



OSGiTM
Alliance

RFC 212 - Field Injection for Declarative Services

Draft

12 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.

Copyright © OSGi Alliance 2014

This contribution is made to the OSGi Alliance as MEMBER LICENSED MATERIALS pursuant to the terms of the OSGi Member Agreement and specifically the license rights and warranty disclaimers as set forth in Sections 3.2 and 12.1, respectively.

All company, brand and product names contained within this document may be trademarks that are the sole property of the respective owners.

The above notice must be included on all copies of this document that are made.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the "Distribution") in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback ("Feedback") on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

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 Document Information.....	2
0.1 License.....	2
0.2 Trademarks.....	3
0.3 Feedback.....	3
0.4 Table of Contents.....	3
0.5 Terminology and Document Conventions.....	4
0.6 Revision History.....	4
1 Introduction.....	4
2 Application Domain.....	5
2.1 Terminology + Abbreviations.....	5
3 Problem Description.....	5
4 Requirements.....	6
5 Technical Solution.....	6
5.1.1 Examples.....	7
5.2 Notes/Alternatives.....	8
5.2.1 Byte Code Generation.....	8
5.2.2 Support for More Collection Types.....	8
5.2.3 Direct Manipulation of Collections.....	8
5.2.4 Volatile vs AtomicXXX.....	9
5.2.5 Support for Collections in Methods.....	9

6 Data Transfer Objects.....	9
7 Javadoc.....	9
8 Considered Alternatives.....	10
9 Security Considerations.....	10
10 Document Support.....	10
10.1 References.....	10
10.2 Author's Address.....	10
10.3 Acronyms and Abbreviations.....	11
10.4 End of Document.....	11

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 1212 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~~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 higher 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 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 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 synchronizing access to the injected field 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 including raising the minimum cardinality as outlined by RFC190. 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 annotating the field, this information is already available.

If the cardinality is not specified as part of the annotation, the cardinality (unary or multiple) for the annotation is detected depending on the type of the field. If the type of the field is an array or one of `java.util.Collection`, `java.util.List`, `java.util.Set`, `java.util.SortedSet`, `java.util.Map` or `java.util.SortedMap`, the default for cardinality is optional multiple (0..n), otherwise the default is unary mandatory (1..1).

If a field references a service of type `SERVICE` and `IN` is a type that is assignable from `SE`, the type of the field must either be `SERVIN` for an unary reference, or for a multiple reference one of `SERVIN[]`, `java.util.Collection<SERVIN>`, `java.util.List<SERVIN>`, `java.util.Set<IN>`, `java.util.SortedSet<IN>`, `java.util.Map<ServiceReference<SERVIN>, SERVIN>` or `java.util.SortedMap<ServiceReference<SERVIN>, SERVIN>`. 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. If the type of the field is not `SE` but a type that is assignable from `SE`, the annotation attribute `interface` must contain the service type `SE`.

If a field reference is defined with the cardinality multiple but the type of the field is not one of the above mentioned supported types, the component is not activated. This error should be logged. In addition, the tooling processing an annotated field should already create an error for this situation.

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. In the case of a clash, the services with a lower service ID are sorted before those with a higher one. Therefore DS injects an ordered sortable-map in the case of type `java.util.Map` and an `sortableordered` 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 a 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 multiple, a new collection without the unbound service is set. If there is no matching service, an empty collection is set as the value. This way, the value of the field can always be used throughout the component code without additional null checks.
- 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. A value set by component code as part of construction the instance will be overwritten. This way, the value of the field can always be used throughout the component code without additional null checks.

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. In addition, the tooling processing an annotated field should already create an error for this situation.
- 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 concurrently 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).
- Type safety can only be validation when using Java 5+. In this case, if a collection or map is used as a field type it should use the generic signatures. If Java 5+ is used at runtime, and the collection types are missing the generic information, a warning should be logged. If the generic information is missing or Java 1.4 is used, the DS implementation can't check whether the component developer is expecting the correct collection type. In this case, ClassCastExceptions might occur at runtime. In addition, the tooling processing an annotated field should already create an error if the generic information for a collection is missing or wrong.
- Static fields and final 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 activate. This error should be logged. In addition, tooling can already report this as an error.

5.1.1 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:

```
@FieldReference(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
        for (final MyService ms : localList) {
            // do something with ms
        }
    }
}
```



```

    } else {
        // no service available, do something else
    }
}


```

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

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 specific 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 . 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
