RFC 142 - JNDI and OSGi Integration

Draft
v0.120
33 Pages

Abstract

10 point Arial Centered.

This document describes how JNDI can be integrated into the OSGi environment. The usage of JNDI in Java Enterprise Edition application servers is given the most focus. One main consideration should be that Java EE applications that comply with the standard should work as written in an application server running on OSGi. This document presents an approach towards supporting JNDI contexts in an OSGi server. This document will also include possible approaches toward supporting the usage of JNDI that is required in a Java EE server.

Please Note: While the requirements section of this RFC is complete, portions of the design specification are still under discussion. The design section of this document should be considered a "work in progress".

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 8.1.

```
Source code is shown in this typeface.
```

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	09 30 2008	*Initial Draft* *John Wells, Oracle john.wells@oracle.com*
Draft1	10 07 2008	*Initial Draft Content* *Robert W. Nettleton, bob.nettleton@oracle.com*
Draft2	10 13 2008	*Modified draft, incorporated some internal feedback from Jeff Trent* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.2	12 3 2008	*Version 0.2 of draft. Added requirement to support lookups of OSGi services. Incorporated some feedback from Graham Charters, including his suggestion for an "osgi" services URL.* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.3	1 9 2009	*Version 0.3 of draft. Added a proposal for a possible solution of the issues around using URL context factories in OSGi/JNDI* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.4	1 22 2009	*Version 0.4 of draft. Added more details on how the factory manager could locate URL context factory implementations. Additional content added concerning the differences between URL Service Handlers and URL Context Factories.* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.5	2 5 2009	*Version 0.5 of draft. Incorporated feedback from EEG group as a result of the meeting on 1 30 2009.* *Incorporated feedback from Ben Hale from SpringSource on JNDI lookups of OSGi services.* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.6	3 3 2009	*Version 0.6 of draft. Incorporated feedback from EEG meeting on 2 27 2009.* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.7	3 5 2009	*Version 0.7 of draft. Early Release Draft* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.8	4 22 2009	*Version 0.8 of draft. Incorporated feedback from EEG as a result of the "Face to Face" meeting in Austin on 3 11 2009.* *Robert W. Nettleton, bob.nettleton@oracle.com*

Revision	Date	Comments
Version 0.9	7 6 2009	*Version 0.9 of draft.* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.10	7 16 2009	*Version 0.10 of draft* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.11	7 29 2009	*Version 0.11 of draft. This version fixes a typo in the "osgi" URL definition section.* *Robert W. Nettleton, bob.nettleton@oracle.com*
Version 0.12	8 20 2009	*Version 0.12 of draft. Fixes some issues found during testing: adds an implementation note on "osgi" URL support, and also adds content defining the support for jndi.properties files.* *Robert W. Nettleton, bob.nettleton@oracle.com*

1 Introduction

JNDI (the "Java Naming and Directory Interface") is a popular registry technology in the enterprise space. OSGi has its own object registry model. This paper shall attempt to define the points whereby the JNDI registry and the OSGi registry can be integrated.

This document will suggest an approach towards integrating JNDI into an OSGi-enabled application. Since JNDI can be considered a technology on it's own, a portion of this solution will allow for a pure JNDI provider to integrate into an OSGi-based server.

In addition to the JNDI itself, the Java EE ("Java Enterprise Edition") specification requires JNDI contexts to be handled in a special way, depending upon the configuration of a deployed application. In particular, application namespaces are expected to be treated separately in Java EE applications. The "environment naming context" (also known as "java:comp/env") is also required to be supported in a Java EE server. An application server is required to provide unique environment naming contexts for each component in an application. An example of a component in this case would be a servlet or an EJB.

2 Application Domain

Enterprise code using Java EE depends heavily on the use of JNDI to register and find objects. The specification details elements of JNDI that must be supported in order to properly deploy Java EE applications.

3 Problem Description

3.1 Basic JNDI Support

Java EE Applications deployed into an OSGi-based server need to be able to access a naming service in order to locate services. In addition, there may be other applications that don't conform to the Java EE specification, but may require a naming service to properly locate services. The OSGi framework needs to provide an extensible mechanism for publishing and finding both JNDI `InitialContextFactory` and `Context` instances.

3.1.1 Use Case 1: Default Context Creation

JNDI clients may choose to rely on a container environment or configuration to determine the JNDI custom implementation to use when using a naming service. This environment can be provided by a container, such as an application server, or can also be configured as a set of system properties in the JVM.

The following is an example of creating a JNDI InitialContext with the default configuration:

```
InitialContext context = new InitialContext();

DataSource dataSource = (DataSource)context.lookup("accountingDataSourceName");
```

3.1.2 Use Case 2: User-specified JNDI implementation

Another common use case with JNDI Clients involves the client specifying the custom InitialContextFactory to use in order to create the InitialContext. The following code snippet demonstrates this use case:

```
Hashtable environment = new Hashtable();

environment.put(Context.INITIAL_CONTEXT_FACTORY,
"com.demo.DemoInitialContextFactory");
```

```
InitialContext context = new InitialContext(environment);
```

In this example, a client configures the JNDI framework to locate and create an instance of the "DemoInitialContextFactory". The JNDI framework is then expected to create a new `javax.naming.Context` instance to return to the client.

Both of these use cases must be supported for JNDI clients running within an OSGi container.

3.1.3 Use Case 3: URL Context Factory Support

The JNDI framework defines a mechanism by which a JNDI service provider can resolve lookups for custom Uniform Resource Locator (URL) forms.

The following code example demonstrates the usage of a URL in a JNDI lookup:

```
InitialContext initialContext = new InitialContext();

// retrieve an object using a URL based on the "java" scheme

HelloBean hello = (HelloBean)initialContext.lookup("java:comp/env/Hello");

// another example, retrieve an object using the custom "util" scheme

ThreadPoolManager manager =

    (ThreadPoolManager)initialContext.lookup("util://ThreadPoolManager");
```

These code examples demonstrate the most common usage pattern for URL based lookups. Many of these types of lookups will be required by customers regardless of the `InitialContextFactory` used.

In a pure JNDI (non-OSGi) setting, service providers can make an implementation of `javax.naming.spi.ObjectFactory` available to create a JNDI context using a URL as an input. This `ObjectFactory` implementation is known as a "URL Context Factory", since the factory creates a context given a URL string. The details of how a service provider makes this factory available can be found at the following link:

http://java.sun.com/products/jndi/tutorial/provider/url/index.html

A URL Context Factory allows a service provider to extend the default behavior of the JNDI runtime without having to provide an `InitialContextFactory`. Once available to the JNDI runtime, a URL Context factory can be used with any initial context implementation, since the `javax.naming.InitialContext` class provides the support for locating factories based on a given URL scheme.

In a pure JNDI setting, the service provider must follow a well-defined naming convention for the URL context factory, and then make this factory's package available via a system property used by the JNDI framework.

Support for JNDI applications in OSGi will require the handling of URL context factories. The default JNDI support for this type of factory will be problematic in OSGi, given the reliance on system properties. This issue is discussed in greater detail in another section of this document.

3.2 Classloading

JNDI was designed and added to the Java platform before OSGi was prevalent, and it does not deal well with the OSGi class loading architecture. In particular, the JNDI code running inside the Java VM by default expects that key JNDI classes, such as context factories and object factories, are loaded using the system class loader. There is no provision in the JNDI specification to plug in a different classloading system such as OSGi.

In most cases, JNDI supports APIs that allow a user to change the way that classes are loaded, so that it is possible to take advantage of the OSGi environment. However, for various detailed technical reasons in the design and implementation of the JNDI APIs, there are reasons why these APIs do not solve the entire problem. In particular, once builder classes are installed with the `NamingManager`, there is no mechanism to un-set or re-set the builder classes. This causes the JVM to hold onto a reference to any installed builder, as well as the classes that represent that builder. This issue is discussed in greater detail in another section of this document.

3.3 OSGi Service Integration

3.3.1 OSGi Service Access

In addition to the Java EE support described above, it may be desirable to allow JNDI clients to access OSGi services via the JNDI context. This would allow clients to migrate to an OSGi services-based model with minimal code changes. The client would use the Context.lookup() method to locate a service, and then interact with that service in the expected way.

In order to support this feature, the OSGi framework should provide a JNDI Context implementation that can resolve lookup requests for the “osgi” namespace.

This access is defined to be read-only, in that only OSGi service access through the JNDI context is permitted.

The following is an example of this type of lookup:

```
InitialContext context = new InitialContext();


ExampleService exampleService =

        (ExampleService)context.lookup(“osgi:services/com.example.ExampleService”);
```

This example utilizes an “osgi”-based URL form that JNDI clients can pass to the lookup() method of the JNDI Context.

3.3.2 Handling of OSGi ServiceReferences

When OSGi clients access services, either by using the BundleContext API or the ServiceTracker, the usage pattern typically involves the following operations:

1. Obtain reference to service (using `BundleContext.getServiceReference()`)

2. Using the ServiceReference obtained in Step 1, obtain the service (using BundleContext.getService(serviceReference))

3. Invoke operations on service

4. Release reference to service (using `BundleContext.ungetService()`)

The javax.naming.Context interface does not support such a model directly. In typical JNDI scenarios, clients access an object using a JNDI name, and then interact with the object. There is no support in the JNDI programming model for informing the provider that a given object is no longer needed by the client.

A JNDI provider that handles requests for OSGi services will need to implement some approach for handling the lifecycles of the OSGi services made available via JNDI.

3.3.3 BundleContext Handling

A JNDI provider that handles lookup requests for OSGi services will need to make service requests on behalf of the bundle that the JNDI client executes in. This generally means that the client's BundleContext instance must be used to invoke the getService() call. The client's bundle context must be used in this scenario for the following reasons:

1. **Classloading** - The JNDI provider needs to insure that the proper classloader is always used to load the classes for a given service. For example: if bundle A attempts a lookup of service "my.test.Service" that happens to be deployed in bundle B, there is always a chance that bundle A will have an "Import-Package" statement that pulls in the "my.test" package from another bundle, C in this case. In this example, when bundle A attempts to cast the object returned from the lookup to the service type would fail, since an object of the interface type loaded from bundle B is returned, but the cast expects the interface type to be loaded by bundle C's loader. The OSGi Specification requires that BundleContext implementations of getServiceReference and getServiceReferences filter out any services with “incompatible” types. When a caller uses the BundleContext API directly, the ClassCastException will not occur, since an incompatible service won't be returned. The JNDI provider in this case must have access to the client's BundleContext.

2. **Service Tracking** – OSGi services returned to JNDI clients must be tracked appropriately in order to allow the OSGi Framework to properly manage service dependencies. Using the JNDI client's BundleContext to acquire and release services is the only way to achieve this.

4 Requirements

1. A JNDI application (defined as an OSGi bundle using JNDI, or a Java EE application that is deployed on an OSGi-enabled container) MUST be able to use the InitialContext class to create a Context object as long as the InitialContextFactory class is exported from an OSGi bundle, or resides on the system class path.

2. When a *Referenceable* JNDI object is retrieved from JNDI, code running in any OSGi bundle MUST be able to successfully retrieve the object, as long as the object's *ObjectFactory* or *DirObjectFactory* is exported by another bundle, or resides on the system class path.
3. There MUST be a way to export an *ObjectFactory* from an OSGi bundle so that it may be used as a "URL Context" (also known as a "URL Context Factory") to allow JNDI access via URL.
4. There MAY be additional support for URL context factories to simplify the task of creating the URL context factory.
5. There SHOULD be a way to add an *ObjectFactory, DirObjectFactory, StateFactory,* or *DirStateFactory* to the list of factories to be searched by default by JNDI by registering a service with the OSGi service registry, in addition to the methods already supported by JNDI.
6. Any OSGi bundle MUST be able to look up and fully utilize a JNDI *Context* object bound in to the Service Registry.
7. There MUST be a standard that describes the mechanism that JNDI will use to search for environment properties when running in an OSGi environment. The standard must describe how JNDI environment properties, especially those stored in resource files, are loaded, and how that differs from a non-OSGi environment.
8. There MUST be a way to access OSGi services via a JNDI context. This feature will require a way to specify the JNDI name that a service may be published under, as well as support for managing the lifecycle of the OSGi service.
9. Once a client accesses an OSGi service via JNDI, the OSGi/JNDI integration solution MUST monitor the usage of the service reference, in order to properly maintain the lifecycle of the service.

5 Technical Solution

5.1 Overview

In general, the components in an OSGi-enabled application server will need to locate and publish Java EE-related services via JNDI. This means that bundles will need to be able to locate JNDI contexts in a standard way, and it

will also be desirable to have bundles publish their `javax.naming.Context` implementations for other bundles to consume.

An important goal of this integration will be to support Java EE applications that are running in an OSGi-based container. These Java EE applications should not have to be modified in order to run in a Java EE server that just happens to be using OSGi infrastructure. This presents a challenge to application server vendors, since typical application servers handle issues such as classloading and service location in a different fashion than OSGi.

Note: While the concept of an application is essential to Java EE development, this specification does not require that an OSGi-enabled server implement that concept by using a one-to-one mapping of applications to bundles. This partitioning may be accomplished in any number of ways. The most significant requirement in this area revolves around the notion that a Java EE application must be kept separate from other deployed EE applications.

5.1.1 InitialContextFactory Creation

Section 3.3.1 lists a small code snippet, that is repeated here:

```
InitialContext context = new InitialContext();
```

In many cases, JNDI clients can create a JNDI context using the default constructor listed above. This requires the application server to provide a context implementation that can represent the current Java EE application. The JNDI framework relies on environment properties (either passed in at construction time, or set by a component container) to determine which `javax.naming.InitialContextFactory` to use to construct the context.

Another common use case with JNDI Clients involves the client specifying the custom InitialContextFactory to use in order to create the InitialContext. The following code snippet demonstrates this use case:

```
Hashtable environment = new Hashtable();

environment.put(Context.INITIAL_CONTEXT_FACTORY,
"com.demo.DemoInitialContextFactory");

InitialContext context = new InitialContext(environment);
```

In this example, a client configures the JNDI framework to locate and create an instance of the "DemoInitialContextFactory". The JNDI framework is then expected to create a new `javax.naming.Context` instance to return to the client.

The JNDI framework also provides an additional integration point that allows a component to control how JNDI contexts are created. A component can provide an implementation of:

```
javax.naming.spi.InitialContextFactoryBuilder,
```

which is a builder for `InitialContextFactory` instances. A component can provide a builder implementation to the JNDI framework by calling the following API method:

```
javax.naming.spi.NamingManager.setInitialContextFactoryBuilder(InitialContextFactor
yBuilder)
```

This method can only be called once per Virtual Machine run, and cannot be overwritten once set. The component must also have the proper security permissions for this call to succeed. Once this method has been called, the JNDI framework will delegate any context creation requests to this builder implementation. This allows

a JNDI provider to have more fine-grained control over the creation of contexts, and provides a single entry point into the naming subsystem. The sequence diagram in Figure 5-1 gives an overview of the interaction involved during an attempt to create a new InitialContext. Except for "Client", all classes listed in this diagram are part of the standard JNDI framework, and exist in the "javax.naming.*" packages.

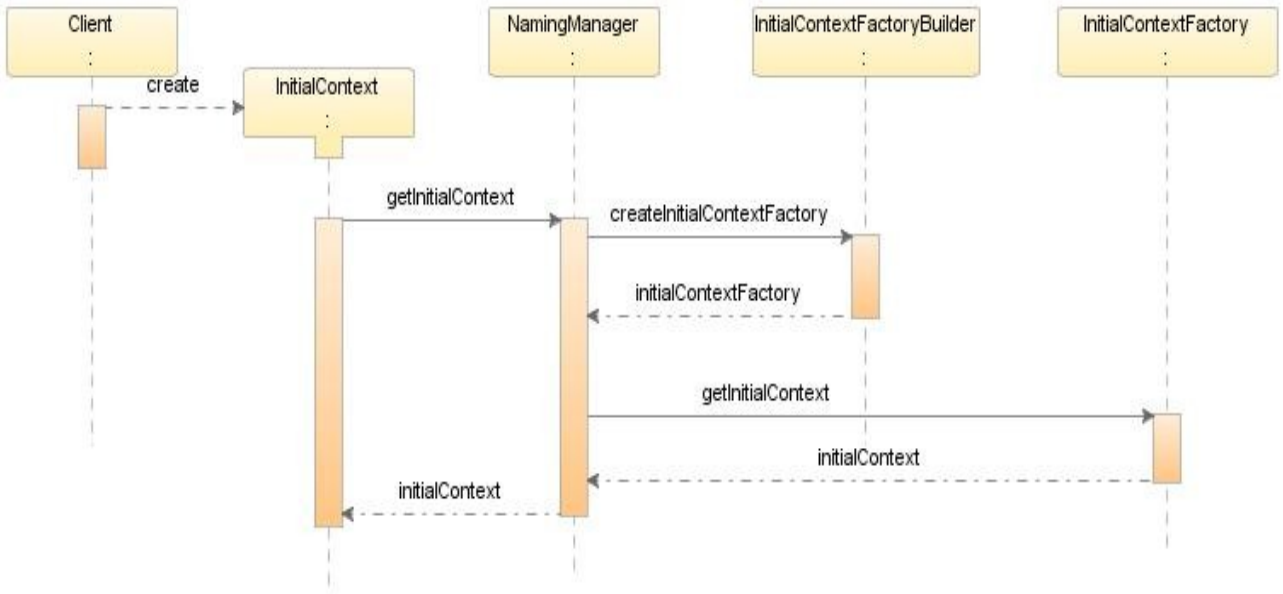

Figure 5-1 Sequence Diagram of JNDI Context Creation with Builder installed

5.1.2 ObjectFactory Creation

The JNDI provides a framework for customizing the binding that is returned to a caller. The most common example would be a call to `javax.naming.Context.lookup(String)`. For example: a JNDI client creates a `Context` instance, and attempts to lookup a reference to a `Printer` object. The following code snippet demonstrates this type of lookup:

```
InitialContext initialContext = new InitialContext();

Printer printer = (Printer)intialContext.lookup("test_printer");
```

In the simplest case, the JNDI service provider may bind the `Printer` object directly to the name "test_printer". There may, however, be a case for binding a `javax.naming.Reference` object instead. Binding a Reference object allows for a level of indirection when this name is resolved. At lookup time, the JNDI can use the Reference to create a Printer object to return to the caller. This could allow for more flexible object creation policies, such as a "lazy" object creation scheme for objects that require large amounts of resources.

This object creation mechanism is provided by the `javax.naming.spi.ObjectFactory` interface. JNDI service providers that need to allow for this type of reference resolution implement the `ObjectFactory` interface. While resolving a lookup() request, a JNDI context is required to test the object's type to determine if the object is a `javax.naming.Reference`. If this test passes, the context implementation is required to call `javax.naming.spi.NamingManager.getObjectInstance(Object,Name,Context,Hashtable)` in

order to resolve the reference. The `NamingManager` will then go through a series of attempts to find a factory to build the object from the `Reference`. The steps for this resolution are defined in the `NamingManager`'s javadoc, and in many cases depend upon the `Context.OBJECT_FACTORIES` system property. The order of object factories listed in this system property can determine which factory is consulted first in an attempt to build the object.

The JNDI's default mechanism for finding ObjectFactory instances is largely dependent upon the value of the system property mentioned above. For this reason, any JNDI integration with OSGi will need to take a different approach in order to support resolving `javax.naming.Reference` instances.

The JNDI provides an integration point with respect to finding ObjectFactory instances that is very similar to the InitialContextFactoryBuilder approach listed in Section 5.1.1. A component can provide an implementation of:

`javax.naming.spi.ObjectFactoryBuilder,`

which is a builder for `ObjectFactory` instances. A component can provide a builder implementation to the JNDI framework by calling the following API method:

`javax.naming.spi.NamingManager.setObjectFactoryBuilder(ObjectFactoryBuilder)`

This method can only be called once per Virtual Machine run, and cannot be overwritten once set. The component must also have the proper security permissions for this call to succeed. Once this method has been called, the JNDI framework will delegate any attempts to resolve a `javax.naming.Reference` with an `javax.naming.spi.ObjectFactory` to this builder implementation. This allows a JNDI provider to have more fine-grained control over the creation of object factories, and also more control over the manner in which references are resolved into objects. The sequence diagram in Figure 5-2 gives an overview of the interaction involved during an attempt to lookup a name that is bound to a `Reference`. Except for "Client", all classes listed in this diagram are part of the standard JNDI framework, and exist in the "`javax.naming.*`" packages.

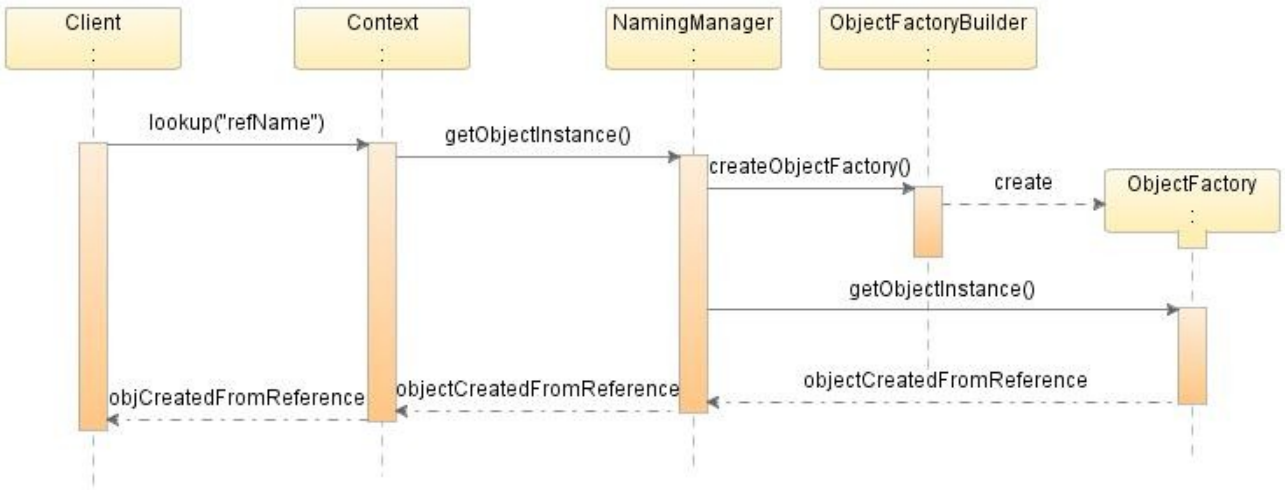

Figure 5-2 Sequence Diagram of Object resolution using ObjectFactoryBuilder

5.2 Factory Manager bundle

This document proposes using a special bundle, known as the “factory-manager” bundle, to handle the integration with JNDI and OSGi. This bundle will be considered a default part of the OSGi framework, and must be installed for the integration to succeed. During bundle activation, this “factory-manager” bundle will provide implementations of `InitialContextFactoryBuilder` and `ObjectFactoryBuilder` to the JNDI `NamingManager`. These builder implementations will interact with the OSGi service registry in order to discover new `InitialContextFactory`, `InitialContextFactoryBuilder`, `ObjectFactory`, `ObjectFactoryBuilder`, and `Context` implementations. This will allow the bundle to dynamically provide JNDI access to any applications and bundles that publish such information on the registry. This bundle will provide a default implementation of the `InitialContextFactoryBuilder` and `ObjectFactoryBuilder` in order to allow existing context factories to be placed on the classpath and used “as-is”. For a more flexible approach, a bundle can publish an OSGi service that implements the `InitialContextFactoryBuilder` interface, the `IntialContextFactory` interface, or both to handle context factory creation. A bundle will also be able to publish an OSGi service that implements the `ObjectFactory` interface or the `ObjectFactoryBuilder` interface, in order to support more flexible reference resolution. The “factory-manager” bundle will interact with the OSGi framework to detect the publishing of services that implement these interfaces, as well as the removal of these services from the OSGi registry. The method of tracking services is considered an implementation detail, and implementations of this specification are free to use any method that is most appropriate (ServiceListener, Service Tracker, etc).

5.2.1 InitialContextFactoryBuilder Integration

The following diagram details how the “factory-manager” bundle can integrate with the JNDI framework to customize the creation of `InitialContextFactory` instances, which ultimately allow an implementer to customize the creation of `javax.naming.Context` instances.

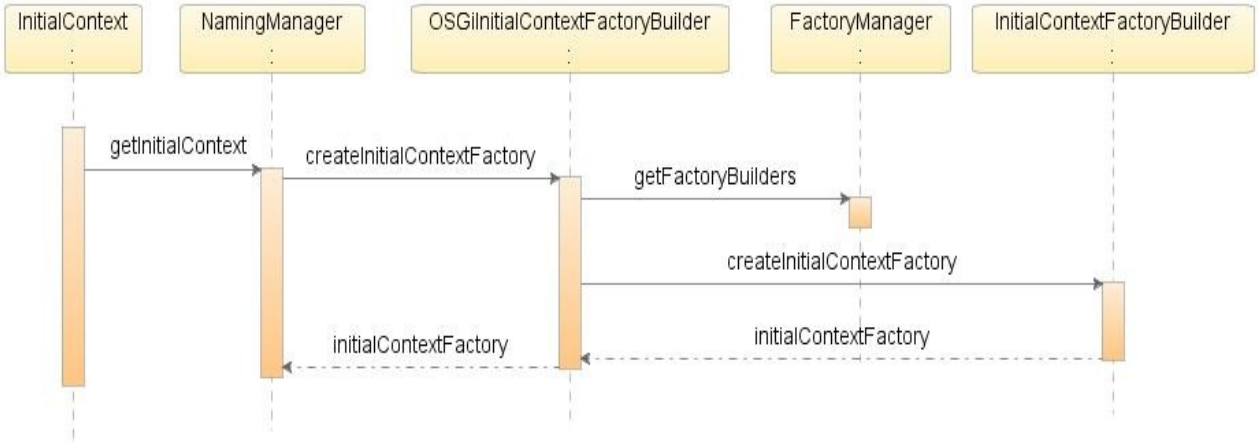

Figure 5-3 Sequence Diagram of OSGi InitialContextFactoryBuilder installed

The diagram above shows one possible way to integrate with JNDI using the standard NamingManager. Some differences with the first diagram (5-1) are worth mentioning:

1. **OSGiInitialContextFactoryBuilder** – This class is an implementation of `javax.naming.spi.InitialContextFactoryBuilder`. Upon activation of the "factory-manager" bundle, this builder is created and registered with the `NamingManager`. After this registration, all attempts to create a JNDI `InitialContext` will be routed through this builder. The builder is free to create new factory implementations, delegate to existing factories, or build proxy implementations as needed. This class can be considered the main entry point between the JNDI framework and the OSGi implementation of JNDI. The proposed implementation will delegate the work of managing these OSGi services to the FactoryManager.
2. **FactoryManager –** This class interacts with OSGi in order to detect any OSGi services that are published that conform to the InitialContextFactoryBuilder interface. This class can utilize OSGi Framework APIs to detect the published services available, as well as track the registration/de-registration of services during runtime. The method used by this component to track services is not specified, and is left as an implementation detail. The presence of this particular class is not necessarily required to implement this bundle. This class is in place to represent the code that will integrate with OSGi in order to detect available JNDI providers. Figure 5-5 below describes how the FactoryManager can register itself with the OSGi service registry in order to locate InitialContextFactoryBuilder and ObjectFactory services.
3. **InitialContextFactoryBuilder –** In this diagram, this class can represent any implementation of the factory builder interface that is available via OSGi services. This implementation can be provided by the "factory manager" bundle itself, or can be one of an arbitrary number of services published in the OSGi service registry.

5.2.1.1 Behavior of FactoryManager with Context Factories

The FactoryManager class is used in the above diagrams to represent the internal implementation of the factory manager bundle. While the class itself is not required by implementors of this specification, there are some requirements on how this bundle locates and returns JNDI services.

The JNDI framework allows customization at several levels. A service provider typically implements `javax.naming.spi.InitialContextFactory` in order to provide a custom JNDI context factory. While this is the common case, it is also possible that a service provider may wish to provide an `InitialContextFactoryBuilder`, in order to support more fine-grained control over the creation of context factories.

This document proposes that the "factory manager" bundle support both interfaces using the following mechanism:

When the `InitialContextFactoryBuilder` instance installed by the "factory manager" bundle is invoked by the framework with a request to provide an `InitialContextFactory` instance, it should use the following search order:

1. If a specific `InitialContextFactory` class is specified by the client, the "factory manager" must attempt to locate a service that supports an interface defined by the classname of the custom factory. If this service is found, it should be returned to the client. If a service is not found to be registered under this factory class name, the "factory manager" should iterate over the list of known implementations of the `javax.naming.spi.InitialContextFactoryBuilder` interface, and attempt to use each to create the `InitialContextFactory`. If one of the `InitialContextFactoryBuilder` implementations returns a non-null result, this result should be returned to the client. The "factory manager" should return the first non-null result found.

2. If a specific `InitialContextFactory` implementation is not requested, the "factory manager" should iterate over the list of known implementations of the `javax.naming.spi.InitialContextFactoryBuilder` interface, and attempt to use each to create the `InitialContextFactory`. If one of the `InitialContextFactoryBuilder` implementations returns a non-null result, this result should be returned to the client. The "factory manager" should return the first non-null result found. The case of a specific factory not being requested is most common in the Java EE use cases, where a client typically uses the no-arguments constructor for javax.naming.InitialContext, and the JNDI environment is assumed to be configured by an external container.

3. If an `InitialContextFactory` is not found after iterating over the list of known `InitialContextFactoryBuilder` services, the "factory manager" should try to return a "default" context factory by examining the list of known OSGi services published under the `javax.naming.spi.InitialContextFactory` interface, and return the first service found that implements this interface. The selection of the context factory to be returned should follow the service ranking rules of the OSGi Service Layer.

4. If an `InitialContextFactory` instance is not available, the "factory manager" should make an attempt to load the class specified by the client's JNDI properties, and to create an instance of this class to return to the JNDI framework. Any factory implementations that are provided by the JDK should be available via this method.

5. If no implementations exist that can support the given JNDI environment, this should be treated as an error condition by the "factory manager" bundle, and a `javax.naming.NoInitialContextException` should be thrown back to the caller.

The process by which the "factory manager" bundle locates `InitialContextFactory/InitialContextFactoryBuilder` instances has the following implications for JNDI service providers:

1. A service provider wishing to provide a JNDI service that will be specifically requested by the client (using the standard JNDI environment properties) must register an OSGi service under the `javax.naming.spi.InitialContextFactory` interface. This service must also be published under the classname of the factory itself. One exception to this rule would be the set of JNDI factory implementations that are provided by the JDK itself. The "factory manager" should make these available to a client without having to register these factories as OSGi services.

2. A service provider wishing to have a more flexible approach to creating JNDI factory instances must register an OSGi service under the `javax.naming.spi.InitialContextFactoryBuilder` interface. This service must examine the parameters passed into the `InitialContextFactoryBuilder.createInitialContextFactory()` method, to determine if this service can satisfy the given request. If this builder cannot satisfy the given request, the builder method must return null.

3. If any `InitialContextFactory` instances are registered as OSGi services, there is a possibility that one of these instances will be used as a default `InitialContextFactory` as described in the section above.

5.2.2 ObjectFactory integration

The following diagram details how the “factory-manager” bundle can integrate with the JNDI framework to customize the resolution of javax.naming.Reference instances.

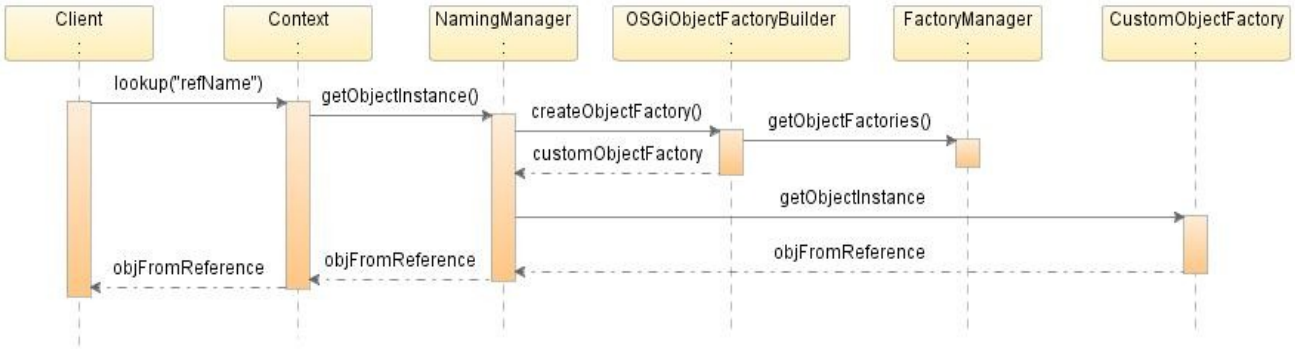

Figure 5-4 Sequence Diagram of Reference Resolution with OSGiObjectFactoryBuilder installed

This diagram shows that the basic integration points with the JNDI framework are quite similar for `InitialContextFactoryBuilder` and `ObjectFactoryBuilder` instances. The “factory-manager” bundle registers an implementation of each interface (at most once) with the `NamingManager`, and these instances are used by the `NamingManager` at various points to resolve objects. In the case of the `InitialContextFactoryBuilder`, this delegation occurs at context creation time. In the case of the `ObjectFactoryBuilder`, this generally occurs when a client attempts a lookup for a name that is bound to a `javax.naming.Reference`.

The following provides more detail on some of the classes listed in Figure 5-4:

1. **OSGiObjectFactoryBuilder –** an implementation of `javax.naming.spi.ObjectFactoryBuilder` that the “factory-manager” bundle registers with the `NamingManager`. JNDI Context implementations are expected to attempt to resolve Reference objects using `NamingManager.getObjectInstance()`. The proposed implementation will delegate the task of managing references to ObjectFactory to the FactoryManager.
2. **FactoryManager** - This class interacts with OSGi in order to detect any OSGi services that are published that conform to the `ObjectFactory` interface. This class can utilize the OSGi APIs to detect the published services available, as well as track the registration/de-registration of services during runtime. The presence of this particular class is not necessarily required to implement this bundle. This class is in place to represent the code that will integrate with OSGi in order to detect available JNDI providers. Figure 5-5 below describes how the FactoryManager can register itself with the OSGi service registry in order to locate `InitialContextFactoryBuilder` and `ObjectFactory` services.
3. **CustomObjectFactory –** an implementation of `javax.naming.ObjectFactory` that can be provided by either the “factory-manager” bundle or any bundle deployed in the OSGi framework.

5.2.2.1 Behavior of FactoryManager with Object Factories

The FactoryManager class is used in the above diagrams to represent the internal implementation of the factory manager bundle. While the class itself is not required by implementors of this specification, there are some requirements on how this bundle locates and returns JNDI services.

The JNDI framework allows customization at several levels. A service provider typically implements `javax.naming.spi.ObjectFactory` in order to provide a custom factory for resolving `javax.naming.Reference` objects. . While this is the common case, it is also possible that a service provider may wish to provide an `ObjectFactoryBuilder`, in order to support more fine-grained control over the creation of object factories. In a non-OSGi environment, providers would likely avoid implementing the builder interface, due to the limitation of only having one `ObjectFactoryBuilder` registered per VM instance. In the integration with OSGi proposed by this paper, JNDI service providers have the option of registering an `ObjectFactoryBuilder` that can be queried in order to create an object from a `Reference` that may not be directly associated with an `ObjectFactory` classname.

This document proposes that the "factory manager" bundle support both interfaces using the following mechanism:

1. If a `javax.naming.Reference` passed into the `ObjectFactoryBuilder` registered by the "factory manager" bundle requires a given factory, the "factory manager" must attempt to locate a service that supports an interface defined by the classname of the custom factory. The "factory manager" implementation can determine this factory classname using the `Reference.getFactoryClassName()`. This service must also support the `javax.naming.spi.ObjectFactory` interface. If this service is found, it should be returned to the JNDI framework.
2. If the `ObjectFactory` class specified in step 1 was not found in the OSGi service registry, the "factory manager" should iterate over the list of known implementations of the `javax.naming.spi.ObjectFactoryBuilder` interface, and attempt to use each to create the `ObjectFactory`. If one of the `ObjectFactoryBuilder` implementations returns a non-null result, this result should be returned to the JNDI framework. The "factory manager" should return the first non-null result found.
3. If a `javax.naming.Reference` passed into the `ObjectFactoryBuilder` registered by the "factory manager" bundle does not include a specific ObjectFactory classname to use for reference resolution, the "factory manager" should first query the Reference to see if an address exists on the Reference of type "URL". If such an address exists, the "factory manager" should attempt to use this URL address to resolve this reference using a URL Context Factory. The "factory manager" should attempt to locate the URL context factory first in the OSGi Service Registry. If no matching service is found, the "factory manager" should attempt to use the JDK default mechanism for attempting to locate the URL Context Factory (the "java.naming.factory.url.pkgs" property).
4. If the Reference in step 3 does not include an address of type "URL", the "factory manager" should iterate over the list of known implementations of the `javax.naming.spi.ObjectFactoryBuilder` interface, and attempt to use each to create the `ObjectFactory`. If one of the `ObjectFactoryBuilder` implementations returns a non-null result, this result should be returned to the JNDI framework. The "factory manager" should return the first non-null result found.
5. If no ObjectFactory implementations can be located to resolve the given Reference, the "factory manager" should use the "java.naming.factory.object" system property in order to obtain the list of object factories installed at the VM level, and attempt to use each of these factories to resolve the reference.
6. After searching using the "java.naming.factory.object" system property, if no ObjectFactory implementations can be located to resolve the given Reference, the "factory manager" should return the

Reference originally passed in to the ObjectFactoryBuilder to the JNDI framework. This allows the ObjectFactory resolution process to follow the approach taken by the JDK as closely as possible.

The steps listed above would also apply for any objects of type `javax.naming.Referenceable` passed into a call to the `ObjectFactoryBuilder`.

Implementation Note:

In order to support the Reference resolution process defined in the javadoc for `javax.naming.spi.NamingManager.getObjectInstance()`, an implementation of RFC 142 will need to defer the search for matching object factories until the JNDI framework invokes `ObjectFactory.getObjectInstance()`. The `ObjectFactoryBuilder` implementation installed by the "factory manager" at startup will be accessed by the JNDI framework during two method invocations: `NamingManager.getObjectInstance()` and `DirectoryManager.getObjectInstance()`. In each of these cases, the manager class first obtains the `ObjectFactory` from the builder, and then returns the result of the `getObjectInstance()` call on this `ObjectFactory` instance. In order to properly support the expected return values defined by `NamingManager` and `DirectoryManager`, the `OSGiObjectFactoryBuilder` should return a wrapper instance of `ObjectFactory` that can attempt the object factory search defined in the steps above in the implementation of the `ObjectFactory.getObjectInstance()` method. If no matching factory can be found, it is the responsibility of this wrapper implementation to return the `Reference/Referenceable` object that was passed into the `NamingManager` or `DirectoryManager`. In addition, implementations using this approach should return a `DirObjectFactory`, in order to support both the Naming and Directory managers. If no actual directory support is provided, the implementation of the `DirObjectFactory.getObjectInstance()` (takes Attributes as a parameter) method should return the Reference/Referenceable object passed into the call to DirectoryManager.getObjectInstance().

The javadoc for `javax.naming.spi.NamingManager.getObjectInstance()` provides more detail on the default search order for resolving a `Reference`:

http://java.sun.com/javase/6/docs/api/javax/naming/spi/NamingManager.html#getObjectInstance(java.lang.Object,%20javax.naming.Name,%20javax.naming.Context,%20java.util.Hashtable)

The process by which the "factory manager" bundle locates `ObjectFactory/ObjectFactoryBuilder` instances has the following implications for JNDI service providers:

1. A service provider wishing to provide an `ObjectFactory` implementation in an OSGi environment must publish an OSGi service that implements the `javax.naming.spi.ObjectFactory` interface, as well as the classname of the custom object factory.

2. A service provider wishing to have a more flexible approach to resolving references must register an OSGi service under the `javax.naming.spi.ObjectFactoryBuilder` interface. This service must examine the parameters passed into the `ObjectFactoryBuilder.createObjectFactory()` method, to determine if this service can satisfy the given request. If this builder cannot satisfy the request, the builder method must return null.

The solution proposed in this document would allow for a seamless integration between JNDI clients and JNDI service providers. JNDI providers could publish their context factories and object factories as OSGi services, and these implementations will be detected by the "factory-manager" bundle. This approach would also have benefits for Java EE application server vendors. A standardized bundle of this type in the OSGi framework could potentially allow for JNDI implementations to be more easily re-used in application servers. This could allow for application servers to pick up new JNDI implementations without changing code.

5.2.3 Bundle activation

The following diagram details how the “factory-manager” bundle's Activator instance initializes the bundle, and uses the FactoryManager to obtain the current list of InitialContextFactoryBuilder and ObjectFactoryBuilder instances, as well as register to be notified when services that implement these interfaces are published or shutdown.

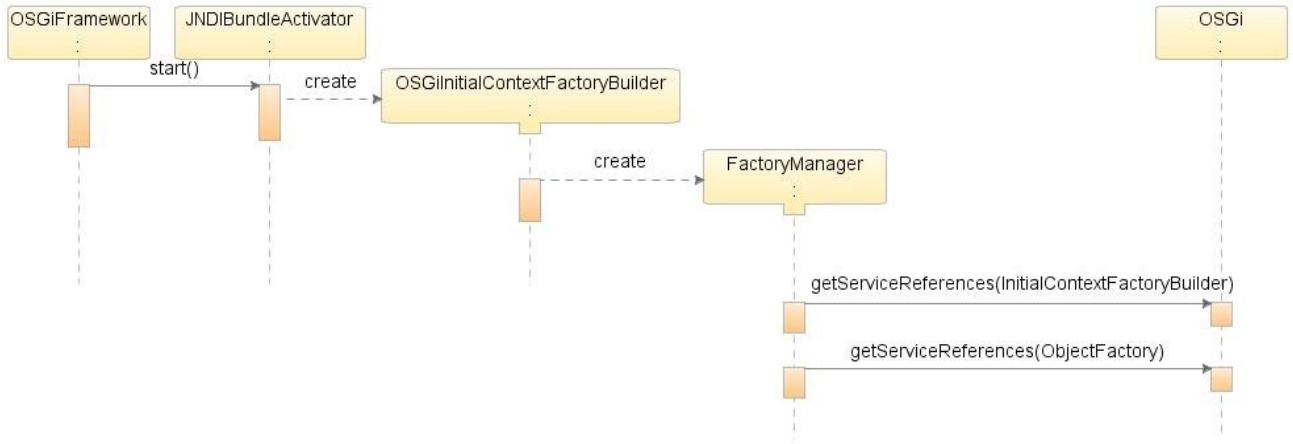

Figure 5-5 Sequence Diagram of Factory Manager startup

This diagram demonstrates the activation process of the “factory manager” bundle. The FactoryManager class listed in this diagram is responsible for obtaining JNDI object factories that are published as OSGi services. The OSGi API used to locate the JNDI object factories is not specified, and can be considered an implementation detail.

5.2.4 JNDI Environment Properties

JNDI clients can use environment properties in order to select a JNDI provider, as well as to pass configuration information to the provider. There are several ways to set environment properties in JNDI, and these properties are processed by the JNDI framework in the following order of precedence:

1. properties passed into the `InitialContext` constructor
2. properties set at the system level
3. properties set in application resource files (jndi.properties)

For an overview of how JNDI properties are processed by the JNDI framework, please see:

http://java.sun.com/products/jndi/tutorial/beyond/env/context.html

Using an application resource file (also known as a jndi.properties file), can allow a client to configure JNDI preferences without setting these preferences in code. This approach can also make a JNDI client more portable, since direct references to a given JNDI provider do not exist in the client's code.

The factory manager outlined in this document is handed the initial set of JNDI environment properties when the JNDI Framework invokes the registered InitialContextFactoryBuilder. This set of environment properties will contain the properties (if any) passed into the InitialContext constructor, as well as any properties set on the System level that the JNDI Framework accepts. The list of accepted properties that can be set at the system level can be found at:

http://java.sun.com/products/jndi/tutorial/beyond/env/context.html

Code using JNDI in non-OSGi scenarios could rely on the JNDI framework to handle the processing of environment properties. In an OSGi container, the factory manager will need to handle a special case that cannot be supported by the JNDI framework. The JNDI Framework attempts to use the System classloader as a mechanism to load jndi.properties files. This approach is not feasible in an OSGi environment, since the System classloader has no visibility into a given Bundle's resources.

The factory manager should provide a limited form of support for jndi.properties files that is more compatible with the classloading in OSGi. The factory manager should support including a jndi.properties at the top level of a given bundle to configure the JNDI client code in that bundle. This set of properties would be used for JNDI client code within the current bundle. This would allow for multiple bundles within a container to have different configuration, and also utilize separate JNDI providers.

5.2.4.1 *Classloader requirements*

Clients must have the current client bundle's classloader set as the Thread context classloader. The factory manager will use the context classloader as a means to access the client's Bundle. Once obtained, the client's Bundle can be used to load a "jndi.properties" file that is located at the top level of the bundle's archive. The context classloader must implement the "org.osgi.framework.BundleReference" interface, described in Section 3.8.9 of the OSGi Core Specification, Version 4.2. In enterprise applications, the Java EE container should fulfill this requirement, since JNDI application code should not require any direct references to the OSGi APIs.

This requirement has the following implications for Java EE Containers, as well as any component container that needs to support this type of JNDI configuration:

Java EE application servers or containers typically initialize much of the environment surrounding a Java EE application component (servlet, EJB, etc). Containers that need to support JNDI environment configuration from properties files must set the client Bundle's classloader to be the current Thread context classloader prior to invoking a method on the application component. This makes the Bundle of the caller accessible to the JNDI factory manager, and allows the factory manager to properly locate a jndi.properties file, if it is included in the bundle's archive.

5.2.4.2 *Precedence ordering of environment properties*

If the factory manager locates a jndi.properties file for a given Bundle, the properties in this file should be merged with the properties that have been passed to the factory manager by the JNDI framework. The factory manager should use the priority order mentioned in section 5.2.4 of this document. One exception to this rule is the handling of jndi.properties files. The factory manager should only process the jndi.properties file from the calling client's Bundle, and not use any properties files that are defined in other bundles. This allows JNDI clients to be configured on a per-bundle basis. The use of System-level properties within an OSGi container for the purpose of configuring a JNDI client should be discouraged.

Certain JNDI properties take multiple values. In these cases, the factory manager must observe the standards for concatenating property values that are set in multiple sources (passed into the InitialContext constructor, set as

System properties, set in a jndi.properties file). The list of standard JNDI properties that can hold multiple values, as well as the rules for concatenating the values, can be found at the following:

http://java.sun.com/products/jndi/tutorial/beyond/env/context.html

5.2.5 OSGi Service handling in FactoryManager

In the diagrams above, the `FactoryManager` class is responsible for maintaining a list of available OSGi services that implement the `InitialContextFactoryBuilder` and `ObjectFactory` interfaces.

The integration points described above allow the "factory-manager" bundle to make the OSGi services that conform to the `InitialContextFactoryBuilder` and `ObjectFactory` interfaces available to any JNDI client code running in the JVM. The dynamic nature of OSGi services opens the possibility that at any time more than one implementation of one of these interfaces may be registered. In the event that multiple services implement the same factory interface, the `FactoryManager` will observe the same rules for service ranking that OSGi service clients use. These rules are detailed in Chapter 5 ("Service Layer") of the OSGi Service Platform specification. In the event of multiple implementations of the same interface, the `FactoryManager` will favor the service with the highest service ranking, then use the lowest service id in the event of a tie. Using this approach will allow JNDI service providers to be used by clients in a predictable fashion, without the possibility of arbitrary factories being returned to the JNDI framework.

5.2.6 URL Context Factory Support

Section 3.1.1 of this document explains the basic requirements for URL context factories, and how these factories can be used to locate objects in JNDI using a URL.

The technical solution detailed in Section 5.2 provides an integration point between the JNDI framework and an OSGi bundle that can handle the task of locating and providing builder classes that can create JNDI contexts. Providing the Builder implementations to the `javax.naming.spi.NamingManager` class is necessary in order to support JNDI running on the OSGi framework, since it allows all context creation requests to be delegated to a single bridge between the JNDI Framework and factory manager bundle.

There is one problem associated with using Builder classes in this way. The support for URL Context Factories in the JNDI framework is provided by the `javax.naming.InitialContext` class. This class handles the task of determining if a URL has been passed to a lookup() request, finding the correct `ObjectFactory` instance to handle the lookup, and then returning the results to the caller. Once an `InitialContextFactoryBuilder` implementation has been registered with the `NamingManager`, all of the default policies of the JNDI framework (including the management of URL context factories) are no longer invoked. At this point, it is the responsibility of the factory manager bundle to ensure that a context implementation returned to the JNDI framework can handle lookup requests for URLs. This support may require wrapping instances of InitialContextFactory as well as the Context interface.

The following diagram details the proposed interaction between the javax.naming.InitialContext instance created by the caller and the factory manager bundle which handles interaction with OSGi in order to obtain JNDI services:

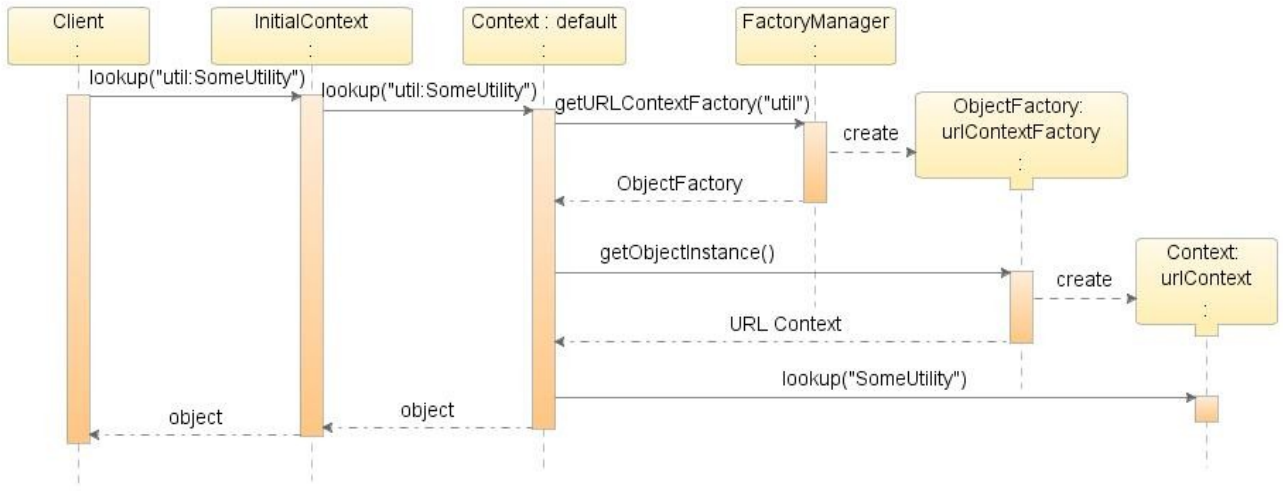

Figure 5-6 Sequence Diagram of URL-based lookup using FactoryManager

The following points are most relevant when considering this diagram:

1. The **Client** class is a JNDI client attempting a lookup using the "util" custom URL scheme. The context being used by the client has already been created and handed back to the client during the call to the `javax.naming.InitialContext` constructor.
2. The **InitialContext** class acts as a wrapper in many cases for JNDI operations. The internals of this class delegate to the actual context created by the `InitialContextFactory`.
3. The **default** `Context` instance queries the `FactoryManager` to find a factory for the given URL scheme
4. The **FactoryManager (described in earlier diagrams)** interacts with the OSGi service registry to find a `javax.naming.spi.ObjectFactory` that can support the requested URL.
5. The **urlContext** instance is used to create a context that supports the "util" scheme, and the lookup request is delegated to that `Context` instance.

5.2.6.1 Locating URL Context Factory implementations

The default mechanism in the JDK used to find a URL context factory involves checking the value of the " java.naming.factory.url.pkgs" system property. This property contains a list of packages that can be used as a search path for URL context factory implementations. In order to be located by the JNDI framework, implementations must be available on the classpath, and must also include a package prefix in the system property listed above.

This document proposes an additional, more straightforward mechanism for publishing and locating URL context factories:

1. A URL Context factory provider registers an implementation as a service under the `javax.naming.spi.ObjectFactory` interface.
2. This service object is registered with a new service property: "osgi.jndi.urlScheme". This property contains a string representing the URL supported by this ObjectFactory.

3. During requests for URL context factory resolution, the `FactoryManager` can query the service registry for an `ObjectFactory` service that supports the URL scheme passed in during the `Context.lookup()` request.

4. If a matching service is found, the corresponding URL context factory is used to resolve the lookup request.

5. If a matching service is not found, the `FactoryManager` will attempt to invoke `NamingManager.getURLContext()` in order to resolve the URL request.

This approach has the benefit of allowing developers to provide URL context factory implementations as OSGi services. Using the service registry to determine whether a given factory supports a URL scheme provides a simpler mechanism for publishing and locating factory instances. This new proposal removes the package-naming restrictions that are present in the default JDK search process for factory instances.

This approach also provides support the default search method for locating factories. This allows existing URL context factory implementations to be supported without additional changes.

While locating URL context factory implementations in OSGi can ease the restrictions on naming and packaging for a factory implementation, it is important to note that compatibility issues may arise if a given provider is expected to work in both environments. If a service provider wishes to create a URL context factory that can work both in the Java SE environment and the OSGi environment, the provider should continue to follow the packaging and naming guidelines documented by the JDK. This will allow a service provider to publish a URL context factory as an OSGi service while running within a framework, but could also potentially be used in a pure Java environment as well.

5.2.6.2 Returning URL Context Factories to the JNDI Framework

The “factory manager” bundle described in this document is responsible for locating URL context factory implementations, and then providing these factories to JNDI clients through the builder interfaces described earlier. The following support for URL context factories must be supported by the “factory manager”:

1. In order to support the type of lookup sequences listed above, the `javax.naming.spi.InitialContextFactory`, as well as any subsequent contexts created by the factory, must be able to intercept the lookup() invocations on the JNDI `Context` by the client.

2. When a client invokes a lookup request using a URL as the lookup string, the JNDI Context that the client interacts with (which is created by the InitialContextFactory that the “factory manager” has returned to the JNDI framework) must detect that a URL has been requested.

3. Once the Context implementation has detected a URL lookup request, this Context is expected to satisfy the lookup request. The process by which the Context implementation interacts with the “factory manager” bundle functionality is not specified, and can be considered an implementation detail.

5.2.6.3 Example of URL Context Factory support

Note: The following section provides details of how an example implementation may be designed to support the requirements for URL Context Factory resolution. This section should not be considered required for implementations of this specification, but is an attempt to describe the most straightforward way of implementing this solution.

In order to support the type of lookup sequences listed above, the `javax.naming.spi.InitialContextFactory`, as well as any subsequent contexts created by the factory, must be wrapped or proxied in order to intercept the lookup() invocations on the JNDI `Context` by the client. Building proxy classes for both the Factory and the Context instances is required in this implementation, due to the interaction between the JNDI framework and the `OSGiInitialContextFactoryBuilder` proposed in section 5.2.1.

The following diagram provides detail on the context creation process that will be necessary in order to support URL-based lookups:

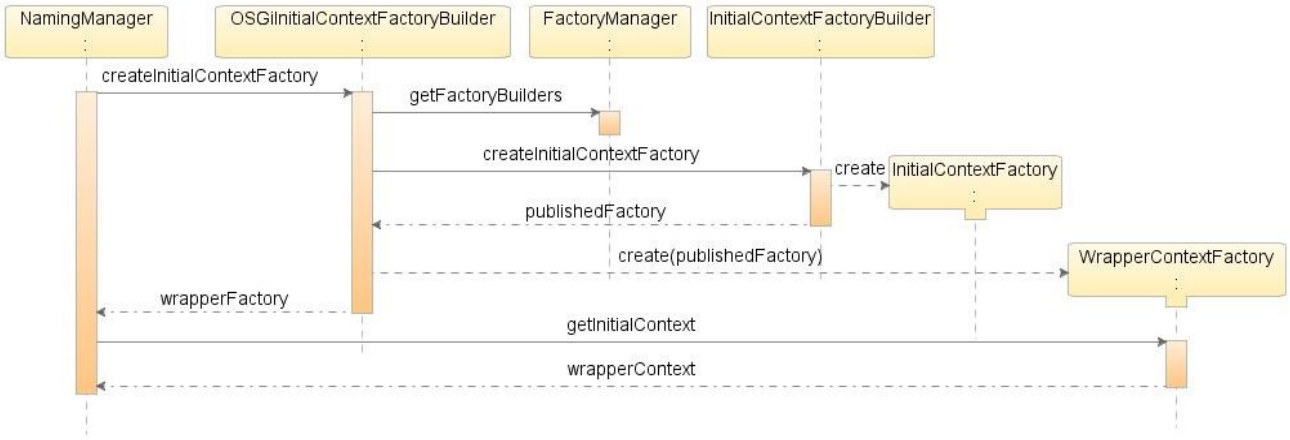

Figure 5-7 Sequence Diagram of InitialContextFactory Wrapper Creation for URL Support

This diagram describes the basic interaction between the JNDI framework and the `OSGiInitialContextFactoryBuilder`. The “Client” and “InitialContext” objects have been removed from this diagram in order to save space. The `NamingManager` is attempting to create a new `InitialContext`. This diagram is similar to Figure 5-3, with the additional calls required to provide wrapper classes for the `InitialContextFactory` returned by the `InitialContextFactoryBuilder`, as well as the wrapper class for the `javax.naming.Context` instance returned by the `InitialContextFactory`. The “wrapperContext” instance returned by the `WrapperContextFactory` corresponds to the “default” context instance mentioned in Figure 5-6.

Since the “wrapperContext” will be used to implement support for URL context factories, it is important to note that the underlying context implementation will not be queried if the “wrapperContext” determines that the given lookup string is a URL. This mimics the behavior of `javax.naming.InitialContext`, since a URL Context Factory can be used with any `InitialContextFactory` implementation. A URL context factory can be used to extend the lookup behavior of any `InitialContextFactory` implementation. This also means that when a URL lookup is being handled, either by the default `javax.naming.InitialContext` implementation or by the method proposed above, the context created by the `InitialContextFactory` is not used to resolve the lookup. The context implementation itself is ignored during a resolution of a URL lookup.

Note:

The proposal above for URL Context factory support focuses on the most commonly used features, which typically involve passing in a URL to Context.lookup() in order to retrieve an object from the context. The following method:

```
javax.naming.spi.NamingManager.getURLContext()
```

can be used by JNDI clients in order to create a URL context based on a given scheme. This method does not delegate to either the `InitialContextFactoryBuilder` *or the* `ObjectFactoryBuilder`*. This means that the JNDI factory manager in OSGi cannot be registered as a builder for this particular method. This will have to be documented as a limitation of the JNDI integration with OSGi. Since this method is in the service provider package, it is unlikely that application developers will require this functionality on a regular basis.*

5.2.7 Limitations

Implementing a bundle to handle this integration with JNDI has the following limitations:

1. The "factory-manager" bundle will need to be installed by default in the OSGi framework. Any bundles that require JNDI will need to be started after this bundle has been installed and also initialized.
2. Due to the nature of the integration point with the JNDI framework, any change to this bundle will cause a full restart of the JVM that the framework is running in. Additionally, if the "factory-manager" bundle is stopped, the classes associated with the factory builder plug-in will not be automatically released, since the JNDI framework does not provide a way to reset this builder reference.
3. The "factory-manager bundle" needs to include the "`DynamicImport-Package`" header in it's manifest file. This is required due to the dynamic nature of JNDI providers. Using this header allows the bundle to respond to any JNDI providers registered on the system, rather than just the bundles that export packages that are imported by the bundle. A component delivered by a JNDI service provider will typically consist of a set of custom factory implementations. The "factory-manager" bundle will not be aware of these types ahead of time. The dynamic import header is required to allow the "factory-manager" bundle to import these types automatically.
4. The "factory-manager" bundle must have the appropriate Java Security permissions to install the `InitialContextFactoryBuilder` and `ObjectFactoryBuilder` instances.

5.2.8 Remote JNDI Clients

It should be noted that while most of the examples mentioned in this document refer to JNDI clients accessing services that are local to the OSGi container that the client is running in, the proposed "factory-manager" bundle will also support JNDI clients that wish to connect to remote services. The JNDI service provider is responsible for publishing an `InitialContextFactory instance` that can provide a remote JNDI context. As an alternative, the JNDI Service Provider could also publish an `InitialContextFactoryBuilder` instance to handle the creation of the InitialContextFactory instance. This factory, which can be created by an `InitialContextFactoryBuilder` or published as a separate `InitialContextFactory` instance, can create JNDI contexts that can connect to a remote JNDI server to access bindings. Provided that such an implementation is available, the "factory-manager" will make this context factory available to clients that require this type of connectivity.

5.3 OSGi Service Integration ~~(Still under discussion at the EEG)~~

5.3.1 JNDI Provider

The factory manager~~OSGi framework~~ will need to provide a default JNDI URL Context Factory for the "osgi" scheme~~provider~~ that can be used by clients in order to resolve lookups of OSGi services published to the registry.

5.3.1.1 Required access to a JNDI Context

The support for the "osgi" URL scheme is provided by a URL Context Factory. Since the lookup request for a URL originally is invoked on a naming context instance, a JNDI context provider (`InitialContextFactory` or `InitialContextFactoryBuilder`) must be available in the service registry at the time of the "osgi"-based lookup.

5.3.2 Service Access

A JNDI client running inside an OSGi-enabled Java EE Container should be able to access a published OSGi service with the following code example:

```
InitialContext context = new InitialContext();

ExampleService exampleService =

(ExampleService)context.lookup("osgi:services/com.example.ExampleService");
```

In this example, a JNDI client running in an OSGi bundle attempts to lookup an OSGi service with the name "com.example.ExampleService". The client uses the "osgi" URL scheme proposed in this document. .

5.3.2.1 Classloader Requirements

In order for the JNDI Factory Manager to properly handle lookup requests for an OSGi service, the Factory Manager must have access to the calling client's BundleContext. This allows the context implementation to get a reference to a given OSGi service on behalf of the calling client, rather than using the Factory manager's BundleContext instance.

Clients must have the current client bundle's classloader set as the Thread context classloader. The URLContextFactory returned by the factory manager to support "osgi"-based lookups will use the context classloader as a means to access the client's BundleContext. This access is provided by the "org.osgi.framework.BundleReference" interface, described in Section 3.8.9 of the OSGi Core Specification, Version 4.2.

This requirement has the following implications for Java EE Containers, as well as any component container that needs to support this type of lookup:

Java EE application servers or containers typically initialize much of the environment surrounding a Java EE application component (servlet, EJB, etc). Containers that need to support this type of lookup via JNDI must set the client Bundle's classloader to be the current Thread context classloader prior to invoking a method on the application component. This makes the BundleContext of the caller accessible to the JNDI implementation, and allows the OSGi service to be properly obtained via the standard OSGi mechanisms.

The OSGi framework will be responsible for making a URL context factory implementation available that can handle the “osgi” URL scheme. At initialization time, the framework should create an instance of this URL context factory, and register it under the javax.naming.spi.ObjectFactory interface. The service should also include a the property “osgi.jndi.urlScheme” to indicate that this service supports the “osgi” URL scheme. The packaging details of this “osgi” URL context factory are not specified. This means that this URL context factory could be packaged within the “factory manager” bundle, within the OSGi framework itself, or within another OSGi bundle. Section 5.2.5 of this document provides more detail about support for URL context factories in general.

5.3.2.2 Handling of OSGi Services

Upon a request for a given OSGi service name, the “osgi” URL context factory will use the following method to locate a matching service:

1. The “osgi” URL context factory will attempt to find an OSGi service that is published under the interface specified in the URL. In the example listed above in Section 5.4.2, the interface to search for would be “com.example.ExampleService”. If an OSGi Service filter is specified as part of the URL, the “osgi” URL Context Factory will use this filter during the attempt to find a matching service.

2. If no services match that interface, the “osgi” URL context factory will attempt to find an OSGi service with the property "osgi.jndi.serviceName" that matches the interface name specified in the URL. For example, a URL with the form “ `osgi:services/anExampleService`” indicates a service name of “anExampleService”. This name does not match the interface name of an OSGi service. The “osgi” URL context factory will search for an OSGi service that includes the “osgi.jndi.serviceName” property. Upon finding a service (or services) with this property, the URL context factory will look for a service with an “osgi.jndi.serviceName” that matches the interface name specified in the URL. If a match is found, the URL context factory will return this service to the JNDI framework. If multiple matches are found, the URL context factory will follow the service ranking rules specified in the OSGi Service Layer specification.

This section includes the following implications for OSGi service providers/implementors:

1. Any OSGi service published to the OSGi registry will be available to JNDI clients at runtime via the service interface.

2. An OSGi service provider that wishes to accommodate mapping an OSGi service to a name that does not match the service interface should publish a given service with the “osgi.jndi.serviceName” property. The “osgi.jndi.serviceName” property should be set to a name that will be used by JNDI clients to locate the service. If the service interface class name is to be used by clients, there is no need to set this property.

The section above proposes one new service property that is related to JNDI:

1. “osgi.jndi.serviceName” - This property can contain a name that will be used when the “osgi” URL context factory attempts to resolve a lookup of an “osgi”-based URL. This property allows a service to be located with a name other than the classname of the service interface. Usage of this property is optional.

The URL context factory implementation for the “osgi” URL scheme will interact with the OSGi Service API in order to locate a service, as well as receive updates on that service's status.

5.3.2.3 Example of Service Handling using Proxies

Note: This section is meant to demonstrate a possible approach towards supporting OSGi service access via JNDI. This particular approach is not required to be used by implementors of the specification.

Before returning a service implementation to the caller, the “osgi” URL Context Factory will generate a wrapper proxy for the service in order to manage the interaction with the underlying OSGi framework.

This proxy could be implemented in several ways, such as:

- Dynamic Proxies used as the wrapper
- A binary library used to dynamically create a class that wraps the Service Reference, such as BCEL or ASM

Maintaining this level of indirection between the service and the JNDI client will allow for the following:

- **Limited Lifecycle Management**: The JNDI Context interface does not provide a mechanism for releasing a resource. Using a proxy approach in this case allows the JNDI implementation to determine when the service is garbage-collected. This provides at least one way to release the resource when the client no longer uses the service. *Note: This approach relies on garbage collection, which is not always guaranteed to occur. This approach is meant as a compromise between the JNDI programming model and the OSGi services model.*
- **General Service Management:** The service reference returned to the caller may, at any time, be de-registered from the OSGi service registry. Having a proxy wrapper allows the JNDI implementation to use the OSGi Service API to detect these scenarios, and throw the correct exceptions. This proxy approach may also allow for the underlying service to be stopped and restarted without the JNDI client having to refresh it's reference.

5.3.3 “osgi” URL Support

The “osgi” URL syntax to be used for resolving OSGi service references via JNDI is as follows:

```
osgi:services/<interface>/<filter>
```

- The scheme of this URL is “osgi”.“services” indicates that this namespace only includes published OSGi services
- <interface> contains the name of an OSGi service that a JNDI client wishes to locate This name may be the actual name of the service interface type, or may be any arbitrary string.
- <filter> contains any service-query filter information. This filter should follow the same syntax that is already present in the OSGi service model.

6 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

6.1 URL Stream Handler Service Analysis

Since the URL Handler Service in OSGi already handles integration support for different URL schemes in an OSGi container, some investigation was done to determine if there was any commonality between the URL Stream Handler integration and the work required by this RFC in order to support URL Context Factories. It was ultimately determined that the APIs and semantics for each framework are not similar enough to foster any re-use that would be helpful in implementing support for URL Context factories. The following sections document the results of this investigation.

6.1.1 Similarities with URL Handler Service Specification

Chapter 11 of the OSGi Specification deals with the integration of Java's URL handler facilities with the OSGi framework. This chapter details some of the difficulties in dealing with some of the integration points in Java for URL handling and content type handling. The key problem lies in the fact that many of the object factory frameworks in Java do allow for a plug-in replacement, but the plug-in typically can only be set once per JVM invocation. This can cause problems with classloading, since classes that are no longer used may not be garbage-collected, since the Java URL handling framework caches references to various factories.

The JNDI/OSGi integration encounters many of the same issues when registering `InitialContextFactoryBuilder` and `ObjectFactoryBuilder` instances with the `NamingManager`. This

specification uses a similar approach of registering one factory that can be used to delegate to any number of factories.

6.1.2 Differences with URL Handler Service Specification

6.1.2.1 Programming Interface

One immediate difference between the JNDI's support for URL context factories and the JDK's support for URL Stream handlers is that each framework provides a separate API for URL support.

Adding support for a new URL handler scheme in the JDK requires subclassing `java.net.URLStreamHandler` as well as `java.net.URLConnection`, while adding support for a new URL context factory in JNDI requires that a developer provide an implementation of `javax.naming.spi.ObjectFactory`. The URL handler API focuses on the creation of a `URLConnection`, while the `ObjectFactory` API only requires the creation of an object or JNDI context based on the contents of a URL. A factory implemented for URL stream handler support will not be interchangeable with a URL context factory. In the same way, a URL context factory will not be interchangeable with a URL stream handler.

6.1.2.2 URL Handler/ObjectFactory locations

While both framework options define a package structure and class-naming scheme for URL support, the requirements are different in each framework.

The javadoc for java.net.URL defines the packaging requirements for a `URLStreamHandler` implementation:

http://java.sun.com/j2se/1.5.0/docs/api/java/net/URL.html

The javadoc for `javax.naming.spi.NamingManager.getURLContext()` defines the packaging requirements for a URL context factory implementation:

http://java.sun.com/j2se/1.5.0/docs/api/javax/naming/spi/NamingManager.html#getURLContext(java.lang.String,%20java.util.Hashtable)

6.1.2.3 Service wrapping

The JNDI/OSGi integration differs from the URL Handler Service Specification in that the interfaces published by JNDI service providers do not need to be wrapped in a service facade. The URL Handler framework must wrap the URL handler interfaces due to access restrictions. The JNDI integration does not require this extra service layer.

6.2 Java EE Requirements Not Yet Supported By this Integration

This section documents requirements and designs that concern features that are specific to Java EE applications. Since the notion of an application does not currently exist in OSGi, these requirements should probably be considered in a later version of the OSGi Enterprise Specification.

6.2.1 Java EE Requirements in JNDI

6.2.1.1 Application Namespaces

A Java EE application can access a variety of resources during the course of it's execution. The Java EE Container provides access to resources such as JDBC datasources, JMS Topics and Queues, EJB references, in addition to other types. When a Java EE application creates a new `InitialContext`, it typically uses the default constructor that takes no arguments. This constructor, with support from the Java EE Container, creates a JNDI context that is suitable for the current application.

The Java EE specification requires that applications have separate namespace boundaries. This means that application “A”'s JNDI bindings are treated separately from application “B”'s bindings. This allows Java EE applications to define reference and binding names without the complexity of name collisions across all of the applications currently deployed in the container.

OSGi frameworks currently do not support the notion of an application as a logical boundary. This means that Java EE application servers that will build upon the OSGi framework will need to create support for handling this. This document proposes a simple model that could potentially be used to solve this problem.

6.2.1.2 Application Deployment

For any type of namespace boundaries to be observed, a Java EE application server will need to deploy applications in such a way that JNDI contexts can be modeled separately. If, during deployment, an application server deploys a Java EE application as a bundle, the container can register an OSGi service that implements the `javax.naming.Context` interface. This service would provide a context implementation that represents the bindings in a given application.

Note: As mentioned earlier in the specification, modeling a Java EE application as a bundle is not explicitly required. The scenario described in Section 5.3.1.1 is meant to illustrate how JNDI service providers and Java EE application servers (using the “factory-manager” bundle) can interact in a way that is compliant with standard OSGi.

6.2.1.3 Context discovery

At various points during the runtime, the bundles that comprise the application server container could track the registration/de-registration of these instances, and provide these Context implementations to Java EE applications at runtime. A Servlet Container, for example, could register a `ServiceListener` for JNDI context-based services.

In order to provide for separation of JNDI namespaces, the Context implementation services should be registered with a set of properties that allow application server components to query for a given application, and perhaps also the version of an application as well. The following properties are proposed:

- javaee.application.name - Representing the application name that this Context instance represents
- javaee.application.version – Representing the application version that this Context instance represents

Allowing bundles to access a Java EE application's JNDI context provides a standard mechanism for locating an application's Java EE services. This also provides an abstraction layer between the bundles that comprise a Java EE application server and the JNDI contexts available for each deployed application.

6.2.1.4 Environment Naming Context

The Java EE specification requires that application servers provide the “environment naming context” to various components within a Java EE application. The environment naming context differs from a plain JNDI context in that the container must enforce several restrictions imposed by the EE specification. The first restriction is on the namespace of the “java:comp/env” context. This context is considered to be specific to each component deployed in a Java EE container. The namespaces for each component's environment naming context need to logically be considered separate. For example: a servlet in application “A” attempts a lookup for a resource reference (“java:comp/env/test_datasource”) defined in the environment naming context. An EJB in the same application may attempt a lookup for a binding of the same name, but the container must treat these bindings as separate objects. This allows components to define namespaces with less chance of collision. The second most noticeable restriction is that environment naming contexts are always read-only, and the Java EE container is required to guarantee this.

It's not clear yet if any support is required at the OSGi framework layer to support this type of JNDI context. In the future, it may be useful to provide primitive operations that simplify the process of managing these special types of JNDI contexts. JNDI service providers could allow for Context instances to be published as services, with properties indicating that this context has Java EE-specific properties. Application server instances can also manage this context handling internally.

This section of the specification is meant to bring to attention the Java EE requirements for JNDI, which tend to be more restrictive on Java EE application servers than a pure JNDI implementation.

7 Security Considerations

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

Obtaining a reference to a JNDI `Context` or an `InitialContextFactory` should be considered a privileged operation and should be guarded by permissions.

The “factory-manager” bundle must be given additional permissions in order to locate and utilize any JNDI providers published as OSGi services. This bundle must have the “GET” permission for every implementation of the JNDI services published in OSGi service registry (`javax.naming.spi.InitialContextFactoryBuilder`, `javax.naming.spi.ObjectFactoryBuilder`, `javax.naming.Context`). This bundle will exist at the system layer, and as such should have more permissions to access services than a standard bundle.

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]. Java Naming and Directory Interface: http://java.sun.com/javase/6/docs/technotes/guides/jndi/index.html

[4]. Java Naming and Directory Interface Tutorial from Sun Microsystems: http://java.sun.com/products/jndi/tutorial/index.html

8.2 Author's Address

Name	John Wells
Company	Oracle
Address	150 Allen Road, Liberty Corner NJ 07938
Voice	(908) 580-3127
e-mail	john.wells@oracle.com

Name	Robert W. Nettleton
Company	Oracle
Address	330 Fellowship Road Suite 100, Mt. Laurel NJ, 08054
Voice	(856) 359-2927
e-mail	bob.nettleton@oracle.com

8.3 Acronyms and Abbreviations

8.4 End of Document