RFC 155 - Blueprint Namespaces

Draft

27 Pages

Abstract

This RFC extends the Blueprint service to support user defined namespaces in Blueprint configuration files.

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.

`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
0	12/18/09	First draft. Glyn Normington, VMware, gnormington@vmware.com
0.1	12/24/09	Convert Peter Kriens' key requirements into problem descriptions with initial questions for discussion. Glyn Normington
0.2	01/18/10	Changes from review at Southampton F2F. Glyn Normington
0.3	05/11/11	Update / extend draft from Apache Aries implementation Valentin Mahrwald, IBM, mahrwald@uk.ibm.com

Revision	Date	Comments
0.3.1	05/19/11	Add a couple of sections around life-cycle, versioning and requirements.
0.3.2	06/24/11	Blueprint builder API draft
0.3.3	07/20/11	Minor edits with regards to custom scopes et al

1 Introduction

The Blueprint service defines some standard namespaces, but users of the Blueprint service would like to define their own namespaces in order to create custom components and to add features to components defined using the standard namespaces.

The work to standardize custom namespace support began in RFC 124 - see reference [3]. However, the requirements and technical solution did not stabilize in time for R4.2.

This RFC clarifies the requirements and proposes a solution.

Note that this RFC will be developed in parallel with a prototype implementation based on the Blueprint service reference implementation.

The Blueprint service support for Configuration Admin is the subject of a separate RFC 156 – see reference [4]. The Blueprint service support for Declarative Transactions is the subject of another separate RFC 164 – see reference [7].

2 Application Domain

The Blueprint service enables an OSGi bundle to define Java objects known as *components* using a XML configuration file stored in the bundle. Components may be published as services in the service registry and may consume services from the service registry.

The Blueprint service is described in the R4.2 Service Compendium - see reference [5].

The XML standard supports namespaces – see reference [8]. Each namespace is defined by a XML schema and identified by a XML schema URI.

3 Problem Description

Add any further concrete use cases that are actually required. Avoid the temptation to make up use cases.

3.1 Configuration Admin Namespace Support

The use case which prompted this RFC to be written is to enable the configuration admin namespace described in RFC 124 to be added as a user defined namespace provided by the blueprint service. This use case is the subject of RFC 156.

3.2 Namespace Handler Provided Classes

Namespaces must be able to provide classes directly, not only through their class name. This will minimize exports and consistency problems. For example, it must be possible for a namespace handler to add a bean whose concrete class is provided by the namespace handler bundle.

In order to support lazy loading and throw away class loader techniques, it should be possible to perform namespace handling before the bundle referencing the namespace has been resolved.

If this requirement can be satisfied by the bundle providing a namespace also exporting packages, then the bundle referencing the namespace need not be resolved before namespace handling is performed. In this case, does the namespace handler need to be able to add the corresponding package imports to the manifest of the bundle referencing the namespace? Supporting manifest transformations doesn't really fit well with the scope of this RFC.

3.3 Component Registry Processing

A namespace handler must be able to manipulate the component registry after the blueprint XML has been parsed but before any components have been instantiated.

Need to define the lifecycle states of the component registry.

This is analogous to Spring's Bean Factory Post Processor and seems to be a design consideration.

Need to determine the real requirement (“Why is this needed?”) and supporting use case(s). One use case is an element that adds default properties or interceptors to every component in a blueprint module. For example a default transaction interceptor that adds a default transactionality to all components. [This is not the current implementation of transactions but could be].

3.4 Namespace Versioning

Peter Kriens asked for “proper versioning”. It seems sufficient for namespace handlers to be versioned and for references to a namespace to be able to select the version (or version range) to be used.

3.5 Enclosed Elements

It must be possible to enclose elements, either from the blueprint namespace or other namespaces, in an element of a user defined namespace. For example, it should be possible to decorate an existing element with a “transactional” attribute from a transaction namespace.

The approach of Spring DM to be able to parse elements from the referenced namespace and decorate elements from any other namespace using attributes from the referenced namespace should satisfy this requirement.

The component registry certainly needs to be accessible to namespace handlers. This for example is a used by the Apache Aries JPA namespace handler, which injects a property into an enclosing component and simultaneously generates (or uses) matching service-reference elements.

3.6 Component Metadata Builder

Whether the full power and complexity of a Spring Bean Factory Post Processor is necessary remains to be seen.

4 Requirements

REQ-COMPATIBILTY: Bundles using the Blueprint service that do not reference non-standard namespaces MUST continue to work correctly without modification.

REQ-SCHEMA: A user defined namespace MUST be defined using a XML schema file and identified by a XML schema URI. A bundle which defines one or more user defined namespaces MUST contain the corresponding XML schema files in a standard directory of the bundle.

Q: What about references are they all to be included by value in the standard directory?

REQ-XMLNS: A user defined namespace MUST be referenced from a blueprint configuration file by using a xmlns attribute.

REQ-HANDLER: Each user defined namespace MUST be associated with a namespace handler. Each namespace handler MUST be published as a service in the OSGi service registry by the bundle that defines the namespace . It is an error to refer to a namespace for which a handler cannot be obtained within a specifiable timeout period. It is reasonable to share the current timeout setting, but only the time remaining after the namespace handler wait should be allowed for module context dependency satisfaction.

REQ-COMPONENT: When an element or an attribute of a user defined namespace is used in a blueprint configuration, the corresponding namespace handler MUST be able to decorate components defined by the blueprint configuration and add components to the set defined by the blueprint configuration.

REQ-VERSIONING: It MUST be possible to version user defined namespaces.

REQ-CONFIG-ADMIN: It MUST be possible to implement RFC 156 in terms of a user defined namespace.

REQ-TRANSACTION: It MUST be possible to implement RFC 164 in terms of a user defined namespace.

REQ-INTERCEPTOR: It MUST satisfy additional requirements specified in RFC 166 [8] (Blueprint interceptors).

Tentative

***REQ-CUSTOM-SCOPES**: The specification SHOULD define how custom scopes relate to namespace handlers.*

***REQ-LOCAL**: It SHOULD be possible to use and validate against schemas, even ones that include or import further schemas without a network connection (assuming all are available locally). This is done by default if everything is included in the standard directory as described above.*

***REQ-LIFECYCLE**: The specification MUST define how Blueprint responds to the namespace handler lifecycle events. It SHOULD allow where appropriate the lifecycle of the Blueprint container to be bound or independent of that of a namespace handler it uses.*

***REQ-PARSABLE**: The specification COULD define if and how Blueprint descriptors including namespaces can be be parsed offline. (For example for generating OBR metadata …)*

***REQ-REF-LIFECYCLE**: Config admin may introduce new components that need to be satisfied before. So some mechanism to add 'reference' type components that the Blueprint container waits for.*

What about context should namespace handlers keep their own context or should we allow them to maintain context for the scope of a parse. Reason for asking is the implementation of global scopes like in transactions or the global attributes in the cm spec.

Also at this stage there should be a section on considered namespace handler use cases such as:

- *config admin (provide different property and value injection and augment components, including new type of service to be waited for)*
- *transactions (register interceptor)*
- *macro expansion*
- *globally enable tracing for Blueprint components*
- *JPA service injection*
- *What about the annotation parsing and transforming use case (can be done but must include initial namespace element that introduces a registry processor).*

4.1 Non Requirements

The following requirement is explicitly excluded. It adds significant complexity without a great deal of benefit to the user.

REQ-SELF-REF: A bundle MUST be able to reference a user defined namespace which the bundle defines.

REQ-PROP-PLACEHOLDER: As discussed in RFC 156 bugs, adding support for property placeholders akin to Spring property placeholders would pose significant technical challenges and has been removed from the solution of RFC 156.

5 Technical Solution

A user defined namespace is defined using a XML schema file. The schema has a targetNamespace attribute whose value is a XML schema URI which uniquely identifies the namespace.

A user defined namespace is referenced from a blueprint configuration file by setting a xmlns attribute of the top level element to the XML schema URI of the user defined namespace. Elements from a user defined namespace may then be used to define components or to decorate component definitions in the configuration file.

The blueprint service provides a mapping of XML schema URI to namespace handler. It is an error to attempt to associate more than one namespace handler with a given XML schema URI. XML schema URIs may include a version number and will be associated with the corresponding namespace handler version.

When the Blueprint extender starts to process a blueprint configuration file, it examines the xmlns attributes of the top level element and uses the corresponding handlers to continue processing the configuration file. It is an error for a configuration file to refer to a user defined namespace for which there is no corresponding handler.

The blueprint service also provides a mapping of XML schema location to XML schema file which may be used to avoid retrieving the schema file from a remote location.

API to be added once the full set of requirements is known.

5.1 Namespace handler sample

An example usage of a namespace is given in the Blueprint snippet below, which extends Blueprint with a hypothetical caching functionality. The example highlights the different extension points a namespace handler may provide. Line (1) defines a cache in the Blueprint, i.e. contributes a top-level component to the Blueprint. Similarly line (4) defines a local cache as an anonymous inner component. Both these definitions can be regarded as purely generational in that they generate new Blueprint definitions without affecting surrounding definitions. Lines (2) and (3) on the other hand serve to modify the enclosing component (one via a custom property and one via a child element). For these cases the aim of the custom elements is to enhance or potentially replace the existing component with a component that supports caching.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:cache="http://www.example.org/xmlns/cache">

  <cache:lru-cache id="myCache"/>                                  (1)

  <bean id="fooService" class="FooServiceImpl"
        cache:cache-return-values="true">                          (2)
    <cache:exclude>                                                (3)
      <cache:operation name="getVolatile"/>
    </cache:exclude>
    <property name="myProp" value="12"/>
  </bean>

  <bean id="barService" class="BarServiceImpl">
    <property name="localCache">
      <cache:lru-cache/>                                           (4)
    </property>
```

```
  </bean>

</blueprint>
```

Corresponding to this use of namespace, the namespace schema must be available from the bundle that provides the namespace handler.

5.1.1 Namespace Handlers

Instances of namespace handlers are registered in the OSGi service registry with the `osgi.service.blueprint.namespace` property to denote the namespace(s) being handled. The NamespaceHandler interface is defined as follows:

```
interface NamespaceHandler {

   /**

    * The URL where the xsd file for the schema may be found. Typically used to return a URL to a

    * bundle resource entry so as to avoid needing to lookup schemas remotely.

    * If null is returned then the schema location will be determined from the xsi:schemaLocation
attribute

    * value.

    */

   URL getSchemaLocation(String namespace);



  /**

   * Called when a top-level (i.e. non-nested) element from the namespace is encountered.

   * Implementers may register component definitions themselves, and/or return a component definition

   * to be registered.

   */

   ComponentMetadata parse(org.w3c.dom.Element element, ParserContext context);



  /**

   * Called when an attribute or directly nested element is encountered. Implementors should parse the

   * supplied Node and decorated the provided component, returning the decorated component.

   */

   ComponentMetadata decorate(org.w3c.dom.Node node, ComponentMetadata component,

                              ParserContext context);
```

Comments:

It is not clear how the method getSchemaLocation could be used to handle xsd:import and xsd:include properly. The namespace handler should respect xml catalog-resolvers from the service registry. This could be the generic mechanism for providing the namespace schema. However, it would complicate the base case of a self-contained schema.

The API also makes no distinction between nested custom tags and custom attributes.

5.1.2 Namespace Handler Lifecycle

During startup of a Blueprint container namespace handler must be satisfied before any other processing is done. This should involve an appropriate GRACE_PERIOD as well as EventAdmin notifications. Processing of a Blueprint only proceeds when all namespace handlers have been obtained.

The previous spec talks about integrating the waiting for namespace handlers into the general tracking of reference elements. However that leaves it a bit unclear what happens when processing of a namespace handler generates additional service references. Logically also the processing of namespace handlers is a precursor to parsing itself.

Also need to define where ComponentDefinitionRegistryProcessors fit into the Blueprint lifecycle.

Another interesting question arises when a namespace handler disappears while a Blueprint bundle that uses the namespace handler is still active. This is in particular interesting in the context of interceptors, which be RFC 166 are registered against the namespace handler's bundle and share the namespace handler's lifecycle.

Two different scenarios spring to mind, where (b) should be the rule and (a) the exception:

(a) The interceptor is stateful and state is maintained by the namespace handler.

(b) *The interceptor is stateless, for example the transaction namespace and interceptor are stateless after the initial parsing of information. In this case there is no reason that either interceptors or generated component definitions should be invalidated.*

Restarting a Blueprint when a replacement namespace handler becomes available sounds like a very hard thing to get right. So maybe just destroy the Blueprint if namespace handler of type (a) are present (or at least destroy the components that are touched) and for namespace handlers of type (b) no changes are necessary.

De-registration of namespace handlers can be met by various different responses. The desired response can be configured both by namespace handlers and by the client bundle. If the response differs, the more restrictive response will be used.

- Do nothing (update-strategy=none). This works for many namespace handlers, which set up new components but need not be present at runtime for operations on them.
- Update if a replacement is available (update-strategy=restart-if-possible). This works like the above if no replacement is available, but also updates (by restarting the complete Blueprint container) if a new namespace handler with compatible version is available or becomes available at a later stage.
- Always restart the Blueprint container (update-strategy=restart-always). This is applicable for namespace handlers, which are required for runtime operations of the components they define. If no immediate replacement is available the Blueprint container will enter the GRACE_PERIOD to wait for a replacement to become available.

5.1.3 Versioning

Namespace handlers must be versioned. Each registered namespace handler must have a service property org.osgi.blueprint.namespace.version, which gives the version of the namespace. Namespaces like this need not obey the generic namespace versioning scheme [ns url]/v[ns version]. In addition namespace handlers define a generic namespace handler capability including the namespace url and version via a 4.3 Provide-Capability header for example for the namespace handler implementer:

```
Provide-Capability: org.osgi.service.blueprint.NamespaceHandler;
 osgi.service.blueprint.namespace:String=http://www.example.org/xmlns/cache;
 version:Version=1.0.0
```

And for the consumer:

```
Require-Capability: org.osgi.service.blueprint.NamespaceHandler;
 filter:="(&(osgi.service.blueprint.namespace=http://www.example.org/xmlns/cache)
 (version=1.0.0))"
```

In addition to using the generic namespace versioning scheme, clients can use an unversioned namespace URL like http://www.example.org/xmlns/cache and add a Require-Capability statement that restricts the version range of the namespace handler. The Blueprint container will honor the version range when selecting an available namespace handler.

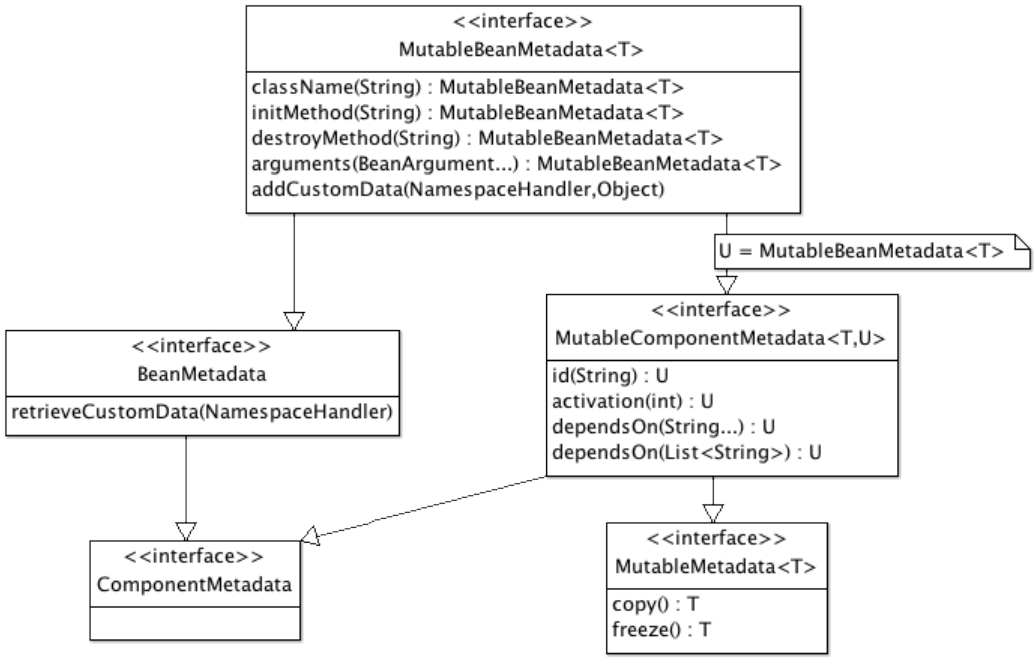
This would make this 4.3 only!

5.1.4 Blueprint API for mutable component metadata and Builder API

A typical concern for namespace handlers is to customize enclosing components for example by adding or replacing a property or a constructor argument. This can be achieved either by wrapping the component metadata or by having access to a mutable component metadata. Of the two a solution using wrapping is likely to introduce conflicts when combining multiple namespace handlers. Instead exposing an API for mutable component metadata would allow the vast majority of use cases while side-stepping most interaction problems between namespace handlers.

Another common concern of namespace handlers is the creation of components. Without a builder API for metadata objects, a namespace handlers has only the options of implementing the interfaces or generating a DOM tree to be handed off to the parser. Both options are not particularly attractive for ease of use, hence a builder API should be available to ease construction of metadata.

The option pursued in this draft is to cover mutable metadata and builder API using the same fluent interfaces. Both options could also easily be split out into a bean based solution for mutable metadata and a fluent interface solution for building component metadata. Example classes of this solution are shown below. Note that methods are omitted.



In the design shown, for every non-marker subinterface X of org.osgi.service.blueprint.reflect.Metadata there is a corresponding MutableX that inherits from X and MutableMetadata<X>. These instances of mutable metadata can be used to create as well as modify existing metadata. MutableMetadata can be obtained either by casting existing metadata objects if they implement the mutable interface or my creating one from a builder interface that will be passed to the a namespace handler. Thread safety guarantees inherited from the existing Blueprint reflection metadata interfaces must be maintained by the MutableMetadata interfaces and implementations.

Modification of metadata instances after the creation of the Blueprint container that contains them is meaningless. To prevent accidental use of this facility metadata instances can be frozen using the freeze method defined on MutableMetadata. This method will produce a copy of the metadata that allows no further modification. This method will be called by the Blueprint extender at the time where no further modification of the component is supported.

```
public interface Builder {
              MutableBeanMetadata<BeanMetadata> newBean();
              MutableBeanArgument<BeanArgument> newBeanArgument();
              MutableBeanProperty<BeanProperty> newBeanProperty();

              MutableReferenceMetadata<ReferenceMetadata> newReference();
              MutableReferenceListMetadata<ReferenceListMetadata> newRefList();
              MutableServiceMetadata<ServiceMetadata> newService();

              MutableRegistrationListener<RegistrationListener> newRegistrationListener();
              MutableReferenceListener<ReferenceListener> newReferenceListener();

              MutableRefMetadata<RefMetadata> newRef();
              MutableIdRefMetadata<IdRefMetadata> newIdRef();

              MutableCollectionMetadata<CollectionMetadata> newCollection();
              MutableMapMetadata<MapMetadata> newMap();
              MutablePropsMetadata<PropsMetadata> newProps();
              MutableValueMetadata<ValueMetadata> newValue();
}
```

Namespace handles that wish to define their own reflection metadata interfaces or provide their own metadata implementations should whenever possible extend or implement the appropriate mutable metadata interfaces. In such a way extension components can easily be handled by other namespace handlers without requiring any additional awareness of the presence of extended metadata.

Another added facility show in the above class diagram is that of custom namespace handler data. This allows a namespace handler to associate custom configuration data (for example transaction configuration in the case of the transaction namespace handler) that can be used later for intercepting method calls or otherwise enriching the bean. The custom data is associated to the lifecycle of the bean and needs no longer be managed by the namespace handler. {*Should this facility be present on ComponentMetadata rather than the more specific BeanMetadata?}*

{Is the old API (ParserContext#enhance) that allows the enclosing component metadata object to be replaced by the namespace handler even still really necessary if there is an API for mutable component metadata?}

5.1.5 Component Registry Processors

Component registry processors provide a namespace handler the opportunity to modify / influence all components in a Blueprint after the Blueprints have been parsed initially. Possible use case it to implement global defaults or install global interceptors (tracing would be one example).

Similar to type converters component registry processors are normal managers with limitations to the dependency handling. A registry processor component and its dependencies must not be processed by another registry processor or itself. A component registry processor is a bean instance with a class that implements the ComponentDefinitionRegistryProcessor (defined in the next section).

To operate a component registry processor is given access to the Blueprint container's component registry. From there it can add, modify and remove components.

{Should component registry processors be enclosed in a separate Blueprint tag like type converters? For example components instantiated by static or in particular dynamic factories cannot be handled by the mechanism described above.}

5.1.6 Additional APIs

Beside the main NamespaceHandler interface several supporting interfaces are needed for ParserContext, ComponentDefinitionRegistry, ComponentDefinitionRegistryProcessors etc.

{This however requires parsing to happen in an actual runtime rather than just parsing for the sake of obtaining the information. Maybe have an opt-in or opt-out mechanism for namespace handlers that can support parsing in circumstances where there is no real bundle available.}

```
public interface ParserContext  {
    /**
     * Returns the DOM Node that was passed to the NamespaceHandler call for which
     * this ParserContext instance was created.
     */
    Node getSourceNode();

    /**
     * Returns the Blueprint bundle for the currently processed Blueprint descriptor.
     * This method returns null if the Blueprint descriptor is only parsed but does not correspond to a
     * live bundle object.
     */
    Bundle getBundle();

    ComponentDefinitionRegistry getComponentDefinitionRegistry();

    /**
     * Retrieve the <code>ComponentMetadata</code> of the component that
```

```
     * encloses the current <code>Node</code> that is to be parsed by a
     * namespace handler.
     *
     * In case of top-level components this method will return <code>null</code>.
     * @returns the enclosing component's metadata or null if there is no enclosing component
     */
    ComponentMetadata getEnclosingComponent();

    /**
     * Retrieve a builder for constructing metadata instances using a flow-style API.
     */
    Builder getMetadataBuilder();

    /**
     * Invoke the blueprint parser to parse a DOM element.
     * @param type the class of the Metadata type to be parsed
     * @param enclosingComponent The component metadata that contains the Element
     * to be parsed
     * @param element The DOM element that is to be parsed
     * @param <T> The expected metadata type to be parsed
     */
    <T> T parseElement(Class<T> type, ComponentMetadata enclosingComponent, Element element);

    /**
     * Generate a unique id following the same scheme that the blueprint container
     * uses internally
     */
    String generateId();

    /**
     * Get the default activation setting for the current blueprint file
     */
    String getDefaultActivation();

    /**
     * Get the default availability setting for the current blueprint file
     */
    String getDefaultAvailability();

    /**
     * Get the default timeout setting for the current blueprint file
     */
    String getDefaultTimeout();
}

public interface ComponentDefinitionRegistryProcessor {

    /**
     * Process a <code>ComponentDefinitionRegistry</code>
     * @param registry
     */
    public void process(ComponentDefinitionRegistry registry);
}

public interface ComponentDefinitionRegistry  {

    /**
     * Determine if the component registry contains a component definition for the given id
     * @param id
     * @return
     */
    boolean containsComponentDefinition(String id);

    /**
     * Retrieve a component's metadata by id
     * @param id The id of the component. This is either the id specified in the Blueprint xml or the
     * generated id of an unnamed component
     * @return the <code>ComponentMetadata</code> or <code>null</code> if the id does not match
     * any registered component
```

```
     */
    ComponentMetadata getComponentDefinition(String id);

    /**
     * Returns a set of the id of top-level blueprint components (both named and unnamed).
     *
     * The ids of unnamed components are Blueprint generated. Anonymous components, which have no
     * id, are not part of the set.
     * @return
     */
    Set<String> getComponentDefinitionNames();

    /**
     * Register a new component
     *
     * The <code>ComponentMetadata</code> argument must have an id. So unnamed components should have an
id
     * generated prior to invoking this method. Also, no component definition may already be registered
     * under the same id.
     *
     * @param component the component to be registered
     * @throws IllegalArgumentException if the component has no id
     * @throws ComponentNameAlreadyInUseException if there already exists a component definition
     * in the registry with the same id
     */
    void registerComponentDefinition(ComponentMetadata component);

    /**
     * Remove the component definition with a given id
     *
     * If no component is registered under the id, this method is a no-op.
     * @param id the id of the component definition to be removed
     */
    void removeComponentDefinition(String id);

    void registerTypeConverter(Target component);

    List<Target> getTypeConverters();
}
```

5.1.7 Additional requirements on the Blueprint specification

- The Blueprint schema needs to be refactored so that elements such as beans are referenceable from extension schemas.

5.1.8 Custom scopes

Although not directly related to namespace handlers, custom scopes provide another axis of extensibility in the Blueprint container. Unfortunately, in the original Blueprint specification scopes leaves the details of custom scopes to the implementations (see section 121.5.5). For cross container compatibility this is an issue.

As clarification and specialization to the original specification scopes are always resolved to QNames by the Blueprint extender. So a scope such as for example “jee:session” is only valid if jee is bound to a namespace (say for example “http://example.org/xmlns/blueprint/jee”). In such a case the scope of the BeanMetadata must be returned as {http://example.org/xmlns/blueprint/jee}session. Invalid scope specifications must throw a ComponentDefinitionException during the parsing phase.

{What are we standardizing here namespaces or scopes? How far should we go with scopes prescribe an extension mechanism that can be used by scope implementers or just the outline of how to use scopes cross container?}

5.1.9 Security concerns

None !?

6 Considered Alternatives

RFC 124 proposed the following technical solution, closely modeled on the namespace support in Spring DM.

6.1 RFC 124 Namespace Extension Mechanism

Third parties may contribute additional namespaces containing elements and attributes used to configure the components for a module context. These additional namespaces are referenced in the standard XML manner using the xmlns attribute of the top level element. The following configuration file references the osgi, osgix, and aop namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:osgix="http://www.osgi.org/xmlns/blueprint-compendium/v1.0.0"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
      http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
      http://www.osgi.org/xmlns/blueprint-compendium/v1.0.0 http://www.osgi.org/xmlns/blueprint-
compendium/v1.0.0/blueprint-compendium.xsd
      http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-
2.5.xsd">

…

</blueprint>
```

In order to be able to interpret elements and attributes declared in third party namespaces, a *namespace handler* must be registered that the container can delegate to. Namespace handlers are registered in the OSGi service registry under the `org.osgi.service.blueprint.namespace.NamespaceHandler` interface. A registered namespace handler must also advertise the service property `osgi.service.blueprint.namespace`.The value of this property is an array of URIs for the schema that the namespace can handle (for example, http://www.springframework.org/schema/aop). When defining a namespace extension and schema the following conventions are recommended (but not required):

- A schema URI with a final path segment of the format “v[version]” – for example http://www.foobar.org/schema/foo/v1.0.0 references that version of the schema (1.0.0 in this case)
- A schema URI without such a final path segment references the latest version of the schema
- A namespace handler should publish in its osgi.service.blueprint.namespace property the URIs of each version of the schema it can handle, together with the unqualified URI if the handler can also handle the most recent version.

During the processing of a configuration file the container must determine the set of schemas used to define the elements in the configuration file. The resulting set of schema uris (for the above example this would be http://www.osgi.org/schema/service/blueprint, http://www.osgi.org/schema/service/blueprint-compendium, and http://www.springframework.org/schema/aop) determine the namespace handlers that must be present in order to process the file. For each schema uri the container creates a mandatory service reference for a service implementing the `NamespaceHandler` interface and with a matching `osgi.service.blueprint.namespace`

property value. Creation of the context waits for these service references to be satisfied in the same way as it would for any service reference explicitly declared via a `service` element – including generation of the wait events if an EventAdmin service is available and the services are not immediately available.

The NamespaceHandler interface is defined as follows:

```
interface NamespaceHandler {

   /**

    * The URL where the xsd file for the schema may be found. Typically used to return a URL to a

    * bundle resource entry so as to avoid needing to lookup schemas remotely.

    * If null is returned then the schema location will be determined from the xsi:schemaLocation
attribute

    * value.

    */

   URL getSchemaLocation(String namespace);



  /**

   * Called when a top-level (i.e. non-nested) element from the namespace is encountered.

   * Implementers may register component definitions themselves, and/or return a component definition

   * to be registered.

   */

   ComponentMetadata parse(org.w3c.dom.Element element, ParserContext context);



  /**

   * Called when an attribute or directly nested element is encountered. Implementors should parse the

   * supplied Node and decorated the provided component, returning the decorated component.

   */

   ComponentMetadata decorate(org.w3c.dom.Node node, ComponentMetadata component,

                              ParserContext context);

}
```

The `ParserContext` passed to the parse and decorate methods provides access to the `ComponentMetadata` of the enclosing component (if any) and to the `ComponentDefinitionRegistry`.

The `parse` callback is invoked when an element from the namespace is encountered that is not directly nested inside of a `bean`, `service`, `reference`, `ref-list`, or `ref-set` element. Most commonly it is invoked when a top-level namespace element is encountered. Very often a custom namespace element serves as a convenient shortcut for creating a component definition. In this case where one namespace element corresponds to one component definition, the simplest implementation is often to return the new component definition as the return value of the parse callback. In more complex cases, a single namespace element may map to a collection of component definitions, or conditionally create components. In such circumstances the context passed to the parse method can be used to discover existing component definitions and to explicitly register new ones.

The `decorate` callback is invoked when an attribute from the handled namespace is encountered in an element that is not from the handled namespace. Decorate typically modifies the component definition for the component in which the attribute was encountered, and returns the modified definition. The `decorate` callback is also invoked when an element from the handled namespace is encountered nestly directly inside a `component`, `service`, `reference`, `ref-list`, or `ref-set` element.

Consider the following configuration snippet involving a hypothetical “cache” namespace:

```
<cache:lru-cache id="myCache"/>                          (1)

<bean id="fooService" class="FooServiceImpl"
       cache:cache-return-values="true">                 (2)

    <cache:exclude>                                      (3)
      <cache:operation name="getVolatile"/>              (4)
    </cache:exclude>

    <property name="myProp" value="12"/>

</bean>



<bean id="barService" class="BarServiceImpl">

  <property name="localCache">

    <cache:lru-cache/>                                   (5)

  </property>

</bean>
```

Given a namespace handler registered to handle the cache namespace, then at:

(1) The `parse` method will be invoked, and will typically return a new component definition defining a cache component.

(2) The `decorate` method will be invoked passing in the `cache-return-values` attribute `Node`, and the `ComponentMetadata` for the `fooService` component.

(3) The `decorate` method will be invoked passing in the `exclude` element `Node`, and the `ComponentMetadata` for the `fooService` component.

(4) There is no callback for this line, it is the responsibility of the `decorate` method at (3) to process the contents of the `exclude` element

(5) The `parse` method will be invoked and would typically return a new component definition defining a cache component. From the `ParserContext` passed to the method, `getEnclosingComponent` will return the component definition for the `barService` component.

6.1.1 Date Namespace Example

The following example illustrates how the Blueprint Service could be extended to process elements from a "date" namespace. In our example, the date namespace contains only one element, `dateformat`. The namespace handler will enable component configuration files to include declarations such as:

```
<date:dateformat id="dateFormat"
          pattern="yyyy-MM-dd HH:mm"
          lenient="true"/>
```

and such declarations will in effect be equivalent to the following component definition:

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <argument value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

The basic steps to implement the handler are as follows:

1. Create the schema file for the extension, and package it inside a bundle
2. Write a NamespaceHandler implementation and fill in the parse method
3. Export the namespace handler as a service in the service registry

6.1.1.1 Creating and packaging the schema file

The namespace handler will be packaged inside its own bundle. Inside the bundle, at location schemas/date.xsd[1] is placed the following file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.com/schema/myns"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:osgi="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    targetNamespace="http://www.mycompany.com/schema/date/v1.0.0"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

   <xsd:import namespace="http://www.osgi.org/xmlns/blueprint/v1.0.0"/>

   <xsd:element name="dateformat">
      <xsd:complexType>
         <xsd:complexContent>
            <xsd:extension base="osgi:identifiedType">
               <xsd:attribute name="lenient" type="xsd:boolean"/>
               <xsd:attribute name="pattern" type="xsd:string" use="required"/>
```

[1] The location given here is just for illustration, you can place the file anywhere inside the bundle that you choose

```
            </xsd:extension>
          </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

The bold line shows that our new `dateformat` element extends the basic `identifiedType` element from the osgi blueprint schema, meaning that it will have an `id` attribute.

6.1.1.2 NamespaceHandler implementation

The complete source code for the DateNamespaceHandler implementation is shown below:

```
package org.osgi.service.blueprint.namespace.example;

import java.net.URL;
import java.text.SimpleDateFormat;

import org.osgi.framework.BundleContext;
import org.osgi.service.blueprint.namespace.NamespaceHandler;
import org.osgi.service.blueprint.namespace.ParserContext;
import org.osgi.service.blueprint.namespace.example.builder.MutableLocalComponentMetadata;
import org.osgi.service.blueprint.reflect.ComponentMetadata;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

/**
 * Sample handler for a "date" namespace based on the example given here:
 * http://static.springframework.org/spring/docs/2.5.x/reference/extensible-xml.html
 *
 * Handles "dateformat" elements with the following form:
 *
 * <date:dateformat id="dateFormat"
 *   pattern="yyyy-MM-dd HH:mm"
 *   lenient="true"/>
 *
 */
public class DateNamespaceHandler implements NamespaceHandler {

            /**
             * The schema file is packaged in the same bundle as this handler class
             * for convenience, in schemas/date.xsd
             */
            private static final String SCHEMA_LOCATION = "schemas/date.xsd";

            private final BundleContext bundleContext;

            public DateNamespaceHandler(BundleContext context) {
                        this.bundleContext = context;
            }

            /**
             * Use the bundleContext to return a URL for accessing the schema definition file
             */
            public URL getSchemaLocation(String namespace)  throws IllegalArgumentException {
                        return bundleContext.getBundle().getResource(SCHEMA_LOCATION);
            }

            /**
```

```
     * Date elements can never be used nested directly inside a component, so
     * nothing for us to do.
     */
    public ComponentMetadata decorate(Node node, ComponentMetadata component,
                              ParserContext context) {
                return null;
    }


    /**
     * Handle the "dateformat" tag (and others in time...)
     */
    public ComponentMetadata parse(Element element, ParserContext context) {

      if (element.getLocalName().equals("dateformat")) {
      return parseDateFormat(element);
      }
      else {
      throw new IllegalStateException("Asked to parse unknown tag: " +
                element.getTagName());
      }
    }

    /**
     * A dataformat element defines a component, which could be a top-level named
     * component or an anonymous inner component.
     */
    private ComponentMetadata parseDateFormat(Element element) {
      String name = element.hasAttribute("id") ? element.getAttribute("id") : "";
      MutableLocalComponentMetadata componentMetadata =
        new MutableLocalComponentMetadata(name,SimpleDateFormat.class.getName());

      // required attribute pattern
      String pattern = element.getAttribute("pattern");
      componentMetadata.addConstructorArg(pattern,0);

      // this however is an optional property
      String lenient = element.getAttribute("lenient");
      if ((lenient != null) && !lenient.equals("")) {
         componentMetadata.addProperty("lenient", lenient);
      }

      return componentMetadata;
    }

}
```

This implementation relies on a helper class, MutableLocalComponentMetadata that we use to create an instance of the LocalComponentMetadata interface. Future versions of this specification may standardize such implementation classes, or an equivalent builder API.

```
package org.osgi.service.blueprint.namespace.example.builder;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.osgi.service.blueprint.reflect.ComponentMetadata;
import org.osgi.service.blueprint.reflect.ConstructorInjectionMetadata;
import org.osgi.service.blueprint.reflect.LocalComponentMetadata;
import org.osgi.service.blueprint.reflect.MethodInjectionMetadata;
```

```
import org.osgi.service.blueprint.reflect.ParameterSpecification;
import org.osgi.service.blueprint.reflect.PropertyInjectionMetadata;
import org.osgi.service.blueprint.reflect.TypedStringValue;
import org.osgi.service.blueprint.reflect.Value;

/**
 * "Just enough" implementation of a mutable LocalComponentMetadata to meet the
 * needs of our namespace handler...
 */
public class MutableLocalComponentMetadata implements LocalComponentMetadata {

		private String name;
		private String className;
		private String initMethodName = "";
		private String destroyMethodName = "";
		private ComponentMetadata factoryComponent = null;
		private final List<ParameterSpecification> constructorSpec =
		   new ArrayList<ParameterSpecification>();
		private final List<PropertyInjectionMetadata> propertiesSpec =
		   new ArrayList<PropertyInjectionMetadata>();;
		private MethodInjectionMetadata factoryMethodMetadata = null;
		private String scope = "singleton";
		private boolean isLazy = false;
		private Set<String> dependencies = new HashSet<String>();

		public MutableLocalComponentMetadata(String name, String className) {
			this.name = name;
			this.className = className;
		}

		public String getName() {
			return name;
		}

		public void setName(String name) {
			this.name = name;
		}

		public String getClassName() {
			return className;
		}

		public void setClassName(String className) {
			this.className = className;
		}

		public ConstructorInjectionMetadata getConstructorInjectionMetadata() {
			return new ConstructorInjectionMetadata() {

				public List getParameterSpecifications() {
					return constructorSpec;
				}

			};
		}

		public void addConstructorArg(ParameterSpecification spec) {
			constructorSpec.add(spec);
		}

		// convenience method for String-based values
		public void addConstructorArg(final String value , final int index) {
			constructorSpec.add(new ParameterSpecification() {

				public int getIndex() {
					return index;
				}

				public String getTypeName() {
```

```
                                        return null;
                                }

                                public Value getValue() {
                                        return stringValue(value);
                                }

                        });
        }

        public Collection getPropertyInjectionMetadata() {
                return propertiesSpec;
        }

        public void addProperty(final String name, final Value value) {
                propertiesSpec.add(new PropertyInjectionMetadata() {

                                public String getName() {
                                        return name;
                                }

                                public Value getValue() {
                                        return value;
                                }
                        });
        }

        public void addProperty(final String name, final String value) {
                propertiesSpec.add(new PropertyInjectionMetadata() {

                                public String getName() {
                                        return name;
                                }

                                public Value getValue() {
                                        return stringValue(value);
                                }
                        });
        }

        public String getInitMethodName() {
                return initMethodName;
        }

        public void setInitMethodName(String initMethodName) {
                this.initMethodName = initMethodName;
        }

        public String getDestroyMethodName() {
                return destroyMethodName;
        }

        public void setDestroyMethodName(String destroyMethodName) {
                this.destroyMethodName = destroyMethodName;
        }

        public ComponentMetadata getFactoryComponent() {
                return factoryComponent;
        }

        public void setFactoryComponent(ComponentMetadata factoryComponent) {
                this.factoryComponent = factoryComponent;
        }

        public MethodInjectionMetadata getFactoryMethodMetadata() {
                return factoryMethodMetadata;
        }

        public void setFactoryMethodMetadata(
```

```
                                        MethodInjectionMetadata factoryMethodMetadata) {
                        this.factoryMethodMetadata = factoryMethodMetadata;
                }

                public String getScope() {
                        return scope;
                }

                public void setScope(String scope) {
                        this.scope = scope;
                }

                public boolean isLazy() {
                        return isLazy;
                }

                public void setLazy(boolean lazy) {
                        this.isLazy = lazy;
                }

                public Set getExplicitDependencies() {
                        return dependencies;
                }

                public void addDependency(String name) {
                        dependencies.add(name);
                }

                public void removeDependency(String name) {
                        dependencies.remove(name);
                }

                private TypedStringValue stringValue(final String val) {
                        return new TypedStringValue() {

                                        public String getStringValue() {
                                                return val;
                                        }

                                        public String getTypeName() {
                                                return null;
                                        }};
                }
}
```

Such an implementation is easily generated using a modern IDE.

6.1.1.3 Registering the namespace handler

With the DateNamespaceHandler and MutableLocalComponentMetadata classes packaged up in the bundle classpath, the last thing to do is ensure that the namespace handler is registered as a service when the bundle is started. The easiest way to do this is to use the blueprint service. In `OSGI-INF/blueprint` create a file called e.g. `module-context.xml` with the following declarations:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.osgi.org/xmlns/blueprint/v1.0.0
      http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

  <bean id="dateNamespaceHandler"
        class="org.osgi.service.blueprint.namespace.example.DateNamespaceHandler"/>
```

```
  <service ref="dateNamespaceHandler"
     interface="org.osgi.service.blueprint.namespace.NamespaceHandler">
    <service-properties>
      <entry key="osgi.service.blueprint.namespace">
        <array value-type="java.lang.String">
            <value>http://www.mycompany.com/schema/date/v1.0.0</value>
            <value>http://www.mycompany.com/schema/date</value>
        </array>
      </entry>
    </service-properties>
  </service>

</blueprint>
```

7 Security Considerations

None.

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 124 v1.0 available in the OSGi Alliance subversion repository in rfcs/rfc0124/rfc-124.pdf.

[4]. RFC 156 is available in the OSGi Alliance subversion repository in rfcs/rfc0156/rfc-156-blueprint-config-admin.pdf.

[5]. OSGi Release 4 Version 4.2 Service Compendium.

[6]. Namespaces in XML 1.0 (Third Edition): http://www.w3.org/TR/REC-xml-names/.

[7]. RFC 164 is available in the OSGI Alliance subversion repository in rfcs/rfc0164/rfc-164-blueprint-transaction.pdf.

[8]. RFC 166 is available in the OSGi Alliance subversion repository in rfcs/rfc0166/rfc-0166-BlueprintBeanInterceptors.pdf

8.2 Author's Address

Name	Glyn Normington
Company	VMware
Address	SpringSource, Kenneth Dibben House, Enterprise Road, Chilworth, Southampton SO16 7NS, UK.
Voice	+44-2380-111512
e-mail	gnormington@vmware.com

Name	Valentin Mahrwald
Company	IBM
Address	IBM UK Limited, Hursley Park, Winchester, Hampshire, SO21 2JN
Voice	+44-1962 818394
e-mail	mahrwald@uk.ibm.com

8.3 Acronyms and Abbreviations

8.4 End of Document