[https://docs.oracle.com/javaee/7/tutorial/overview003.htm#BNAAY](https://docs.oracle.com/javaee/7/tutorial/overview003.htm" \l "BNAAY) Component

<https://docs.oracle.com/javaee/7/tutorial/overview008.htm#GIRDR> JNDI

**Glosary!!**

**Graphs of usages!!**

Components are pretty complex, they might need some additional objects to successfully work. The component might be customized without changing the code. A naming environment is an entity that allows the management of the additional stuff that the component needs, so it provides such functionality. In JavaEE, the container implements such entity (the JNDINameEnv class here) and gives it to the component as a Naming context. It is like a local namespace for the stuff of the component.

The JNDINameEnv is the interface for a naming environment. Some of the implementations are descriptors.

The class is intended to manage the naming environment of the components. Maybe, once the container has an implemented naming environment, this class will be the one that will register all the stuff that is managed by that environment (the dependencies) in the environment context naming context (the subdirectory under java:comp/env) in the server. The registration of the dependencies is done by the naming manager, which receives the binding previously created by the class.

The class provides a general binding operation (to publish all the stuff in the environment) and another one less general, which only publishes the parameters.

In addition to publish the dependencies, the class will keep the count of how many times an environment has been published, together with the instance of the environment. So the class provides also the possibility to get the environment given its ID, and also the environment associated to the current invocation in the container. This might be useful when an invocation on the component requires the use of one of its dependencies, and this might lead to an invocation on the environment.

Before asking the naming manager to publish the dependencies of the component, the corresponding bindings must be created. A binding is a pair <value, object>. The method addJNDIBindings is responsible for doing so. Given an environment, a scope and a (previously initialized) collection of bindings, that method will go through each set of possible dependencies and will create the corresponding binding, if it applies to the scope. Some of the bindings are created directly in the method and some other created by additional methods. The possible dependencies are:

* EnvironmentProperties: added by addEnvironmentProperties.
* ResourceEnvReferenceDescriptors: added by the method. Binds names
* ResourceDescriptors: added by addAllDescriptorBinings.
* EjbReferenceDescriptors: added by the method.
* MessageDestinationReferenceDescriptors: added by the method.
* ResourceReferences: added by…

Note that every binding is created with a name obtained by the descriptorToLogicalJndiName method, and relates it to an implementation of NamingObjectProxy and not to the actual dependencies that were obtained from the environment. Some of those implementations are inner classes into this class. Depending on the properties of the dependency, an appropriate proxy is selected.

One of the methods used to create the bindings is getCompEnvBinding. It is responsible for creating an object of CompEnvBinding, an implementation of JNDIBinding, for the given ResourceEnvReferenceDescriptor. Since the binding is defined by the name and the object related to it, this method creates the name of the resource with the descriptorToLogicalJndiName() method, and associated to the pertinent object. In order to know which specific proxy has to be associated, the method performs a series of validations on the resource to know the exact object that it is. In the case that the specific proxy is not known, a general one is used. Having the name and the value, the binding is created and returned. Note that CompEnvBinding is an inner class.

Among the validations that are performed in getCompEnvBinding to determine the object that the resource is referring to, the method checks whether such object is a Validator. In affirmative case, the object that will be bound to the produced name will be a ValidatorProxy, which is one of the inner implementations of a NamingObjectProxy. The constructor of ValidatorProxy is not public; instead, to obtain instances the create() method is used. Depending on the attributes of the class, different instances are returned or an exception is thrown. The exception might be never thrown, because of the ifs.

When creating the bindings, the addAllJNDI… method has to know whether a dependency has to be added to a given scope. The method dependencyAppliesToScope() is responsible for this. To do so, it checks if the name that was received in the parameters has a prefix that relates it to the also given scope. Note that this method is invoked from the overridden dependencyAppliesToScope() which receives a Descriptor and a Scope.

A JavaEE component can be registered to a naming server, such as JNDI. The dependencies, listed in the descriptor, need to be also published.

In order to use the JNDI server, the container provides an environment in the form of a context (<https://docs.oracle.com/cd/E19159-01/819-3671/ablky/index.html>). The use of the server can be changed in deployment/run time, thanks to the context. It holds some important info of how the server is used.

The class is intended to manage the JNDI environment of a set of components. It provides operations to bind a component to the namespace, unbind it from there, add it there, get the environment for a component ID, get the current environment (the one that is being invoked), get the ID of the component.

The class uses the abstract JNDINameEnvironment for dealing with environments. This is an interface that, when implemented by an object, will provide access to the naming context of that object. In this case, descriptors of components implement such interface. So we could say that an object of JNDIEnv represents both the object (its descriptor actually) and its naming environment.

Based on that information, provided operations use instances of such class in the parameters. Some additional classes are used:

* RefCountJNDINameEnvironment: counts how many components with the same ID have been registered
* FactoryForEntityManagerWrapper
* EjbContextProxy
* ValidatorProxy
* ValidatorFactoryProxy
* WebServiceRefProxy
* EjbReferenceProxy
* CompEnvBinding
* ScopeType

Whenever a component (in the form of an environment) is registered, we calculate the bindings for the additional relevant info, which is contained in the environment, a naming manager is used to publish them. After that, we register the new publishing by means of the RefCount class. EnvManager holds a Map of instances of that class, indexed by the ID of the component that was registered. So the new publishing either creates a new map entry or updates the count of publishings in the already existing one.