Blueprint Container 1.1

Draft

15 Pages

Abstract

Update Blueprint Container to 1.1 with some features requested via bug reports or directly from users.

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 8.1.

```
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	*21/05/12*	*Initial Draft Requirements*
0.2	*2nd November 2012*	*First stab at design.*
0.3	*27th February*	*Updates from Nov 2f2 & tidy-up for draft publication.*
0.4	*20th August*	*Updates to Requirement design.*

1 Introduction

This RFC will propose some new minor feature enhancements to the existing 1.0 Blueprint Container specification. The feature requests have come from bug reports or through user experience.

2 Application Domain

The following was taken from the Blueprint 1.0 requirements and design documents (RFP 76 & RFC 124. respectively)

The primary domain addressed by this RFP is enterprise Java applications, though a solution to the requirements raised by the RFP should also prove useful in other domains. Examples of such applications include internet web applications providing contact points between the general public and a business or organization (for example, online stores, flight tracking, internet banking etc.), corporate intranet applications (customer-relationship management, inventory etc.), standalone applications (not web-based) such as processing stock feeds and financial data, and “frontoffice” applications (desktop trading etc.). The main focus is on server-side applications.

The enterprise Java marketplace revolves around the Java Platform, Enterprise Edition (formerly known as J2EE) APIs. This includes APIs such as JMS, JPA, EJB, JTA, Java Servlets, JSF, JAX-WS and others. The central component model of JEE is Enterprise JavaBeans (EJBs). In the last few years open source frameworks have become important players in enterprise Java. The Spring Framework is the most widely used component model, and Hibernate the most widely used persistence solution. The combination of Spring and Hibernate is in common use as the basic foundation for building enterprise applications. Other recent developments of note in this space include the EJB 3.0 specification , and the Service Component Architecture project (SCA).

Some core features of the enterprise programming models the market is moving to include:

- A focus on writing business logic in “regular” Java classes that are not required to implement certain APIs or contracts in order to integrate with a container
- Dependency injection: the ability for a component to be “given” its configuration values and references to any collaborators it needs without having to look them up. This keeps the component testable in isolation and reduces environment dependencies. Dependency injection is a special case of Inversion of Control.
- Declarative specification of enterprise services. Transaction and security requirements for example are specified in metadata (typically XML or annotations) keeping the business logic free of such concerns. This also facilitates independent testing of components and reduces environment dependencies.

- Aspects, or aspect-like functionality. The ability to specify in a single place behavior that augments the execution of one *or more* component operations.

In Spring, components are known as “beans” and the Spring container is responsible for instantiating, configuring, assembling, and decorating bean instances. The Spring container that manages beans is known as an “application context”. Spring supports all of the core features described above.

3 Problem Description

3.1 Non-Damped Reference Managers

Blueprint reference managers have mandatory damping, however, not all use cases desire damping.. The timeout period can be changed, but this requires additional configuration and does not allow damping to be turned off for an individual reference. There is a need for non-damped reference managers. One benefit of this would be to avoid a timeout period which would allow Blueprint to immediately flag the component as having missing dependencies should the service not be available.

3.2 Blueprint Grace Timeout

When the Blueprint grace period is reached, the Blueprint container no longer checks for dependencies, even if outstanding dependencies are then satisfied. At this point, the only way the container can be refreshed is to restart the bundle. There needs to be a better way to control the life-cycle of blueprint containers that does not risk them becoming zombies.

3.3 Bug 2233 –'Satisfied' Lifecycle Notification

The lifecycle events for Blueprint aren't sufficient to get a complete understanding of what Blueprint is currently doing. If blueprint enters the grace period waiting for dependencies, there is no event fired to say when it's dependencies have been satisfied. This is an important piece of information to know when trying to track down blueprint problems.

3.4 Injection of Service Properties

If a User wants to access the service properties of an injected service, they must first be injected with the service reference and then programmatically use the service reference to access the properties. Not only is it an inconvenient extra step to have to do to get to the properties, it also ties the bean to the OSGi APIs, which can impact the ability to unit test the bundles.

3.5 Bug 2192 – Factory Services lifecycle issues

If a factory Service is removed and replaced by another factory service, Blueprint doesn't recreate the beans that were created with the original factory.

This is a Blueprint example for the problem:

```
<reference id="foo" interface="javax.persistence.EntityManagerFactory/>

<bean id="not_working_after_update" class="SomeClass">
    <property name="bar">
        <bean factory-component="foo" factory-method="createEntityManager"/>
    </property>
</bean>
```

In the example, the bean created by the service Reference foo, should be deleted and re-created, if the EntityManagerFactory service that foo is bound to, is removed and replaced by another EntityManagerFactory service.

3.6 Bug 1295 - Allow Namespace handlers to use inline Blueprint elements

Custom namespace handlers should be able to include / in-line elements from the Blueprint Schema. Bug 1295 was raised against RFC 155 (Namespace Handlers), but requires changes to the core Blueprint specification that are potentially valuable even in the absence of a namespace handler standard.

```
<tx:transacted>
  <bean ... />
</tx:transacted>
```

3.7 Bug 2406 – Add blueprint extender capability definition

The latest Core OSGi specification introduced the standard capability namespace for extender implementations, and the Blueprint extender needs to implement this new namespace e.g.

```
Provide-Capability: osgi.extender; osgi.extender="osgi.blueprint";
 uses:="org.osgi.service.blueprint.container,
 org.osgi.service.blueprint.reflect";version:Version="1.1"
```

3.8 Bug 2484 – Address lack of Blueprint opt-in header

Best practice design for extenders is for them to require an opt-in header specified in the extendee bundle's manifest. Blueprint 1.0, however, treats the bundle-blueprint header as an opt-out (when no value is specified). This means the default behavior for blueprint 1.0 is to search all bundles for blueprint configuration, unless they contain the opt-out header, resulting in significant performance issues for large deployments. Blueprint needs a way to address this, ideally without breaking backwards compatibility.

3.9 Allow service properties to be added via <property> elements in Service Manager

When you want to define service properties within a Service Manager, you have to define a service-properties element, and then define entry elements for each property, which seems unnecessarily verbose and over-complicated.

3.10 Bug 2547 – Make it easier to consume optional services

The blueprint container allows reference-managers to be “optional”, meaning that they may have no backing service. As identified in Error: Reference source not found, when a reference manager has no backing implementation it can wait for a long time, then it throws ServiceUnavailableException. Whilst reducing this timeout to zero is helpful in error cases for mandatory services, for optional reference managers “no backing service” is a valid, main-path state. Throwing an exception for every one of these invocations is both wasteful and counter-intuitive – it is not an “exceptional” case.

Currently to avoid this scenario blueprint components must be declared as reference-listener objects. They then receive notifications of the reference bind/unbind events. This adds significant additional threading complexity to the blueprint component, and (assuming it is not safe to hold a lock/monitor while calling the reference) there is still a race condition that can result in a ServiceUnavailableException.

Blueprint's purpose is to make exposing and consuming services simpler. The situation outlined above is not simple, and needs to be improved.

4 Requirements

Blueprint1 – A component MUST be able to request that a Reference Manager is not damped.

Blueprint2 – It MUST be possible for a Blueprint Container to use the grace period without risking permanent failure (requiring a bundle restart) if the period expires. Once the grace period has been reached, the Blueprint container MAY decide to partially start the beans and services that have had their dependencies satisfied, or it MAY decide to continue to wait for outstanding dependencies. Relevant Lifecycle events MUST be issued for each scenario.

Blueprint3 – The Blueprint Container MUST issue an additional lifecycle event when the dependencies of a component are satisfied.

Blueprint4 – A component MUST be able to have the service's properties injected directly without requiring the bean to use OSGi framework APIs.

Blueprint5 – The Blueprint Container MUST ensure that beans created by factory services are kept in sync with the life-cycle of the factories. For example, if the factory service is replaced, the beans from the old factory should be removed and new beans created using the new factory service.

Blueprint6 – Custom namespaces MUST be able to include or have inlined Blueprint elements from the Blueprint Schema. This MAY mean changing the Blueprint Schema to define the Blueprint element types upfront,

and refer to them throughout the Schema structure, rather than, as it does today, in-lining the element type definitions within the Schema structure.

Blueprint7 – The Blueprint Container MUST support the extender capability definition.

Blueprint8 – Blueprint Service Managers MUST support <property> elements.

Blueprint10 – It MUST be possible to configure Blueprint processing such that the extender does not process any bundles that do not contain the Bunde-Blueprint header.

Blueprint11 – It MUST be possible to use Optional Services without waiting for unsatisfied services.

5 Technical Solution

5.1 Non-Damped Reference Managers

To allow Reference Managers to be configured to not damp the services that they reference, the existing optional timeout property with be able to be set to -1, This new value indicates that the reference should not be damped, and should throw a ServiceUnavailableException should the reference not exist at the time the element is processed by the Blueprint container. This attribute is only available on the reference element, and not on the reference-list element, as reference-lists are not damped.

Reference Manager damping can also be specified at the blueprint element level using the new optional `timeout` attribute. The attribute applies to all Reference Managers for the corresponding Blueprint container. This attribute can have the same values as the timeout attribute on the reference Manager, so setting this to -1 would mean all of the Blueprint container's Reference Managers would not damp their services.

The timeout attribute on individual reference elements will take precedence over the blueprint element's attribute.

Adding these attributes will require a new version of the Blueprint XML schema.

5.2 Blueprint Grace Period enhancements

When the grace period is reached, Blueprint will have two new options available in order to avoid creating 'zombie' Blueprints. These are specified on the existing directive called, blueprint.graceperiod which is specified on the bundle symbolic name. The new values are `allowPartial` , `forever`. For clarity, we will also introduce new values `fail` and `none`, which will have the same meaning as true and false respectively.

When a value of `allowPartial` is specified, Blueprint must wait for the graceperiod and when the graceperiod is reached, rather than issuing a FAILURE event and performing “Destroy”, it must create the blueprint container and set up as much of the blueprint as possible, based on the set of satisfied mandatory references in the same way it would if the graceperiod were set to `false`.

When a value of `forever` is specified, Blueprint must wait for the graceperiod and when the graceperiod is reached, rather than issuing a FAILURE event and performing “Destroy”, the Blueprint runtime must issue a new GRACE_PERIOD event and begin a new grace period.

TBD: Consider changing the default behaviour to allowPartial, as this is probably the most useful graceperiod.

We will need a new version of the Blueprint XML schema for this new attribute.

5.3 New Satisfied Life-cycleEvent

When the Blueprint Container processes service references and issues a GRACE_PERIOD event because there are missing mandatory dependencies, it will now issue a SATISFIED event when all mandatory dependencies are finally satisified and before proceeding to the “Register Services” process.

The Blueprint Event property DEPENDENCIES will have the same array of Strings containing the dependencies that was issued in the corresponding GRACE_PERIOD event.

5.4 Injection of Service Properties

When looking for an appropriate bean set method to call for injection of a service object, as a last option, Blueprint will now also look for a method with the following signature:

set{PropertyName}(T ref, java.util.Map<String, ?> props)

PropertyName is the name of the bean property, specified in the property element, for example

```
public class C {
  public void setProxy(T ref, Map<String, ?> props) { ... }
}
<reference id="p" interface="T"/>
<bean id="c" class="C">
  <property name="proxy" ref="p"/>
</bean>
```

It is important to note that the injected service properties are an unmodifiable map of properties, and is a snapshot at the time the map is injected.

5.5 Factory Services

Currently when factory services are used, the actual bean created by the factory service, is injected. If the factory service is replaced by another factory service, the blueprint container doesn't replace the existing beans with ones created using this new factory.

The Blueprint container will ensure that whenever a factory service is replaced, all beans created by the original factory service are removed and replaced by new beans created by the new service. The factory service will also now return a damped proxy, rather than the actual bean.

5.6 Inlining of Blueprint elements

In order to support the ability for Custom Namespace Handlers to use inlined Blueprint elements,the Blueprint schema needs to be amended to declare the element types outside of the nested groups and reference them within the groups e.g.

```
<xsd:group name="allComponents">
    <xsd:choice>
        <xsd:element ref="service"/>
        <xsd:element ref="ref-list"/>
        <xsd:element ref="ref-set"/>
        <xsd:group ref="targetComponent"/>
    </xsd:choice>
</xsd:group>
<xsd:element name="service" type="Tservice"/>
<xsd:element name="ref-list" type="Tref-collection"/>
<xsd:element name="ref-set" type="Tref-collection"/>
```

5.7 Extender Capability

The Core OSGi specification defines a capability namespace for extender implementations. To enable Blueprint extendees to express a requirement for a Blueprint extender and also ensure classpath consistency with the chosen extender, the Blueprint extender must now specify the following capability:

```
Provide-Capability: osgi.extender; osgi.extender="osgi.blueprint";
 uses:="org.osgi.service.blueprint.container,
 org.osgi.service.blueprint.reflect";version:Version="1.0"
```

The Blueprint container must not extend a bundle that is wired to another provider of the Blueprint extender capability.

A Require-Capability header that wires to this extender capability opts the bundle in to being processed by the blueprint extender. An example of the Require-Capability headers is as follows:

```
Require-Capability: osgi.extender; filter:="(osgi.extender=osgi.blueprint)"; ;
 path:List<String>="lib/account.xml, security.bp, cnf/*.xml"
```

The path attribute follows the same pattern as the Bundle-Blueprint header described in section 121.3.4. Unlike, Bundle-Blueprint, absence of a path attribute means the blueprint extender searches for the blueprint xmls in the default location (i.e. OSGI-INF/blueprint/*.xml.

5.8 Allow <property> elements in Service Managers

The current mechanism for defining service properties in Service Managers using the <service-properties> elements and <entry> sub-elements is quite cumbersome, and is designed with the idea that the properties will be put into a Map.

Service Managers should allow one or more <property> elements to be defined, in the same way that Bean Managers do. All defined properties will be used as the service properties when the service is registered.

If a service manager is configured both with the existing <service-properties> mechanism, and also with the <property> elements, these will be merged into a single Map.

If there are any duplicate keys, <property> elements take precedence over <service-properties>.

5.9 Blueprint Extender Configuration (Bug 2484)

Best practice design for extenders is for them to require an opt-in header specified in the extendee bundle's manifest. Blueprint 1.0, however, treats the Bundle-Blueprint header as an opt-out (when no value is specified). This means the default behavior for Blueprint 1.0 is to search all bundles for blueprint configuration, unless they contain the opt-out header, resulting in significant performance issues for large deployments. Rather than change the default blueprint extender behavior this specification adds the ability to configure the extender behavior.

Blueprint 1.0 also has a number of other configurations provided on the Bundle-Blueprint header or in the Blueprint XMLs. These are arguably better suited to being configured on the extender, rather than per-Blueprint. For example, blueprint.timeout, (grace perioid timeout), default-timeout, default-activation, default-availability.

The Blueprint extender is configured through configuration admin. The pid for the Blueprint extender configuration dictionary is `osgi.blueprint.Extender`.

5.9.1 Extendee Header Behavior

The extendee header behavior can be configured by setting the `osgi.blueprint.header` in the `osgi.blueprint.Extender` configuration dictionary. The default value is `OptOut` and configures the extender to behave as defined by the Blueprint 1.0 specification.

The configuration value of `OptIn` switches the default and opt-out behaviors of the extender, specifically:

- Absence of the `Bundle-Blueprint` header means a bundle is not processed for Blueprint configurations.
- If the `Bundle-Blueprint` header is specified with no value then the extender must search for Bluperint configurations in the default location.

The remaining extender processing of the `Bundle-Blueprint` header is unchanged.

5.9.2 TODO: Other things configured through the dictionary

5.10 Add “Default” implementations for blueprint reference managers

Blueprint reference managers are responsible for locating and tracking suitable services from the OSGi service registry. They are also required to produce a proxy object which wraps the tracked service, as and when the tracked service becomes unavailable a new service replaces it inside the proxy. No reinjection of the proxy object is required.

If the tracked service becomes unavailable and there is no suitable replacement then invocations of the proxy object are required to wait until a replacement is found, or throw ServiceUnavailableException if this wait period times out.

This functionality could be usefully extended with the concept of “default service implementations”. Default service implementations can be thought of as locally visible services with the minimum possible ranking. This means that they aren't visible to other components, will never take precedence over a real service, can be used whenever no suitable service is available.

5.10.1 Declaring a default implementation

A default service implementation can be declared for a reference manager either by using an attribute to refer to an existing blueprint component, or by declaring an inline bean inside a <default> element.

Example 1:

```
<reference interface="java.util.List">
  <default>
    <bean class="java.util.ArrayList"/>
  <default>

</reference
```

Example 2:

```
<bean id="example" class="java.util.ArrayList"/>

<reference default="example" interface="java.util.List"/>
```

When a default service implementation is declared in this way it is important that the default object be of the correct type to match the interfaces listed in the reference manager. If this is not the case then the blueprint container implementation is required to throw a `ComponentDefinitionException` when creating the reference manager.

5.10.2 Default implementations and the Null Proxy pattern

For some services it may be difficult to provide a suitable default implementation. Equally blueprint managed bundles may not wish to make themselves providers of the service that they are using (for example the HttpService), because it will significantly restrict the range of implementations with which they are compatible.

In this case the Null Proxy pattern can be used to support clients that want the benefits of a default implementation, but without the effort of implementing a default. The null proxy pattern involves generating a proxy object that performs no action, and returns `null` or null-like values, for all method calls. This would mean that all void methods do nothing, all methods that return numeric primitives return zero, all boolean methods return false, and all methods that return references return `null`.

The null proxy pattern can be enabled in blueprint using a new Environment Manager “blueprintNullProxy”, which has prototype scope. Whenever this environment manager is used the blueprint container must create a new null proxy matching the expected type of the receiver. For example in the following:

```
<reference default="blueprintNullProxy" interface="java.util.List"/>
```

The type of the blueprintNullProxy created would be `java.util.List`.

6 Considered Alternatives

7 Security Considerations

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

8.2 Author's Address

Name	Tim Mitchell
Company	IBM
Address	
Voice	
e-mail	tim.mitchell@uk.ibm.com

Name	Graham Charters
Company	IBM
Address	
Voice	
e-mail	charters@uk.ibm.com

Name	Tim Ward
Company	Paremus
Address	
Voice	
e-mail	tim.ward@paremus.com

8.3 Acronyms and Abbreviations

8.4 End of Document