**Glossary**

* **Binding:** is a pair *<name, object>* that associates a name with an object.
* **Component:** self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components.
* **Dependency:** any component to which a specific component will communicate in order to work.
* **Environment:** set of all the dependencies of a component.
* **Naming context:** directory in the naming server where all the dependencies from a naming environment are stored.
* **Naming environment:** entity that names the dependencies in the environment of a component in order to manage them.
* **Naming service:** service that provides services to register, unregister and lookup for objects given their names.

**Functional Role**

According to Oracle’s web site: “A Java EE component is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components.”[[1]](#footnote-1) Those other components are called *dependencies*, and the set of all the dependencies of a component is called its *environment*.

As part of the flexibility that Glassfish containers (in which components are deployed) provide to the developers of applications, the environment of a component can be customized without modifying the source code of the related component. Such capability is possible through the abstraction of the environment into an entity that manages the dependencies by naming them, which is implemented by the container; this new entity is a *naming environment*. Whenever a component needs to access its naming environment, the container provides such access in the form of a *naming context*. The naming environment uses a *naming service* to register the dependencies and allow them to be looked up.[[2]](#footnote-2) Not all the dependencies might be reachable from any other component; some could be globally usable, and some others only locally. In order to prevent access conflicts, scopes are associated to dependencies to indicate their reachability.

In particular, Glassfish implements a JNDI server to support the naming service. Thus the provided context is actually a *JNDI naming context*, and the environment is a *JNDI Naming Environment*.

The class *ComponentEnvManagerImpl* is responsible for managing JNDI naming environments; this are represented with the class *JNDINameEnvironment*. In order to give a more meaningful explanation of the four assigned methods, we will describe some attributes and the most important methods. Most of the following information concerning the ComponentEnvManagerImpl was obtained by directly inspecting the source code since the documentation is almost empty; comments inside the code were also helpful.

The attributes which are relevant for our portion of the code are:

* Logger: a private attribute of type *Logger*, injected from the container. Used to print information.
* Naming manager: a package attribute of type *GlassfishNamingManager*, injected from the container. Used to publish and unpublish objects.
* Invocation manager: a private attribute of type *InvocationManager*, injected from the container. Used to get information about the invocations in the containers.
* Map of IDs-naming environments: a private attribute of type *ConcurrentMap<String, RefCountJndiNameEnvironment>*. Associates to each ID of a naming environment its instance of RefCountJndiNameEnvironment.

The class also uses some inner classes:

* RefCountJndiNameEnvironment: is intended to store an instance of a JNDINameEnvironment and the number of times that it has been registered.
* ScopeType: is an enumeration which represents the scope types that could apply to any dependency.
* CompEnvBinding: is a specific implementation of the JNDIBinding interface.

As suggested by its name, and evident in the source code, ComponentEnvManagerImpl is an implementation of the interface *ComponentEnvManager*. We will explain the functional role of the class in terms of the operations that such interface provides:

* Get a naming environment given its ID. This method receives the ID of a JNDINamingEnvironment and returns the corresponding instance, or null if the name is not found. The implementing class looks for the naming environment in a map of instances, and prints some information in a logger.
* Get the naming environment on which the current invocation (from the container) is being executed. This method returns the corresponding instance, or null if no naming environment is being invoked. The implementing class uses an invocation manager in order to get this information, and the previous method.
* Get the ID of a naming environment given the instance. This method returns the unique ID of the received instance. The implementing class uses an external class in order to get such ID; this is also printed in the logger.
* Bind a given naming environment of a component to its namespace. This method receives a JNDINameEnviroment and returns the ID of the component, after having bound all the dependencies; it might throw a *NamingException*. The implementing class uses the method addJNDIBindings() to get the bindings of the dependencies associated to each scope and to publish them through the naming manager. The naming environment is also registered in the map, and some information is written in the logger.
* Add dependencies to the namespace of a component, given its naming environment. This method receives a JNDINameEnvironment, and collections of *EnvironmentProperty* and *ResourceReferenceDescriptor*, and adds these to the namespace of the received naming environment; it might throw a NamingException. The implemented class uses the method addJNDIBindings to get the binding corresponding to the received dependencies and makes the naming manager to publish them.
* Unbind a given naming environment of a component from its namespace. This method receives a JNDINameEnvironment and unbinds all the dependencies; it might throw a NamingException. The implementing class uses the method addJNDIBindings() to get the bindings of the dependencies, and makes the naming manager to unpublish them. The naming environment is unregistered.
* Get the current application environment. This method returns the current application environment (as an *ApplicationEnvironment*) if not running in a specified container.[[3]](#footnote-3)

Besides the previous operations, the class ComponentEnvManagerImpl has two additional public methods:

* Register a naming environment, given the corresponding instance and its ID. Uses the ID of the JNDINameEnvironment instance to add it to the map. If it already existed then the counter is increased.
* Unregister a naming environment, given the ID of the naming environment. Receives the ID and decreases the counter of the corresponding instance in the map; if this gets to zero, the entry is deleted.

After this overall explanation of the class as a whole, we are able to expose the functional role of the four methods that were assigned to us:

* private void addJNDIBindings(JndiNameEnvironment env, ScopeType scope, Collection<JNDIBinding> jndiBindings):

This method has the responsibility of creating and adding the bindings for the dependencies in the naming environment, provided that they apply to the given scope.

The *env* parameter represents the naming environment to be explored; the scope is the scope type for which the method will add the bindings; the *jndiBindings* is the collection of the bindings to which the created ones will be added. A binding is a pair *<name, object>* that will be used by the name manager to publish the dependency.

The 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 invoked methods. Every binding is created with a name obtained by the descriptorToLogicalJndiName method, but the bound object is not the actual dependency obtained from the environment; instead, an appropriate implementation of NamingObjectProxy is used. Some of this implementations are inner classes.

Two of the assigned methods are used here:

* getCompEnvBinding is called to create the bindings for some of the dependencies.
* dependencyAppliesToScope is called to check whether a dependency applies to a scope.
* private CompEnvBinding getCompEnvBinding(final ResourceEnvReferenceDescriptor next):

This method is in charge of creating the appropriate binding for the received descriptor (of a dependency).

The *next* parameter represents the dependency for which the CompEnvBinding object will be created.

The method creates the name of the dependency through descriptorToLogicalJndiName. A series of conditions are checked in order to instantiate the appropriate proxy object to be bound; if no specific case is matched, a generic one is used. In any case, an object of an implementation of the interface NamingObjectProxy is created.

With the name and the object, the instance of ComEnvBinding to be returned is created.

* private boolean dependencyAppliesToScope(String name, ScopeType scope):

This method checks if a dependency applies to a scope, given the name of the dependency and the scope.

The *name* parameter represents the name of the dependency, and the *scope* parameter represents the scope.

The method compares the prefix of the name with the scope and determines in this way the applicability of the dependency to the scope.

* public Object create(Context ctx) throws NamingException:

This method corresponds to the specific implementation that the inner class *ValidatorProxy* gives for the interface NamingObjectProxy. It is intended to create and return an instance of *Validator*, based on the information of the context *ctx*. It might throw a NamingExeption.

The constructor of ValidatorProxy is private since it represents a proxy; that is why this method makes sense. In ValidatorProxy there are attributes to represent the *proxied* object: the *validator* and the *validatorFactory*.

When this method is called the *validator* is returned. In case it was null, the context is used to create an instance. This called is not produced here, but will be probably done when any component looks up for this dependency in the naming manager.

**Graphs of usages!!**

**Issues:**

In getJndi(), getJNDI(String) might return null.

In addJNDI(), not checked null of collection.

In create() exception might be never thrown, double ifs.

1. <https://docs.oracle.com/javaee/7/tutorial/overview003.htm#BNABB> [↑](#footnote-ref-1)
2. <https://docs.oracle.com/javaee/7/tutorial/overview008.htm#GIRDR> [↑](#footnote-ref-2)
3. http://glassfish.pompel.me/com/sun/enterprise/container/common/impl/ComponentEnvManagerImpl.html [↑](#footnote-ref-3)