RFC 212 - Field Injection for Declarative Services

Draft

15 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

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 **5**

2 Application Domain **5**
2.1 Terminology + Abbreviations 6

3 Problem Description **6**

4 Requirements **7**

5 Technical Solution **7**
5.1 Supported Field Types 7
5.2 Field Injection Strategies 8
5.2.1 Static References 8
5.2.2 Unary Dynamic References 8
5.2.3 Multiple Dynamic References 9
5.2.4 Multiple Dynamic References – Update Strategy 9
5.2.5 Multiple Dynamic References – Replace Strategy 9
5.3 Aggregate Types 9
5.3.1 Replace Strategy 9
5.3.2 Update Strategy 10

5.4 XML Schema....10
5.5 Annotation....11
5.6 Component Development....11
5.7 Examples....11
5.8 Updates to DS....13

6 Data Transfer Objects....13

7 Javadoc....13

8 Considered Alternatives....14
8.1 Byte Code Generation....14
8.2 Volatile vs AtomicXXX....14
8.3 Support for Collections in Methods....14

9 Security Considerations....14

10 Document Support....15
10.1 References....15
10.2 Author's Address....15
10.3 Acronyms and Abbreviations....15
10.4 End of Document....15

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)

Revision	Date	Comments
Update	25.07.14	Update based on CPEG call: Removed support for some types, clarified field handling mechanism, renamed event strategy to update strategy Carsten Ziegeler (Adobe Systems Incorporated)
Update	30.07.14	Updated based on CPEG Virtual F2F 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 higherhiger 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` - the key of the entry is a map containing the service properties (Map<String, Object>) and the value is the service (IN).

The `java.util.Map` containing the service properties additionally implements `Comparable`. The `compareTo()` method compares map objects in the same way `ServiceReference.compareTo` does based on service ranking and service id~~, highest ranking first. If two services have the same service ranking, the one with the lowest service id is ranked higher~~. The provided `java.util.Map.Entry` implements `Comparable` in the same way.

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`
- any ~~subtype~~type assignable to ~~the above aggregate types of~~ `java.util.Collection` if the component provides the implementation and the `update` 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 situation~~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. 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

5.2.1 Static References

For fields holding a static reference, either unary or multiple, the value of the field is set once before the component activator is called and never touched again by DS. The usage of the provided value is therefore thread-safe. If a change in the referenced services occurs, the component instance is discarded and a new instance is created. For static references only this strategy, named the `replace` strategy, is allowed and therefore for static references of cardinality multiple, DS will always provide the implementation of the collection. If a different strategy is specified in the XML, the component is not activated and this error must be logged.

5.2.2 Unary Dynamic References

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:

- If the reference becomes satisfied, the field is set to a value according to the used type
- If the bound service is replaced (see 112.5.10), the field is set to the new value.
- If a reference becomes unsatisfied, the field is set to `null`.
- If the service properties of a bound service are modified, the value of the field is updated if it contains the service properties (`java.util.Map` or `java.util.Map.Entry`). In 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~~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 unary reference, the component is not activated. This error must be logged. In addition, the tooling processing an annotated field should already signal an error for this situation.

5.2.3 Multiple Dynamic References

For fields with a dynamic reference of type multiple two different strategies can be used: the `replace` and the `update` strategy.

5.2.4 Multiple Dynamic References – Update Strategy

In the case of the update strategy for references of cardinality multiple~~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~~before the component activator is called. (More about the aggregate implementation in chapter 5.3.)
- For each bound service, `Collection.add()` is called on the aggregate.
- If a service is unbound, `Collection.remove()` is called on the aggregate.~~aggreagte.~~
- If the service properties of a bound service are modified, `Collection.add()` followed by `Collection.remove()` is called if the aggregated type contains the service properties (`java.util.Map` or `java.util.Map.Entry`). In other cases, the collection is not modified.

5.2.5 Multiple Dynamic References – Replace Strategy

With the `replace` strategy, always a new ~~im~~mutable 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~~Before the component activator i s ~~constructed~~called, the field is initialized with a~~n empty~~ collection containing the currently bound references. A value set by component code as part of construction the instance will be overwritten.
- When a new service is bound, the field is set to a new collection including the new service.
- When a service is unbound, the field is set to a new collection without that service. If there is no matching service, an empty collection is set as the value.
- If the service properties of a bound service are modified, the field is updated with a new collection if the aggregated type contains the service properties (`java.util.Map` or `java.util.Map.Entry`).

~~If a~~The 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~~d~~. 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

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

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`

Other field types are not supported for the replace strategy. 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 based on object identify, 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 as described by `ServiceReference.compareTo()`.

5.3.2 Update Strategy

If the update 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 afterbefore the constructorcomponent activator is called.

If the DS implementation provides the implementation of the aggregate type, the provided collection is based on object identity, 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 type of the field must either be `java.util.Collection` or `java.util.List`. If a different type is used, the component is not activated and an error is logged. Tooling can already detect this error at build time. 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 developer 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 type assignable to `java.util.Collection`. of the supported types. As all supported types are subtypes of `java.util.Collection`, `Collection.add` and `Collection.remove` are used on the aggregate type 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. The field should be declared as final. The DS implementation is always treating the field as if declared final and might cache the field value and therefore will never be aware of any changes to the field value.

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`, `scope`, `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 `update`. If it is not specified, `replace` is used as the default. For unary references specifying `update` is considered an error and the component is not activated. This case must be logged.

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`. For unary references specifying `valuetype` is considered an error and the component is not activated. This case must be logged.

5.5 Annotation

A new annotation `@FieldReference` is added that can be used to annotate a field. The attributes `policy`, `policy-option`, `cardinality`, `scope`, `target` and `service` 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 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`, or `java.util.List` the cardinality defaults to optional multiple (0..n), otherwise it is set to unary mandatory (1..1).
If the field is marked as `volatile`, the policy defaults to dynamic, otherwise it defaults to static.
If the field is marked as `final`, the strategy is set to update.
If the type of the field or the aggregated type for collections is not SE (the service type) but a type that is assignable from SE, the annotation attribute `service` 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 value for `valuetype` in the component XML is 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 be treated with care ~~never be altered~~ by client code. ~~However, t~~There is no way for the DS implementation to check/ensure ~~this from with the DS implementation~~whether client code is altering the value of the field or the contents of a collection set to the field. Therefore it's ~~up to the component developer to follow this rule.~~suggested to not change the value of the field or the collection from client code.
- 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 update 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 update strategy, DS provided collection:

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

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

Example for multiple reference with update strategy, component provided collection

```
    @FieldReference(policy=ReferencePolicy.DYNAMIC,
                    strategy=ReferenceStrategy.UPDATE)
    private final List<MyService> serviceList =
                                new java.util.concurrent.CopyOnWriteArrayList();

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

5.8 Updates to DS

Section 5.1 introduces a comparable `java.util.Map` for service properties. Whenever a map is based to a method used for method injection, this implementation will be passed, allowing the implementor of the method to easily sort the references based on the provided map.

6 Data Transfer Objects

A DTO for field injection is required which is similar to the reference DTO with the difference that it points to a field Instead of listing the different methods. In addition it contains other information like the used strategy.

The `ReferenceDTO` introduced with RFC-190 is enhanced with a field of type String named `field` containing the field name. In order to distinguish between the different reference types, a field of type String named `strategy` is added to ReferenceDTO containing one of the following values: `LOOKUP`, `METHOD`, `FIELD_EVENT`, or `FIELD_UPDATE`. If the value is either FIELD_EVENT or FIELD_UPDATE, the field name is available via `field` and the fields for the methods all return `null`.

~~A new `AbstractReferenceDTO` is introduced holding the common information for both field and method injection, `ReferenceDTO` and the new `FieldReferenceDTO` inherit from that base class.~~

~~TODO: Better name for AbstractReferenceDTO ?~~

~~TODO: Rename ReferenceDTO to something like MethodReferenceDTO ? We might have constructor injection in a future version...~~

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