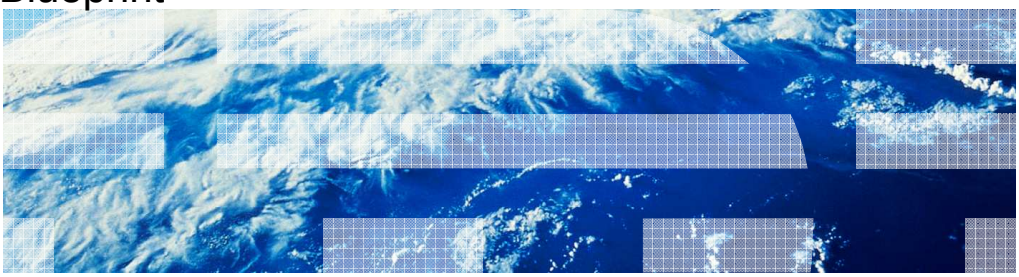
IBM WebSphere Application Server Feature Pack for OSGi Applications and Java Persistence API 2.0

Blueprint



WebSphere software

This module provides education on Blueprint for the IBM WebSphere® Application Server Feature Pack for OSGi Applications and Java™ Persistence API 2.0

Table of contents

This is a quick overview of the items that are covered within this presentation.

The Big Picture section is a very high level look at Blueprint and its function.

In the details section you will learn the source of the Blueprint Specification, where it is obtainable from, how it can be useful and so on.

Then you will learn about the source of the implementation in the Feature Pack.

Next, you will cover as the key concepts of Blueprint, touching a little on how they can be used.

Finally, you will take a quick look at troubleshooting, where you will learn a few common problems, mistakes, and pitfalls.

Section

Big picture overview

This section will cover the overview.

The big picture (1 of 4)

- **Blueprint**
  - What is Blueprint? – The 10,000 mile high view.
  - What does Blueprint do?
  - Where does Blueprint come from?

In this section you will be informed of Blueprints purpose, a little on how Blueprint gets its job done, and touching on where it comes from, both specification, and implementation.

The big picture (2 of 4)

- **What is Blueprint?**
  - Dependency injection for OSGi bundles
  - Inversion of control
  - XML defined wiring for bundle contained components

5 Blueprint

So, what is Blueprint?

Blueprint can be described as dependency injection or inversion of control or XML defined wiring for bundle contained components.

Dependency injection means that Blueprint injects dependencies to components.

In a traditional model, a component would contain code that reached out, looked up, and used its dependencies. Blueprint is all about enabling the removal that sort of code. Instead, the component provides setters for its dependencies, and Blueprint will invoke them, passing the dependency as configured by the Blueprint configuration. This means the component can concentrate more on performing its business logic, and less on the technicalities of how to use OSGi Services, or looking up and obtaining other dependencies.

As a rather nice side effect, removing this logic from components can enable them to be hosted in a test environment, where the dependencies injected are just stubs. This provides for easier testing of components.

Dependency injection, when used in this manner, can be considered an ‘Inversion of Control’, as the framework (Blueprint) is now handling the dependency acquisition, rather than the component itself.

Finally, Blueprint is configured in XML; a component wanting to make use of Blueprint instructions by providing XML within itself.

The big picture (3 of 4)

- **What does Blueprint do?**
  - Looks after ‘Blueprint Bundles’
  - Uses XML to configure Blueprint concepts…
    - Beans
    - Services
    - References to services
    - Constant values

So, having described what Blueprint is, how does it work?

The quick one line answer is that Blueprint uses an “Extender Pattern” in OSGi.

This means that Blueprint is sitting there, watching for Bundles to be activated, and then checking if the Bundle contains Blueprint XML.

A bundle is considered to be a Blueprint bundle when it contains one or more Blueprint XML files. These XML files are at a fixed location under the OSGI-INF/blueprint/ directory or are specified explicitly in the Bundle-Blueprint manifest header.

Once the extender determines that a bundle is a Blueprint bundle, it creates a Blueprint Container on behalf of that bundle. The Blueprint Container is responsible for passing the Blueprint XML files, instantiating components and wiring the components together

Components can be beans, services, references to services, or values. There is more on this in the details section.

The big picture (4 of 4)

- **From where does Blueprint come?**
  - OSGi Service Platform Release 4 V4.2
    - Enterprise specification
    - "Blueprint container specification 1.0“
  - Implementation from Apache Aries
    - Snapshot of consumed version held for service purposes
  - Specification based on the ‘Spring Dynamic Modules’ project
    - Many similar concepts

Blueprint is a part of the OSGi enterprise specification. More precisely, the “OSGi Service Platform, Release 4, V4.2 – Enterprise Specification”

The specification can be found at www.osgi.org. When you go to that URL look for the specifications from the left navigation panel.

Section 121, page 201 onwards defines the “Blueprint Container Specification 1.0”.

It is this specification that defines how the Blueprint implementation should behave.

This module contains the basic concepts from the specification, as implemented in the feature pack, throughout the details section.

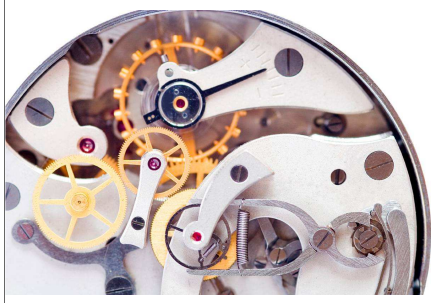
The implementation of Blueprint being used within the feature pack is from the Apache Aries incubator project (originally from Apache Geronimo).

A snapshot of the code consumed is held for service purposes, OSSC approval is required for this code.

Finally, the Blueprint Specification is derived from the Spring Dynamic Modules project. Indeed, the specification itself states this.

Section

Component details



This section covers component details.

Blueprint details (1 of 26)

- Overview
  - Managers/Components
    - Beans, services, references to services
  - Defining values
    - References, values, collections, type conversion
  - Environmental managers
    - Blueprint provided managers
  - XML definitions
    - Namespaces, global configuration attributes, locations

Blueprint occupies nearly 100 pages of the Enterprise specification, clearly the entire work is too lengthy to cover in a module of this sort. The aim here is to demonstrate the key concepts of Blueprint, with the occasional tip IBM developers and testers hit during development, or from the usage seen so far. Remember this is a new specification and a new implementation.

Covered here are Managers and Components, the basic top level blobs in Blueprint, these divide down into Beans, Services, and References to Services; Defining Values.

Next is how Blueprint can inject values into Components, including references to other Components, Fixed Values, Collections, and Type Conversion

There will also be information on Environmental Managers, A few blobs provided by Blueprint.

Finally a little further information is presented on what can be controlled by the XML.

Blueprint details (2 of 26)

- Managers/Components – Beans.
  - Bean overview
    - Blueprint can specify how to
      - Construct a bean
      - Control a beans scope
      - Invoke life cycle methods on a bean
  - Bean construction
    - Class constructor
    - Static factory method
    - Instance factory method
  - Bean scope
    - Singleton, prototype
  - Bean life cycle
    - Initialization, destruction

Managers are the things that make the components. The manager is defined by the XML, and creates the appropriate component, as instructed by the XML.

Beans are the way Blueprint represents a Java class.

Blueprint lets you configure beans to be published as OSGi services, injected into other beans, and have other components injected into it

Before you can think about that though, there is the small issue of bean construction. Once that is understood there is the question of how many specific beans there should be and how a bean knows it is being constructed or about to be destroyed.

Those points are covered over the next few slides.

Blueprint details (3 of 26)

- Managers/Components – Beans.
  - Bean construction
    - Class constructor

```
public class Account {
  public Account(long number) { ... }
}

<bean id="accountOne" class="Account">
  <argument value="1"/>
</bean>
```

      - Blueprint matches the constructor on a class using an algorithm extensively defined in the specification

11 Blueprint

Beans being Java classes, and Blueprint using XML for its configuration, means a bit of XML is needed to explain the Blueprint and how to instantiate a bean.

Blueprint supports three ways to construct a Bean, this being the first of those three, invoking the constructor on the class directly.

This is as simple as telling Blueprint the class to use by way of the class attribute on the bean element. In the example here it is just ‘Account’, in normal usage this would be a fully qualified class name, like ‘my.package.MyClass’.

The arguments to be passed to the constructor (if any) are specified using “argument” elements inside the bean element.

In the example, notice that the argument is specifying its value as a string - “1” - yet the constructor requires ‘long’, Blueprint handles the conversion before invoking the constructor.

When multiple constructors are available, the specification defines an algorithm to be followed, see section 121.9.1 ‘Signature Disambiguation’

Blueprint details (4 of 26)

- Managers/Components – Beans.
  - Bean construction
    - Static factory constructor

```
public class StaticAccountFactory {
 public static Account createAccount(long number){
  return new Account(number);
 }
}

<bean id="accountTwo" class="StaticAccountFactory"
      factory-method="createAccount">
  <argument value="1"/>
</bean>
```

12 Blueprint

The second way Blueprint can construct a bean is by way of using a static Factory Method.

Factories are common in Java, and Blueprint supports their usage directly, through the 'factory-method' attribute on bean.

When using a factory, the class attribute points at the factory class, and the factory-method contains the name of the method to invoke to create an instance of the bean. This method must be static on the factory class.

Once again, arguments can be passed, this time to the construction method.

The construction method is only specified by name, once again, if multiple methods are present with the same name (but with differing arguments), Signature Disambiguation as declared in the spec is used to select which is invoked.

Blueprint details (5 of 26)

- Managers/Components – Beans.
  - Bean construction
    - Factory instance constructor

```
public class AccountFactory {
  public AccountFactory(String factoryName){ ... }
  public Account createAccount(long number) {
    return new Account(number);
  }
}

<bean id="factory" class="AccountFactory">
  <argument value="account factory"/>
</bean>
<bean id="accountThree" factory-ref="factory"
      factory-method="createAccount">
  <argument value="1"/>
</bean>
```

13 Blueprint

Finally, Beans can be constructed using a Factory Instance.

A Factory instance is another class, with a non static method, invoked to obtain an instance of the class required.

In Blueprint, you represent the Factory using its own bean, in the example, the bean has the ID ‘factory’, and is specified as being the Java class ‘AccountFactory’.

In addition here, this example is initializing the factory bean, by passing an argument to its constructor. Effectively, here the factory bean is being constructed using the first method discussed previously, that of directly invoking the classes constructor.

Then, to declare the bean manager for the actual bean, the example uses the ‘factory-ref’ attribute to point to the factory bean, and use the ‘factory-method’ attribute to tell Blueprint the method to invoke on the factory bean to obtain an instance of the bean required.

Once again, the example also shows passing an argument to the create method, to show it can be done.

Blueprint details (6 of 26)

- Managers/Components – Beans.
  - Bean scope
    - Set using scope attribute.

```
<blueprint>
  <bean id="aBean" class="ABeanClass" scope="prototype" >
  </bean>
</blueprint>
```

    - Singleton
      - A single instance of the bean is created, and is used wherever that bean is required.
      - Singleton is the default for beans declared at the top level in the xml.
    - Prototype
      - A new instance is created for each usage.

14 Blueprint

After knowing how to create beans the inevitable question that follows is, “How many beans do I want?”

Blueprint provides the scope mechanism, to allow configuration of how many beans should be created.

Specifying scope “singleton” will cause the same instance of the bean to be used for every place the bean is used in Blueprint.

Scope prototype, alternatively will result in a new bean instance for every usage.

A few caveats, “singleton” is the default for beans declared at the ‘top level’ of the xml, that is immediately inside the root Blueprint element. The default for any Beans declared inline within other components is “prototype”.

Also, any Bean used to provide a service, will only have a single instance used to provide that service, even if the bean is declared as prototype scope. More on services will follow.

Blueprint details (7 of 26)

- Managers/Components – Beans.
  - Bean life cycle
    - Initialization
    - Destruction

```
public class Account {
  public Account(long number) { ... }
  public void init() { ... }
  public void destroy() { ... }
}

<bean id="account" class="Account"
   init-method="init" destroy-method="destroy">
  <argument value="6"/>
</bean>
```

15 Blueprint

Finally for beans, a bean may need information on when it has been created, or about to be destroyed.

Blueprint provides for this, by way of the ‘init-method’ and ‘destroy-method’ attributes, which specify method names on the bean that Blueprint will invoke once all the properties have been injected (for init), and when the Blueprint Container is destroying the object instance.

Destroy-method is not supported for prototype scope, it is up to the application to handle the destruction in this case.

Blueprint details (8 of 26)

- Managers/Components – Services.
  - Services overview
    - Lazy creation / proxies
    - Service interface registration
    - Service properties
    - Service ranking
    - Registration listeners

Blueprint makes it possible to register Services in the OSGi Service Registry, without needing to include all the code required to handle that inside the application.

As OSGi offers flexibility in service registration, that flexibility is provided in Blueprint, which means more options to explain…

Blueprint details (9 of 26)

- Managers/Components – Services.
  - Lazy creation / proxies
  - XML example..

```
<service id="serviceOne" ref="account" ... />

<service id="serviceTwo" … >
  <bean class="Account">
    <argument value="123"/>
  </bean>
</service>
```

17 Blueprint

When Blueprint registers a service, it actually registers a proxy, not the actual service.

This allows Blueprint to delay creation and instantiation of the bean backing the service until the service is actually required.

It also enables Blueprint to have the ability to deregister a Service, if the service's dependencies become unsatisfied.

From a usage perspective, this is transparent. A service is seen in the service registry, and acts as expected when used.

The XML example is missing the part that specifies how the service should be seen in the registry. There is more on that in the following foils.

Blueprint details (10 of 26)

- Managers/Components – Services.
  - Service interface registration
    - Interface

```
<service id="serviceTwo" interface="Account">
  <bean class="AccountImpl"/>
</service>
```

    - Interfaces

```
<service id="serviceTwo" ref="AccountImplId">
  <interfaces>
    <value>java.io.Serializable</value>
  </interfaces>
</service>
```

    - Auto-export

```
<service id="serviceTwo"
auto-export="interfaces"
        ref="AccountImplId"/>
```

18 Blueprint

When registering a service in an OSGi registry, it has to be registered under the interface, or interfaces that it implements.

Blueprint allows fine control over which interfaces you publish its services under.

For a single interface, it offers the simple ‘interface’ attribute.

For multiple interfaces, if provides the ‘interfaces’ element, which can contain multiple value elements.

And finally, there is the magic ‘auto-export’ attribute, which supports a variety of options.

When auto-export includes the **disabled** option, which is the default value, if the auto-export attribute is not specified. The list of interfaces must be specified using the interface attribute or interfaces sub-elements.

When it includes the **interfaces** option, it will register the service using all public interfaces implemented by the service class and any of its super classes.

If the option is **class-hierarchy** it will register the service using the service class and any of its public super classes.

If the **all-classes** option is used, it combines interfaces and class-hierarchy options.

Blueprint details (11 of 26)

- Managers/Components – Services.
  - Service properties and ranking

```
<service id="serviceFour"
        ref="myAccount"
auto-export="all-classes"
    ranking="3">
  <service-properties>
    <entry key="mode" value="shared"/>
    <entry key="active">
      <value type="java.lang.Boolean">true</value>
    </entry>
  </service-properties>
</service>
```

Just as OSGi allows services to be published with additional properties, and with a service ranking, so does Blueprint.

Blueprint provides the service-properties element, in which multiple entry elements can be placed, containing the properties that should be published along with the service.

Additionally, Blueprint allows the rank of a service to be set by way of the ‘ranking’ attribute. Ranking is used when looking up services, if multiple viable matches are found, the one with the highest ranking value is selected. Ranking defaults to 0 if not set.

Blueprint details (12 of 26)

- Managers/Components – Services.
  - Registration listeners

```
public class RegistrationListener {
  public void register(Account a, Map p) { ... }
  public void unregister(Account a, Map p) { ... }
}

<service id=”serviceSix”
        ref=”myAccount”
auto-export=”all-classes”>
  <registration-listener
     registration-method=”register”
     unregistration-method=”unregister”>
       <bean class=“RegistrationListener”/>
  </registration-listener>
</service>
```

20 Blueprint

As Blueprint uses proxies to delay registration of the real service, and enable the registering and unregistering as a service’s dependencies become available or unavailable, it provides RegistrationListeners through which Blueprint code can be notified when the publish or unregistering occurs.

The example shows how a registration-listener element can be declared to specify a bean that provides methods identified by the registration-method and unregistration-method attributes.

The signature for the registration and unregistration callback methods depends on whether the service object implements the ServiceFactory interface. If the service implements the ServiceFactory interface, both methods must have a void anyMethod(ServiceFactory, Map) signature, where anyMethod represents an arbitrary method name.

If the service does not implement the ServiceFactory interface, both callback methods must match the void anyMethod(? super T, Map) signature, where the type T is assignable from the service object's type. The first argument to the callback method is an instance of the service object, and the second argument is the registration properties. If a registration listener has multiple overloaded methods for a callback, then every method with a matching signature is invoked.

Blueprint details (13 of 26)

- Managers/Components – Service references.
  - Service references overview
    - Reference versus reference list
    - Matching a service
    - Optional and mandatory
    - Reference listeners

So far, this module has covered beans, and how to publish beans as services.

Next is how Blueprint allows you to make use of OSGi services, by way of dependency injection.

Blueprint allows you to inject an instance of a service directly into a bean, and (optionally) require the service to be available before the bean should be created.

Additionally you can specify additional options to match a specific service, for when a simple Interface match is insufficient, or you can use Blueprint to track all services matching a given interface. Use of Reference Listeners enables a Blueprint component to track the dynamic comings and goings of services within the OSGi Registry.

Blueprint details (14 of 26)

- Managers/Components – Service references.
  - Reference versus reference list
    - Differences.
      - Proxy blocking versus non-blocking unavailable case
      - Configuration of timeout. (0 means indefinite)
      - Proxy versus reference in a list.

```
<reference id="serviceRef"
    interface="Account" timeout="5000"/>

<reference-list id="serviceRefList"
    interface="Account"
    member-type="service-object"/>
```

Blueprint offers two ways of connecting to OSGi Services, Reference and Reference List.

The second is not merely a list of the first, there are some functional differences between the two.

Reference is intended to link to a single service, even if there are multiple matches for the service, and if the one it chose goes away, and there is an alternative still present, it will switch to using that service instead. From the applications point of view, this is transparent, it just sees the service as always there, and does not need to concern itself with which particular implementation is actually being used at any given time. Calls made while there are no matching services, will block until one becomes available, or until when a timeout occurs.

The example here, serviceRef, will match any service implementing the “Account” interface, and will wait a maximum of five seconds before timeout if there is no match.

“Reference List” is Blueprints way of allowing you to see all the potential matches for a service and optionally be informed when new ones come along or when existing ones go away. This notification is by way of “Reference Listeners”. There is more on this to come.

The List object created by a reference-list is dynamic, and will grow and shrink as matching services are added or removed from the Service Registry. Entries in a Reference List are not just ‘References’, each entry will map to only one actual service, and will not switch to a new implementation if its selected one goes away. For this reason the entries in a Reference List have no timeout, and will create a Service Unavailable exception immediately if an attempt is made to use one where its backing service has gone.

Additionally Reference Lists provide flexibility in the type of the object within the list. The example is using member-type “service-object”, which injects an object that implements the service, the alternative is to specify member-type of “service-reference”, which will inject OSGi Service References instead, where the application code will then get to do the lookup from the ServiceReference to a service implementation.

Blueprint details (15 of 26)

- Managers/Components – Service references.
  - Matching a service
    - Interface
    - Filter
    - Component-name

```
<reference  id="serviceRef"
     interface="Account"
component-name="otherComponentId"
        filter="(top=high)"/>
```

Sometimes, matching a service based just on an interface alone, is not enough.

Perhaps there are lots of potential matching services out there, but you know in advance the one you require was published with a property of ‘top’ set to value ‘high’.

Blueprint allows you to specify a filter string, with the content specified as an OSGi filter expression, to help refine the match further.

The component-name attribute can be used to match a service provided by another Blueprint component, which can be a quick way to quickly match a service provided within the same Blueprint bundle, where both the usage and publishing are being managed by Blueprint.

Blueprint details (16 of 26)

- Managers/Components – Service references.
  - Optional and mandatory
    - Only considered during initialization.

```
<reference  id="serviceRef"
     interface="Account"
       timeout="30000"
  availability="mandatory"/>
```

24 Blueprint

A service reference manager can require that at least one service that matches its selection criteria must exist before the Blueprint Container initialization can continue. The requirement is controlled with the availability attribute. The availability attribute can have two values, “optional” and “manditory”.

If **optional** is used matching the selection criteria may or may not exist.

If **Mandatory** is used at least one service matching the selection criteria must exist.

By default, mandatory availability is assumed. The default availability setting can be changed for all service reference managers in the Blueprint XML using a default-availability attribute on the Blueprint element.

A service reference manager with mandatory availability that has a matching service is considered satisfied. A service reference manager with optional availability is always considered satisfied even if it does not have any matching services. The Blueprint Container initialization will be delayed until all mandatory service reference managers are satisfied.

It is important to understand that mandatory availability is only considered during Blueprint Container initialization. After initialization, the mandatory service reference can become unsatisfied as services come and go at any point.

Blueprint details (17 of 26)

- Managers/Components – Service references.
  - Reference listeners
    - bind/unbind, all matching methods are called.
    - List versus non-list behavior
    - stateful services
      - use list. else will not see when proxy switches behind the scenes.

```
public class ReferenceListener {
  public void bind(ServiceReference r) { ... }
  public void bind(Account service) { ... }
  public void unbind(ServiceReference reference) { ... }
}
<reference-list id="refList" interface="Account"
                availability="optional">
  <reference-listener bind-method="bind"
                      unbind-method="unbind">
     <bean class="ReferenceListener"/>
  </reference-listener>
</reference-list>
```

25 Blueprint

Reference Listeners can be used in both Reference and Reference List. The example here is using reference-list, although it will work equally well with a reference.

A Reference Listener can provide Bind and Unbind methods, which will be invoked when a Service is selected to be used, or when a Service is no longer used.

If a Reference Listener has multiple matching methods (as the example does for 'bind') then ALL matching methods will be invoked.

Both bind and unbind callback methods can have any of the following signatures where "anyMethod" represents an arbitrary method name.

If the signature is **void anyMethod(ServiceReference)** then the argument is a ServiceReference object of the service being bound or unbound.

If the signature is **void anyMethod(? super T)** then the argument is the service object proxy being bound or unbound. The type T must be assignable from the service object.

If the signature is **void anyMethod(? super T, Map)** then the first argument is the service object proxy being bound or unbound. The type T must be assignable from the service object. The second argument provides the service properties associated with the service.

For a Reference List, the callback methods are invoked whenever a service is added or removed from the List.

For a Reference, the callbacks are only invoked when the service transitions from having no backing service, to having one, and from having one to not having one. It is important to note that the callbacks will NOT be invoked when a Reference switches backing services as from the applications view point, there is still a Service it can use.

Thus when operating with stateful services, you will want to use a Reference List, to be able to track if the service you are interacting with goes away.

Blueprint details (18 of 26)

- Defining values.
  - References
    - Usage of other defined values, or components
  - Values
    - Constants, typing
  - Collections
    - Maps, lists, sets, arrays, properties
  - Type conversion
    - Blueprint type converters

The Blueprint Container specification defines many XML elements that describe different types of object values. These XML value elements are used within the manager definitions. For example, they can be used in a bean manager to specify argument or property values, or in a service manager to specify the service properties' values. The XML value elements are converted into the actual value objects and injected into the manager components.

This section will cover the various elements that Blueprint supports, with quick examples.

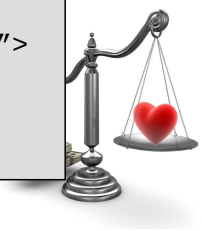
Blueprint details (19 of 26)

- Defining values - References
  - Usage of other defined values, or components

```
public class AccountManager {
  ...
  public void setManagedAccount(Account a) { ... }
}

<bean id="accountOne" class="Account">
  <argument value="1"/>
</bean>

<bean id="accountManager" class="AccountManager">
  <property name="managedAccount">
    <ref component-id="accountOne"/>
  </property>
</bean>
```



27 Blueprint

In this example the ‘ref’ element is used. It is highlighted in brown toward the end of the XML

The ‘ref’ element lets you use another Blueprint component as the value to be injected to a property, or used as a constructor argument.

In the example, this example is injecting an instance of the ‘Account’ class into the Account Manager, by referencing the Account bean’s id.

Blueprint also provides a very similar element “idref”, where ref causes the Component to be injected.. Using “idref” merely injects the id. If ”idref” were used in the example instead of ref, the string ‘accountOne’ would be injected as a string to the Account Manager, this is useful to ensure that all the dependencies of another component have been satisfied before the using component is activated.

Blueprint details (20 of 26)

- Defining values - Values
  - Values
    - Constants
    - Typing

```
<bean id="acct" class="Account">
  <property name="desc" value="#1 account"/>
</bean>

<bean id="acct" class="Account">
  <property name="desc">
    <value>#1 account</value>
  </property>
  <property name="acctNumber">
    <value type="java.math.BigInteger">456</value>
  </property>
</bean>
```

28 Blueprint

Properties can be specified as attributes, or elements..

When using elements, the content of the <value> element is used as the value, and the value element can use the “type” attribute to specify a type into which the content should be converted.

If the type is not specified, the content will be converted to the type it is being injected into.

Blueprint details (21 of 26)

- Defining values - Collections
  - Collections
    - Null, lists, sets, arrays

```
<list>
  <value>123</value>
  <value type="java.math.BigInteger">456</value>
  <null/>
  <set value-type="java.lang.Integer">
    <value>1</value>
    <value>2</value>
  </set>
  <array value-type="java.lang.String">
    <value>FISH</value>
  </array>
</list>
```

Here you see some simple examples of various Blueprint collections..

The list element represents a java.util.List object, the set element represents a java.util.Set object, here the set specifies that its content should be converted to java.lang.Integer. The array element represents Object [ ] in this case, with a specified type of java.lang.String [ ]

Finally, hidden in this example is <null/> which represents Java null.

Blueprint details (22 of 26)

- Defining values - Collections
  - Collections
    - Maps

```
<map>
  <entry key="myKey1" value="myValue"/>
  <entry key-ref="account" value="myValue"/>
  <entry value="myValue">
    <key>
      <value type="java.lang.Integer">1</value>
    <key/>
  </entry>
  <entry key="myKey2" value-ref="account">
  <entry key="myKey3">
     <value type="java.lang.Long">345</value>
  </entry>
</map>
```

Here you see various examples of usage of the ‘map’ element, which represents java.util.map

The map entries can be specified using a variety of attributes, elements, or both. Types can be indicated with the type attribute on value elements.

Blueprint details (23 of 26)

- Defining values - Collections
  - Collections
    - Properties

```
<props>
  <prop key=”pie”>good</prop>
  <prop key=”salad” value=”bad”/>
</props>
```

Finally, Blueprint provides a props element, which maps to a java.util.Properties class, where the keys and values are of type string.

The value can be specified using content of the element, or as an attribute.

Blueprint details (24 of 26)

- Defining values – Type conversion
  - Type conversion
    - Blueprint type converters

During injection, the Blueprint Container converts the XML value elements into actual objects. It converts the elements based on the type of the injected property.

The Blueprint Container provides several built-in conversions, such as converting: string values into all primitive types, wrapper types, or any types that have a public constructor that takes a String value; array elements into collection objects with compatible member types; or list or set elements into array objects with compatible member types.

In addition to the built-in conversions, a Blueprint bundle can provide its own converters. The custom converters are bean managers that provide an object that implements Blueprint's Converter interface. The custom converters are specified within the type-converters element under the Blueprint element. The type converters are initialized first during Blueprint Container initialization so other managers can take advantage of the custom converters. The Blueprint Container specification has more information about writing custom converters.

Blueprint details (25 of 26)

- Environmental managers
  - blueprintBundle
  - blueprintBundleContext
  - blueprintContainer
  - blueprintConverter

The Blueprint Container specification also defines several special environmental managers that have set IDs and provide access to the environmental components. They have no XML definition and cannot be overridden, since their IDs are reserved and cannot be used with other managers. The objects provided by environment managers can only be injected into other managers using a reference.

The Blueprint Container specification defines four environment managers:

The **blueprintBundle** manager provides bundle's Bundle object.

The **blueprintBundleContext** manager provides bundle's BundleContext object.

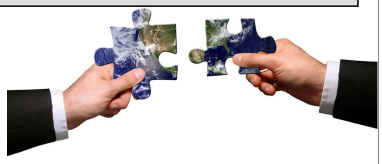
The **blueprintContainer** manager provides the BlueprintContainer object for the bundle.

The **blueprintConverter** manager provides the Converter object for the bundle that provides access to the Blueprint Container type conversion facility.

Blueprint details (26 of 26)

- XML definitions
  - Namespaces
  - Global configuration attributes
  - XML locations.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
default-activation="eager" default-timeout="40000"
default-availability="optional"
> ...
</blueprint>
```



Blueprint xml is schema validated, as such, its elements MUST come from the correct namespace.. ”http://www.osgi.org/xmlns/blueprint/v1.0.0”

The root Blueprint element supports three global configuration attributes which can set the default timeout for service reference proxies, activation for beans, and availability for service references.

Blueprint XML is looked for inside the bundle at the location or locations specified within manifest header directive Bundle-Blueprint

If this header if absent, xml is checked for inside the default location of /OSGI-INF/blueprint within the bundle.

Section

Troubleshooting

This section will provide some troubleshooting hints.

Troubleshooting Blueprint (1 of 2)

- Logging
  - Enabling trace for Apache Aries Blueprint in WebSphere Application Server
  - How to find the trace
- Common problems
  - Lack of an XML namespace
  - Incorrect XML file location
  - Badly formed / invalid XML
  - Use of additional unsupported namespaces
  - Unsatisfied dependencies

As Blueprint is consumed from Apache Aries, its trace package is org.apache.aries.blueprint.*

Associated trace of Aries=all,* (and org.apache.aries.*) will be of use.

Aries Blueprint trace is redirected from its original trace or logger of SLF4J by way of the Java Logger into WebSphere Application Server's trace. Blueprint trace, once enabled, will appear within the normal trace log output.

Finally, here are a few of the more common issues seen during development.

For the "Lack of an XML Namespace" problem, a lot of sample XML (including the parts in this document), are written without a namespace declaration, to keep them shorter and cleaner. As Blueprint is schema validated, the namespace declaration MUST be present, and MUST be correct, otherwise the XML is not identified as Blueprint XML, and will not be processed.

For the "Incorrect XML File Location" problem, Blueprint XML will only be processed from locations in the Bundle-Blueprint header, or from within the default /OSGI-INF/blueprint if you are used to JEE, its possible to confuse the OSGI-INF with META-INF which will lead to the XML not being processed.

For the "Badly Formed or Invalid XML" problem, as the XML is schema validated, if the XML cannot be passed, or is in disagreement with the Schema, the XML will fail to be processed.

For the "Use of Additional Unsupported Namespaces" problem, many Blueprint elements allow elements from namespaces other than Blueprint to be present (for extensibility). However Blueprint requires that any Blueprint Container (thing responsible for reading and understanding the XML) should understand ALL namespaces used. Aries Blueprint provides Namespace Handlers as extensions to the spec to support additional Namespaces, their usage is considered advanced, and unlikely. However if additional Namespaces are found within Blueprint XML, the Blueprint container will enter a GRACE_PERIOD, while it waits to see if a handler for that Namespace becomes available. If none does, the Container will exit, refusing to attempt to interpret XML with content it does not understand.

For the "Unsatisfied Dependencies" problem, if Blueprint XML requires a service that does not exist in the registry, the Container will enter a GRACE_PERIOD while it waits to see if the dependency can be satisfied. Just as with an additional Namespace, if the required service does not appear, the Container will exit.

Troubleshooting Blueprint (2 of 2)

- External references
  - www.osgi.org
    - For the OSGi enterprise and core specification

OSGi.org has the Blueprint specification, which defines the correct behavior for Blueprint in case of query.

Summary

- Overview
- Managers/Components
  - Beans, services, references to services.
- Defining values
  - References, values, collections, type conversion
- Environmental managers
  - Blueprint provided managers
- XML definitions
  - Namespaces, global configuration attributes, locations.
- Troubleshooting

In this session you saw an overview of what Blueprint is, where it came from and what it does.

You took a deeper dive into how it works and what you will need to do to use it.

Finally, you saw some of the problems observed during the development and testing of the feature pack which ships the functionality of this new specification.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send email feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_wasosgijpafep_OSGi_blueprint.ppt

This module is also available in PDF format at: ../wasosgijpafep_OSGi_blueprint.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, disclaimer, and copyright information