RFC 166 - Blueprint Bean Interceptors

Draft

16 Pages

Abstract

This RFC describes the requirements and solution for Blueprint Bean Interceptors as an extension of the Blueprint container specification.

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

0.5 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.

Namespace Handler – the component of the blueprint container runtime responsible for handling the blueprint transactions namespace and other **Blueprint namespaces**. The mechanism by which this is 'plugged in' to the blueprint container is not convered in this specification. This could be blueprint container implementation specific or a standard mechanism may be defined in another specification.

```
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	09/07/10	Initial version Joe Bohn, IBM, jbohn@us.ibm.com with input from: Graham Charters, IBM, charters@uk.ibm.com Andrew James Osborne, IBM, ozzy@us.ibm.com
0.1	09/08/10	Removed references to Interceptors being services in Command Line API and JMX API sections

Revision	Date	Comments
0.2	11/12/10	Updates based upon discussions from 9/8/10 EEG meeting and 9/17/10 F2F meetings: • Register Interceptors are Services per bean/bundle • Add service references (optional or mandatory) for interceptor services • use Service Ranking to order interceptors • Require that all bean managers provide a unique bean-id, even for inline or anonymous beans. • Lifecycle of Interceptor tied to lifecycle of Namespace Handler
0.3	11/24/10	Minor changes based upon discussion at Walldorf • clarify service rank order • clarify Namespace Handler requirements • clarify base blueprint requirement requirement regarding bean-id for inner/anonymous beans. • Clarify URI parameter requirement for interceptor service property.
0.31	05/09/11	Valentin Mahrwald Minor clean up for draft

1 Introduction

RFP 137 [8]. was approved defining requirements for Blueprint Bean Interceptors. This RFC describes a proposal to meet those requirements. A slightly modified version of the introduction from the RFP is included next in this introduction for convenience.

The Blueprint Container specification version 1.0 in OSGi Compendium Release 4.2 defines a dependency injection framework for OSGi bundles that supports the unique dynamic nature of beans and services. Bundles in this programming model contain a number of XML definition resources which are used by the Blueprint Container to wire the application together and start it when the bundle becomes active.

A proposed extension to the Blueprint Container Specification is the Blueprint Declarative Transaction feature [4]. One of the necessary capabilities of the Blueprint Declarative Transaction feature is a mechanism to *intercept* a

Blueprint Bean method invocation to add logic for transaction management. The same capability to intercept a blueprint bean method invocation and perform some function can also be useful for other features on Blueprint bean definitions that leverage Blueprint Namespace [3]. and require some runtime capability when processing blueprint bean methods.

This RFC refines the requirements and describes a solution for Blueprint Bean Interceptors as an extension of the Blueprint Container Specification.

Note that this RFC will be developed in conjunction with the Blueprint Declarative Transactions [4]. and Blueprint Namespaces [3]. RFCs. This RFC may be leveraged to fulfill the capabilities defined in the Blueprint Declarative Transactions design and may propose additional requirements for the Blueprint Namespaces design.

2 Application Domain

2.1 Overview

The Blueprint Container manages the creation and life cycle of blueprint beans. The Blueprint Namespace extension allows for the introduction of custom components or additional features on standard blueprint components.

Blueprint Namespace extensions can be applied to bean definitions as well as other components. When applied to bean definitions, a namespace handler can be used to enhance a blueprint bean definition or inject properties into the bean. As such, the Blueprint specification and namespace extensions are primarily concerned with the creation of blueprint beans. However, with the current capabilities they could not be involved in subsequent method invocation against the bean.

When introducing custom features for a blueprint bean there are times when it may be necessary to enhance the functionality of the bean during execution and perhaps on a per-method invocation basis. For example, to support capabilities such as those required by Blueprint Declarative Transactions it is necessary to gain access into the flow of control before and after a bean method invocation. Blueprint Declarative Transactions requires this access to facilitate transaction management.

Therefore, a solution for Blueprint Bean Interceptors requires some mechanism to extend a bean by permitting logic to be injected prior to and immediately after a bean method invocation.

It seems reasonable that such an extension would always be associated with a comparable extension in the blueprint bean component definition which is currently accomplished using the Blueprint Namespace extension. Therefore, it seems reasonable that a Blueprint Bean Interceptor would be associated with a Blueprint Namespace and should be introduced into the system by the namespace handler. It should therefore be possible to support the implementation of Blueprint Bean Interceptors using Namespace Handlers. However, this should not preclude possible future extension mechanisms such as Annotations which may also provide similar capabilities to extend a blueprint bean component definition and therefore could also introduce and support blueprint bean interceptors.

2.2 Example using Declarative Transaction Support

As mentioned earlier the Declarative Transaction Support that is currently being proposed (see [4].) has a need to intercept bean methods. One possible implementation for this capability would be to use Blueprint Bean Interceptors as defined in this RFC. In addition to the Transaction Namespace Handler required to parse the custom namespace a blueprint bean interceptor could also be created. The interceptor includes logic that is processed before a bean method invocation (pre-call), and after a method invocation (post-call). In such a scenario, a declarative transaction namespace entry is applied to the blueprint bean definition in the XML configuration file. The namespace handler is invoked when parsing the XML configuration file. At this point it is important to be able to associate the interceptor with the bean definition. This interceptor can then be retrieeived when the bean definition is being interpreted by the Bean Manager. When the blueprint bean is later instantiated, the associated interceptor(s) is bound to the bean (possibly using a proxy) such that the appropriate interceptor logic can be invoked before and/or after the bean method invocation. For Blueprint Declarative Transaction support, the logic creates any necessary transactions prior to the bean method call and commits or rolls-back those transactions following the method call as necessary for the method result.

A unique method correlator is also required to match processing before the method invocation (pre-call) with logic after the method invocation (post-call). In the case of Blueprint Declarative Transactions this correlator is used to match the transaction created in the pre-call with its response for commit or rollback in the post-call.

Given that there could be more than one interceptor associated with a particular bean, and that the processing of one interceptor could have some impact on the processing of another interceptor, some mechanism must be provided to ensure a deterministic order in invocation of interceptors for a given bean method pre-call and post-call.

It is intended that the bean interceptor capability is used in conjunction with Blueprint Namespaces as an optional feature when blueprint bean method level enhancement is also a requirement.

3 Problem Description

A Blueprint Namespace extension to a bean definition may require some logic that must be invoked in conjunction with a blueprint bean method invocation. As already discussed, this is the case with Blueprint Declarative Transactions to support the creation of a transaction prior to invoking methods on a bean and close the transaction following the bean method invocation. Declarative Transactions uses a Blueprint Namespace extension to declare the level of transaction support required for the blueprint bean methods (see [4]. for details). Without a feature such as Blueprint Bean Interceptors there is no solution to intercept the bean method invocations to begin, commit, or rollback a transaction.

This proposal is to provide a mechanism to define, register, and invoke interceptors before and after bean method invocations when appropriately specified in the Blueprint XML for the associated namespace or similar extension mechanism. The extension mechanism could then register an interceptor for the bean that would perform necessary functionality for the namespace.

4 Requirements

1. Blueprint Bean Interceptors MUST support interceptor invocation prior to a bean method call (pre-call) with the access to component metadata and all facets of the method necessary for successful pre-call processing (such as method name, parameters, etc...)

2. Blueprint Bean Interceptors MUST support a unique method invocation correlator that can be used to match a method pre-call with subsequent post-call results.

3. Blueprint Bean Interceptors MUST support interceptor invocation following a successful bean method call with access to component metadata and all facets of the method necessary for successful post-call processing (such as method name, parameters, correlator, result, etc..).

4. Blueprint Bean Interceptors MUST support interceptor invocation following an exception result to a method call with access to component metadata and all facets of the method necessary for successful post-call exception processing (such as method name, parameters, correlator, exception, etc...).

5. Blueprint Bean Interceptors MUST provide a mechanism to support deterministic ordering (relative rank) of interceptor invocation pre-call and post-call in the event that more than one interceptor is registered for a given bean method.

6. It MUST be possible to implement RFP 138 Declarative Transactions in terms of a Blueprint Bean Interceptor but it is not a requirement that Declarative Transactions leverage this capability.

7. Blueprint Bean Interceptors MUST NOT interfere with the normal flow of control for a bean which is not defined to utilize this capability.

8. It MUST be possible to implement Blueprint Bean Interceptors using RFC 155 Namespace Handlers[3].

5 Technical Solution

To implement a Bean Interceptor there are several primary tasks that must be considered. First, one must define the structure and behavior of a Blueprint Bean Interceptor itself. Second, one must define some mechanism to associate one or more Bean Interceptors with particular Blueprint Beans. Finally, one must define a mechanism for invoking the appropriate Interceptor methods in conjunction with calls to the Blueprint Bean methods.

A slightly more detailed list of the items that must be supported in a Blueprint Bean Interceptor solution follows:

1. Create a mechanism to associate Interceptors with Blueprint Bean definitions (we will use custom Namespaces).

2. Define a standard interface that Interceptors can implement (the `Interceptor` interface)

3. Define a mechanism for the Blueprint Bean Manager to locate and invoke Interceptors for Blueprint Bean method invocations.

4. Define a mechanism to order the processing of multiple Blueprint Bean Interceptor invocations.

5. Define a mechanism to deal with potential errors produced by a Blueprint Bean Interceptor implementation.

6. Define a mechanism to correlate pre-method invocations with post-method or error results.

5.1 Namespace Handler – Associating Interceptors with Beans

As mentioned above, the first primary task is to associate the Interceptors with the designated Beans. At this point in time Namespace Handlers are the primary mechanism for customization and so this design will be in terms of customization using Namespace Handlers. However, additional mechanisms of customization may be introduced in the future such as custom annotations. When such mechanisms are introduced they may also provide an appropriate mechanism to associate bean interceptors with specific bean definitions.

Namespace handlers are a natural mechanism to use for introducing Interceptors. In fact, it is difficult to conceive of a scenario where an interceptor would be required without some corresponding metadata extensions. For example, the capabilities required for Declarative Transactions require not only the introduction of a custom Namespace to associate the feature with a bean but also the runtime capability provided by Blueprint Bean interceptors to act on the method level. In the case of Declarative Transaction support the Namespace Handler has a need to allocate a transaction at the start of certain methods and commit or rollback that transaction based upon the results from the bean method. At the same time the Namespace is necessary to define the criteria applied the transaction.

5.1.1 Registering Interceptors

A Namespace Handler must first obtain a reference to the Interceptor. The Interceptor itself, it is simply a POJO that implements the `Interceptor` interface. The interface specifies the various methods that must be supported to participate as a Blueprint Bean Interceptor. The interceptor implementation can be created within the Namespace Handler itself or it can be created by the Blueprint Container and injected into the Namespace handler similar to any other Jjava object.

Next, the Interceptor must be associated with the Blueprint Bean metadata. This RFC proposes that this is accomplished by registering a service in the Service Registry for the interceptor using the bundle-Id of the bundle being parsed, the bean-id for the associated bean, and the URI for the specific Namespace as properties on the service. The Interceptor service should also be registered as implementing the the `Interceptor` interface (see seciont 5.2).

The set of service properties that should be advertised on the Interceptor when registered are as follows:

- `osgi.service.blueprint.bean-id` = the bean-id as specified in the blueprint plan or generated by the blueprint container.

- `osgi.service.blueprint.bundle-id` = the bundle-id for the blueprint bundle that is being parsed

- `osgi.service.blueprint.namespace` = the URI for the Namespace handler registering this Interceptor in the Service Registry. In the event that the Namespace handler supports multiple URIs the namespace handler should chose just one for registration with the interceptor service. The URI only serves to differentiate this interceptor service from other interceptor services that apply to the same bean instance.

Finally, there needs to be some association between the bundle being parsed that requires this interceptor and the service registered for the interceptor. This is accomplished by the Namespace handler creating a Service Reference and adding it to the component definitions for the bundle being parse. The Service Reference can be either mandatory or optional depending on the required availability of this interceptor.

Note, at this time interceptors are only being proposed for Blueprint Beans and hence it is expected that interceptors will only be registered against Blueprint Bean components. However, this should not be presumed in an implementation. Future enhancements or extensions may find it useful to also registered Interceptors against other component types and therefore any implementation specifically created for a Blueprint Bean component should gracefully ignore any registration or invocation for a different component type.

5.1.2 Using bean-id for identification in Interceptor Service Registration

The use of ~~fact~~ the bean-id along with bundle-id and namespace when registering an interceptor service has a significant implication. It implies that bean-id must be available for any bean that a namespace handler may desire to attach an interceptor, including anonymous and inner beans. Therefore, it is proposed by this RFC that the blueprint specification is changed~~to require~~ so that it no longer requires ~~that~~ `ComponentMetadata.getId()` to return~~s~~ null for inlined and anonymous bean managers and in fact requires that all bean managers return a unique bean id within the Blueprint Container.

5.1.3 Locating and Ordering Interceptors

The purpose of registering an interceptor with a bean is to enable it to be later retrieved and invoked. Therefore it is necessary to have some means to retrieve the interceptor reference. Also, because multiple interceptors can be registered for the same bean definition, it must be possible that multiple interceptors are applicable for a given bean and they must be invoked in some deterministic order. To accomplish this interceptors are ranked using the rank of the service registered for the interceptor. Interceptors are ordered by descending rank in a manner consistent with Services. In the event of a tie by rank the order of registration is used. In other words, priority is given to the highest ranked interceptors and the earliest registered interceptors.

The retrieval of interceptors is performed by Blueprint Bean Manager when instantiating a bean. The Bean Manager must leverage the service references to locate applicable interceptors for the specified bean, honoring the service reference availability of mandatory or optional and honoring the service rank to process interceptors in the appropriate order. If an interceptor service reference is mandatory then an exception should be thrown is the required interceptor service is unavailable for some reason. If the interceptor service reference is optional an unavailable service reference should be ignored by the Blueprint Bean Manager.

5.1.4 Summary of Namespace Handler requirements

- Register an interceptor service for each bean instance in the service registry using the bundle context of the namespace handler for each bean instance that requires runtime enhancement.

- Create mandatory or optional service references added to the component definitions for the bundle being parsed for each interceptor services registered for a bean by this namespace handler.

5.2 The Interceptor Interface

All interceptors must implement the common interface to provide for pre, post, and error Interceptor processing on a bean method invocation. The proposed API for Interceptor is as follows:

```
/**
 * An Interceptor interface provides support for custom interceptor implementation.
 */
public interface Interceptor {

/**
 * This is called just before the method m is invocation.
 * @param cm : the component's metada
 * @param m: the method to be invoked
 * @param parameters: method parameters
 * @return correlator which will subsequently be passed to postCall*
 * @throws Throwable
 */
public Object preCall(ComponentMetadata cm, Method m, Object... parameters) throws
Throwable;

/**
 * This method is called after the method m is invoked and returned normally.
 * @param cm: the component metadata
 * @param m: the method invoked
 * @param returnType : the return object
 * @param preCallCorrelator correlator returned by preCall
 * @throws Throwable
 */
public void postCallWithReturn(ComponentMetadata cm, Method m, Object returnType,
Object preCallCorrelator) throws Throwable;

/**
 * The method is called after the method m is invoked and causes an exception.
 * @param cm : the component metadata
 * @param m : the method invoked
 * @param ex : the <code>Throwable</code> thrown
 * @param preCallCorrelator correlator returned by preCall
 * @throws Throwable
 */
public void postCallWithException(ComponentMetadata cm, Method m, Throwable ex,
Object preCallCorrelator) throws Throwable;



}
```

5.3 Processing Interceptors

It is the responsibility of the Blueprint Bean Manager to locate and manage the interceptors as associated with Blueprint Beans. The Blueprint Bean Manager is also responsible to construct Blueprint Bean instances. The

Blueprint Bean Manager must therefore ensure that it constructs Bean instances that will honor the Interceptors that have been defined. This would involve the following steps:

1. Interrogating component definitions for service references that registered with the `Interceptor` interface for this bean.

2. Construct an instance of the Bean in such a way that future invocations of methods on the bean will invoke the `Interceptor`s that have been defined.

3. Invoking the interceptors on each method invocation. The mechanisms used to instantiate Blueprint Beans are not prescribed in the Blueprint Container RFC [5]. using specific objects and are therefore left as an internal implementation detail of a Blueprint Bean Manager. Therefore, this RFC will not call out explicit constructs or mechanisms (such as a BeanRecipe) that should be involved in the interrogation of the interceptors or construction of a bean that will honor the intention of the interceptors. This RFC will simply specify the expected behavior and assume that this is implemented by the Bean Manager as appropriate. However, one possible mechanism that can be employed would be to allocate a proxy object for the bean that would defer to the interceptors before and after each method call against the Bean.

4. If multiple interceptors are associated with a bean they must be invoked in the order specified by the rank specified in the Service for the interceptor. The processing should invoke the highest ranked interceptor `preCall` method and add the knowledge of that interceptor and returned correlator to a logical “stack” for later use with the Bean method returns. Processing should then continue with the next highest interceptor until all interceptor `preCall` methods have been invoked. The bean's method should then be invoked. When processing the result of the bean method (either the return value or exception) interceptors would be removed from the logical “stack” to process the appropriate `postCall*` method. This will effectively invoke the `postCall*` methods in reverse order from the `preCall` methods as interceptors are removed from the stack and provide symmetry on the calls.

5. Care must be taken to preserve the correlator returned from the `preCall` Interceptor method invocations so that they can be passed into the appropriate `postCall*` method invocation after the Bean method completes.

6. Any exceptions returned from an Interceptor `preCall` or `postCall*` method should be logged and interrupt program flow. (see 5.5).

7. Interceptors should not modify the method, parameters, results or exceptions provided to or returned by a Blueprint Bean method invocation. These parameters are only provided to the interceptor for reference. Introducing a solution to protect the bean data is recommended but not required. The only case where it is appropriate to make any change is if the Interceptor itself has a need to thrown an exception during processing in which case it may override a bean method exception (see 5.5).

5.4 Ordering Interceptors (Rank)

Order of Interceptors is determined by the self defined rank provided by the Namespace Handler when the Interceptor Service is registered. The rank is defined as a simple integer and is processed from highest to lowest. In the event of multiple Interceptors with the same rank the order is based upon the order of registration which is evident using the service id. The service with the lower service id was registered earlier (see section 5.2.5 of OSGi core specification R4V42).

Multiple interceptors associated with a single bean must be processed in order by rank and placed in a logical stack for processing. This is to provide symmetry for interceptor processing. Therefore, if there are 2 interceptors prioritized as A and B the processing order would be the following:

- A.`preCall`
- B.`preCall`
- bean method invocation
- B.`postCall*`
- A.`postCall*`

5.5 Error processing for Interceptors

There are cases where an interceptor may need thrown an exception. This is an indication that an error has occurred in the interceptor and processing either cannot or should not continue. One potential use of this function by an Interceptor would be to block the bean itself and additional interceptors from being invoked. For example, a validation interceptor may have a need to block subsequent access to additional interceptors and the bean method if the appropriate credential was not present. Any exception thrown from within an Interceptor in the `preCall` or `postCall*` processing should be treated as a termination condition. The exception should be logged, and re-thrown.

An exception can be thrown from an Interceptor in the `preCall`, `postCallWithReturn`, or `postCallWithException`.

1. `preCall` - In the event an exception is thrown by an Interceptor in a `preCall` the bean method should not be invoked and all Interceptors that have not yet been processed should be ignored. The `postCallWithException` methods of each interceptor that has already processed the `preCall` should be executed in the correct order before the exception is propagated back to the caller.

2. `postCallWithReturn` - In the event an exception is thrown by an Interceptor in a `postCallWithReturn` method the exception should be propagated and all remaining Interceptors should be invoked in the appropriate order with the `postCallWithException` method to indicate the error processing.

3. `postCallWithException` – In the event an exception is thrown by an Interceptor in the `postCallWithException` method the exception should be propagated and all remaining Interceptors should be invoked in the appropriate order using the `postCallWithException` method to indicate error processing.

4. If multiple exceptions are thrown from multiple Interceptors in multiple methods (`preCall` or `postCall*`) the first exception thrown should be the one propagated back to the caller.

The blueprint bean itself may also throw an exception. If this is the case, the exception should be propagated back to the caller if it is not a `RuntimeException`. In the case of a `RuntimeException` it should be returned to the caller if and only if an exception was not also thrown from an Interceptor while processing the `postCallWithException`. Basically, an exception thrown by an Interceptor is considered of greater importance than a `RuntimeException` from the bean.

5.6 Correlating pre/post/exception Interceptor calls

The `preCall` method returns a correlator Object that can be used by the `Interceptor` to match `preCall` and `postCall*` method invocations. The Object can be anything that is useful for the `Interceptor` and will simply be preserved and returned on matching methods. The Bean manager will ensure that a bean definition that

includes interceptors will invoke the `preCall` method on the associated interceptor(s) and save the correlator for subsequent processing with the same interceptor `postCall*` method.

5.7 Interceptor Lifecycle

There is a high cohesion between an interceptor and the namespace handler that rquires the interceptor capability to fulfill it's purpose. It was considered if the Interceptor should be managed independently of the namespace handler. This seems to be an approach that would introduce a lot of complexity for very little benefit. Any changes in a namespace handler that leverages the interceptor support would most likely result in changes to the corresponding interceptor to react to schema or other changes in the namespace. Furthermore, it is unlikely that there would be changes in an underlying interceptor without corresponding changes in the namespace handler. Therefore, for the purposes of this design it is assumed that the lifecycle of an interceptor is consistent with the lifecycle of a namespace handler and most likely both will be delivered in the same bundle.

Because the namespace handler is the primary element that registers and manages any associated interceptors it seems appropriate that lifecycle scenarios are primarily (and perhaps exclusively) driven from the namespace handler. The namespace specification should address the issue of managing blueprint containers that have been extended by the namespace handler such that the blueprint bundle can be updated (perhaps re-parsed) if the namespace handler is updated. The creation, removal, or updating of associated interceptors via advertised interceptor services in the service registry and corresponding service references in the blueprint components would be managed by the namespace handler implementation following guidelines from the blueprint namespace handler specification.

6 Considered Alternatives

6.1 Bytecode Weaving required implementation

Bytecode Weaving is another alternative that could be considered to implement Bean Interceptor capability. For details on Bytecode Weaving reference [6]. and [7].

There is nothing in this design that prohibits a Bytecode Weaving solution. A Bytecode weaving solution could be triggered by a Namespace Handler which could weave the interceptor logic into the specified bean. Mechanisms similar to those defined in this design would still be necessary to prioritize and coordinate the activities of multiple interceptors and maintain knowledge of them in the Component Definition Registry. At this point in time it appears that requiring a Bytecode weaving solution would not result in a simplified or vastly superior solution and would appear to bring more complexity into the design for little benefit. Therefore, a Bytecode Weaving approach was not pursued as the primary solution for this design.

7 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

8 Document Support

8.1 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

[3]. RFC 155 – Blueprint Namespaces is available in the OSGi Alliance subversion repository in rfcs/rfc0155/rfc-155-blueprint-namespaces.pdf.

[4]. RFC 164 – Blueprint Declarative Transaction is available in the OSGi Alliance subversion repository in rfcs/rfc0164/rfc-0164-blueprint_transaction.pdf.

[5]. RFC 124 – Blueprint Service is available in the OSGi Alliance subversion repository in rfcs/rfc0124/rfc-124.pdf.

[6]. RFP 139 – Bytecode Weaving Requirements is available in the OSGi Alliance subversion repository in rfps/rfp-0139-bytecode-weaving.pdf

[7]. RFC 159 – Bytecode Weaving Design is available in the OSGi Alliance subversion repository in rfcs/rfc0159/rfc-159-BytecodeWeaving.pdf.

[8]. RFP 137 – Blueprint Bean Interceptors Requirement is available in the OSGi Alliance subversion repository in rfps/rfp-0137-Blueprint_Bean_Interceptors.pdf

8.2 Author's Address

Name	Joe Bohn
Company	IBM
Address	P O Box 12195, 196A/503/N231, 3039 Cornwallis Rd., Research Triangle Park NC 27709-2195
Voice	919-543-1096
e-mail	jbohn@us.ibm.com

8.3 Acronyms and Abbreviations

8.4 End of Document