volatile is not enough. And this synchronization would need to be the same as is used by the

DS implementation to alter the collection. An option for this would be to use concurrent collections, however this poses the problem that a collection might change during method invocation of a component . The concurrent collections are part of Java 5+.

- Synchronizing (read) access to a field can result in bottlenecks in multi threaded environments. While volatile is also a mechanism of synchronizing, it's influence on performance is usually way lower than doing synchronized blocks or using other locking mechanisms.

5.2.4 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.

5.2.5 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.

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