Norna version 1.0
Draft

Lars J. Nilsson

September 2002

Abstract

The Norna framework is a dynamic Service and Application framework for rapid development and dynamic deployment of applications under a unified runtime. The framework is written in and is primarily intended for use in the Java computer language [1] version 1.4.

This paper presents the Norna framework and is intended for both framework developers and service/application developers.

[1] Java is a trademark if Sun Microsystems, http://www.sun.com/

Contents

Chapter 1

Introduction

Welcome to the Norna specification. This paper is intended for those of you interested in developing Norna framework implementations and services. We will try to keep it informal as well as straight to the point.

1.1 Overview

The Norna framework design is intended for rapid development of services within a single runtime. The main objectives when designing it was:

- Fine grained security control
- Dynamic service loading/unloading
- Modular design through interfaces

It is also specified to depend on Java version 1.4. This version contains numerous features that was considered important for a framework dealing with not only traditional services but also application and servers to offer. Among those features one can note so called non-blocking IO [1] and built in Perl 5 [2] style regular expressions.

1.2 Definitions

This section defines key words and common denominators used within this document.

1.2.1 Key words

This specification defines the following key words:

MUST An absolute requirement. Failing to comply with this requirement will render the implementation incompatible with this specification.

[1] Blocking versus non-blocking IO refers to wether a process the requests a *read* operation from a socket or network device will block until content is available or return immediately.

[2] As in the computer language Perl, www.perl.org

SHOULD This item is not an absolute requirement but implementors are *strongly* encouraged to comply to the described item.

MAY This is a loose recommendation. There may be many reasons for not implementing the item, and implementors can freely choose not to do so.

Any of the above key words can be trailed by an accompanying "not" which denotes an inverse of the key word. An example: Implementors MUST NOT ignore the "must" key word.

1.2.2 Notation

For code examples a fixed width font will be used. It will also be indented. Here follows an example of a trivial interface declaration in Java:

```
package trivial;

public interface Trivial {
    [...]
}
```

Key words will be displayed in all upper case, like this: You MAY criticise this white paper.

1.2.3 Terms

In order to clearly define the framework specified in this document certain terms will be used. The use of these shall be distinct but will not be typographically marked. The following terms will be used:

Service A service is an application, a service or a server loaded and controlled by a Norna framework implementation. The name is meant as a compound since it is not limited to the more traditional service definition of modules that only offer services to its environment.

Platform A Norna implementation in which services are executed. The default implementation is named "Urd" and is developed in tandem with the Norna specification.

1.3 Platform Definition

The platform is a concrete implementation of the Norna specification. The default implementation is called Urd and is developed by the Norna team. Since Norna is targeted to the Java language most platforms will be OS independent. However, the Java dependency is not a requirement and implementors MAY choose to implement parts or indeed the whole of the specification in other languages, but MUST be able to launch services in Java regardless of the platform implementation language.

1.3.1 Languages

Java

The main language for the Norna specification and all platforms is Java version 1.4 [3]. Version 1.4 for chosen since it offers significant features for server oriented application over its previous versions. Among those features the following was regarded as significant:

- Non-blocking IO offers a better scalability for servers in an order of magnitude. It also allows a lower process profile for the services since *read* operation no longer needs to be performed by a dedicated thread.
- Build in regular expressions is a strong tool to use in a any text related protocols. This have been available in separate packages for some time, but the inclusion in the main Java API standardize its use across different platforms.
- Speed improvements for reflective operations. The Java Reflection API is a very strong tool for Java programming in general and is almost a requirement when partitioning code into separate class loaders to allow inspection without class dependencies.

Others

Although no support for native code or other languages than Java is specified in this version this is something that might appear in future revisions and platform implementation MAY choose to include such features.

[3] Sun Microsystems, http://java.sun.com/j2se/

Chapter 2

Framework

2.1 Introduction

This chapter specifies the framework design and its components.

2.2 Overview

The Norna framework is a modular interface based design for services and common rules for platform implementations.

The framework comprises of a set of interfaces in the "net.larsan.norna" package, which defines the platform interface towards its hosted services, and another set of interfaces in the "net.larsan.norna.base" package, which defines the interfaces services may choose to implement to specify its functionality in regards to its platform host environment.

Aside from the interfaces the specification also defines several areas of behaviours for the platforms to implement. These areas are:

Service archives How to package and interpret standard services.

Platform namespace How services shall be named, identified and located.

Class loading Class loading security and how to share service interfaces across class loader boundaries.

The specification also contains a number of default services which MUST be implemented by every platform. These services include a user services for authentication and authorization which is based upon the JAAS [1] API, a logging service and a preference service.

2.3 Namespace

2.3.1 Overview

One problem that arises in any environment is that of addressing. This gets more problematic in a dynamic environment that not only spans multiple class

[1] The Java authentication and Authorization Specification

loader boundaries but also might encompass multiple virtual machines.

The Norna answer is a namespace syntax which uniquely defines the current platform and every public service within it. The addressing in the namespace is done using a common URL syntax:

```
protocol://host[:port][/path]/file
```

The URL protocol for Norna the namespace MUST be named 'norna'.

The namespace root is considered the host name of the current platform. For example: A host named 'www.larsan.net' using port 8666 for its Norna installation would start every formal URL with:

```
norna://www.larsan.net:8666
```

The sub-paths following the host name is platform dependent and is given to the services when they are first loaded. This sub-path will form the unique ID for any given service within any given namespace.

Each service within a platform is mounted on a path within the namespace. This path is the unique ID for the service during it's lifetime and is the ID by which it is identified to other services.

Namespace paths are relative to the entry-point from which they are called.

- A path starting with "norna://" is an absolute URL and might span several namespaces.
- A path starting with a single slash, "/", is relative to the root of the current namespace.
- Any other path is relative to the current location in the current namespace.

The namespace forms a context for each platform. This context is available for the services through the "Context" interface which gives them access to the namespace. The namespace as seen from any given context might expand over several logical boundaries, such as class loaders or different runtimes, but MUST be kept transparent to the individual services. For example, the two services:

```
norna://localhost/services/Echo
norna://localhost/services/Calculator
```

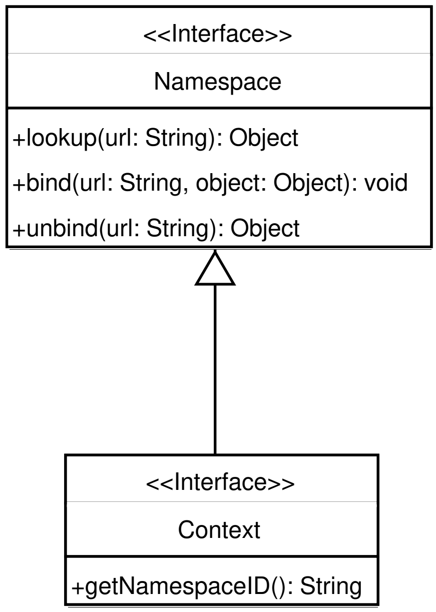
Might exist on different runtimes, but they are mounted within the same namespace, e.g. "localhost", so if the "Calculator" service would attempt to use the "Echo" service it must be able to do so as if the "Echo" service existed locally within its own runtime.

2.3.2 Context

The context is an interface published to the services be the platform. It is the service entry-point to the local namespace. Services taking part of the local namespace is said to be "loaded" when they are connected to the namespace and "unloaded" when they are not. In accordance to the interface design of Norna a service must declare its interest in loading the namespace by implementing the "Loadable" interface.

Context Interface

The context is available to every service implementing the "Loadable" interface. The context interface is an extension of the namespace interface, shown below:



Thus every service with access to a context can lookup, bind and unbind objects within the namespace. It should be noted that must platforms will limit access to most parts of the namespace for any given service. A platform SHOULD limit access to everything except sub-paths of a service location using a "ContextPermission".

The namespace ID of a service is its namespace unique name and is further discussed in the section about service addressing.

Delegation

In order to enable dynamic loading of resources through the namespace, Norna specifies a method of namespace delegation. It does so through a "Delegator" interface which is a trivial extension of the "Context" interface. The "Delegator" interface can be implemented by services and posted in the namespace. The interface will then mark a delegation boundary within the namespace for lookups so that every namespace lookup for a sub-path where the "Delegator" was bound will be referred to the "Delegator" instead of the main namespace.

Let's look at a practical example. A service wants to post user information on the namespace for other services to access. But instead of posting names and addresses as object instances in the namespace it chooses to use the "Delegator" interface and handle every namespace lookup for a user dynamicly. The service is bound in the namespace at:

```
/services/Users
```

It now binds a "Delegator" interface implementation the this path:

```
/services/Users/info
```

Now every lookup on a sub-path to '/service/Users/info' will be delegated to the "Delegator" implementation instead of the main namespace. So if another service wants to find the address of user "dummy" it might lookup the following object through its context:

```
/services/Users/info/dummy/address
```

The look up above would get translated into two parts, the first part finding the the "Delegator" implementation posted at "/services/User/info" and the second part invoking the "Delegator" with the following method:

```
lookup("dummy/address");
```

From the platforms point of view, this would be implemented with something like this:

```
Delegator del = (Delegator)lookup("/services/Users/info");
del.lookup("dummy/address");
```

Again, platform implementations are free to guard access to the namespace using a "ContextPermission". This will likely determine were a given service may safely bind a delegator instance.

2.3.3 Service Addressing

Every service is bound to the Norna namespace. This path of the binding also acts as the ID for the service. A platform MUST guaranty that a service instance ID is unique and immutable through a service lifetime. However, the platform MAY choose to give a particular service a different ID for each service lifetime.

- A service ID is only guaranteed to be valid during the service lifetime.

There are three distinct variation on a service ID, depending on the view point from which it is used. The most commonly used ID is the is the Namespace ID. A platform implementation MUST be able to recognize and use all of a service available ID.

Namespace ID

This ID locates the service unique position within its namespace and is the root path of the service context. The Namespace ID is the most commonly used ID within a single platform. It has the following characteristics:

- It is unique within a single platform only
- It is the path relative to the root URL of the platform namespace
- It is only valid during a single service lifetime

A service will get hold if it's Namespace ID though the context interface it gets by implementing the "Loadable" interface. A platform MAY choose to give the services other ways of knowing their Namespace ID but services SHOULD NOT depend on it.

For example: A service mounted the URL "norna://localhost/service/Echo" would have the following Namespace ID:

```
/service/Echo
```

Global ID

The Global ID is the full namespace URL of a service. It is global since it not only contains the Namespace ID but also contains the namespace name of the service platform. It has the following characteristics:

- It is unique over multiple platforms
- It is the full URL of the service
- It is only valid during a single service lifetime

For example: A service mounted the URL "norna://mailhost/service/Echo" would have the same Global ID as it's URL, e.g.:

```
norna://mailhost/service/Echo
```

Public ID

The Public ID is the ID the service choose to give itself. It can be composed of any alphanumeric characters in any given order and is presented to the platform through a "SoftwareInfo" interface the service MUST implement. It has the following characteristics:

- It is not guaranteed to be unique
- It is chosen by the service alone
- It is may be valid over several service lifetimes

2.3.4 Reserved Addresses

The namespace ID */norna/–* (where the trailing hyphen recursively includes all subdirectories) is reserved. This namespace hold the Norna Framework services and the service registry.

/norna/log The Norna log service

/norna/users The Norna user service

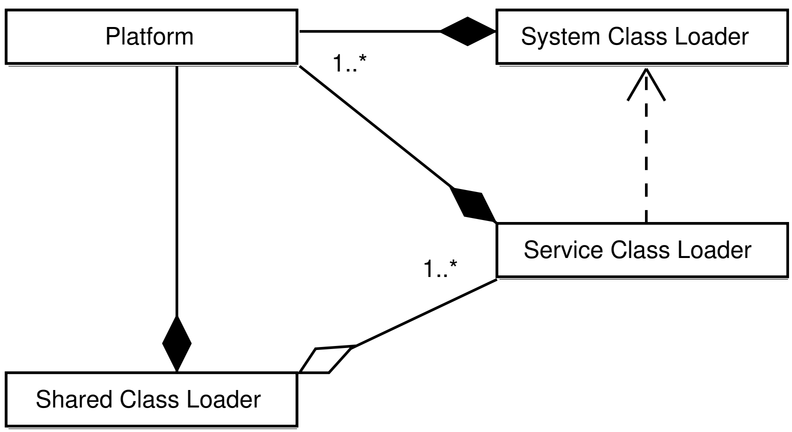
/norna/registry The Service Registry

2.4 Class loading

2.4.1 Overview

In order to secure service instances platforms SHOULD partition the class loaders by using one class loader for each service. Only in extreme low resource environments should the platform allow several services to share a class loader.

Java security mechanism uses class loaders as namespaces for the loaded classes, and as such two identical classes will still not be interchangeable if they are loaded by two different class loaders. To overcome this the class loader Norna allows single services to "import" and "export" packages publicly to make sure a service interface is only loaded from one single place. This implies that every platform must use a "shared class loader" for public classes every service choose to export.



For example: If service "x" and service "y" is loaded by different class loaders "x" will not be able to use "y" since their classes are loaded from two different places. Norna solves this by allowing every service to export one or more packages. Service "x" now places its public contract in an interface in a separate package and exports that package. This exported interface will now be available for all services within the platform.

Platforms SHOULD make sure no service is allowed access to shared packages without an explicit import declaration.

2.4.2 Public Packages

Packages exported by a service is made public through a shared class loader. The shared class loader shares its lifetime with the platform which opened it. Thus, once exported a shared class cannot be updated within the lifetime of the containing platform.

Exported Packages

A service may choose to export one or more packages. It is recommended that services export a package containing their public contract if they have any since this is the only way services loaded by different class loaders can share the service functionality.

Once exported, a class cannot be updated within the lifetime of the containing platform.

For security reasons a service MUST explicitly declare all packages it wants to export before it is loaded and a platform MUST NOT allow any service to export undeclared packages.

Imported Packages

In order to use public exported packages a service must first import them. Just as exported packages MUST be explicitly named before the service is loaded a platform SHOULD refuse a service access to any shared package it has not explicitly imported.

2.5 Service Archive

2.5.1 Overview

In order to be able to support multiple platform implementations Norna specifies an archive format the all platforms MUST support. The archive uses the Java JAR format using specialized fields in the JAR manifest.

One service is loaded through one archive only. The platform uses the manifest to find the service main class and also to find packages the service wishes to export or import.

A future extension might provide for multiple services to load from the same archive but the current specification only allows for a one-to-one mapping between services and archives.

2.5.2 Manifest Headers

In order to successfully load a service from an archive the platform looks for customized header fields. A JAR archive contains a manifest file which is composed of header fields with values, usually in ASCII. The JAR file format can be found at http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html.

Manifest headers MUST be constructed in strict conformance to the JAR specification. And a platform MUST:

- Ignore unrecognized headers
- Only parse the main section of the manifest
- Refuse to load services carrying malformed headers

Norna-Service

The main class of the service will be located by the platform using a manifest header named "Norna-Service". This header MUST exist for the service to be loaded and MUST consist of a single value containing the fully qualified class name of the service.

Export-Package

A service MUST declare all exported packages using a specialized header field named "Export-Package" containing a comma separated list of all packages the service wants to export.

Import-Package

A service MUST declare all imported packages using a specialized header field named "Import-Package" containing a comma separated list of all packages the service wants to import.

2.6 Service

2.6.1 Overview

A service in the Norna Framework is defined by a set of interfaces. The root package of these interfaces is the "net.larsan.norna.base" package. A service may choose to implement one or more interface depending on what functionality it requires from the platform, with the exception that the service definition interfaces are mandatory.

The platform in its turn, inspects the main class – as defined by the "Norna-Service" archive manifest header – and acts on the interfaces the service has declared. This must happen before the service is instantiated.

2.6.2 Service Interfaces

The service interfaces as defined in the "net.larsan.norna.base" package concerns service definition and lifetime. Of those interfaces the lifetime interfaces are optional but the definition interfaces MUST be implemented.

2.6.3 Service Lifetime

A service lifetime is not specified by the Norna Framework. Instead there are decided transitions where a service might go from one place to another, and the service interfaces roughly corresponds to different states within the lifetime of one service.

Then we find that the lifetime of a service cannot be generalized. But the lifetime order of the service interfaces can, and the interfaces are used by the platform in this order:

1. Initializable (init)
2. Loadable (load)

3. Startable (start)
4. Startable (stop)
5. Loadable (unload)
6. Initializable (destroy)

Every one of the above interfaces are optional and when the only interface which is allowed to be repeated is the "Startable" interface, the rules for transitions becomes:

- A call to `Initializable.init` may legally be followed by by method calls to `Loadable.load`, `Startable.start`, or `Initializable.destroy` but no other interface calls.
- A call to `Loadable.load` may legally be followed by method calls to `Startable.start`, `Loadable.unload`, or `Initializable.destroy` but no other interface calls.
- A call to `Startable.start` may legally be followed by `Startable.stop` but no other interface calls.
- A call to `Startable.stop` may legally be followed by `Startable.start`, `Loadable.unload`, `Initializable.destroy` but no other interface calls.
- A call to `Loadable.unload` may legally be followed by method calls to `Initializable.destroy` but no other interface calls.
- A call to `Initializable.destroy` may not be followed by any interface calls.

A service which is constructed by the platform but not yet have entered any interface imposed transitions is said to be "created". But none of the above interfaces are mandatory, a service can be "created" by the platform and directly report a "ready" status to mark itself ready for use.

2.6.4 Service Definition

There are three interfaces the collaborate to define a service as seen from the platform. The interfaces in this category are mandatory.

Service

A service MUST implement the "Service" interface. This is a prerequisite for the platform to load the service in the first place. The class which is named as the main Norna service class MUST implement this interface and the platform MUST NOT instantiate a service that does not fulfill this contract.

The service interface is what the platform see and uses. It will not be publicly published. Visible parts of the service are defined in the "ServiceHandle" and "SoftwareInfo" interfaces.

```
package net.larsan.norna.base;

import net.larsan.norna.*;

public interface Service {

    public void setStatusCallback(StatusCallback callback);

    public SoftwareInfo getServiceInfo();

    public ServiceHandle getServiceHandle();

}
```

The "StatusCallback" interface is a call back through which the service may notify the platform when it changes status. A service status is in its turn an immutable trivial class implemented as singleton static member of itself. The singleton implementation makes sure the referential integrity is kept for this object. For example, the following code is guaranteed to work:

```
Status current = // ... current status
if(current == Status.READY)
```

A service SHOULD make every attempt to use the call back handler. This will give the platform an opportunity to determine when the service is ready for public use or have finished extensive initiation.

ServiceHandle

The service handle is the public handle of a service which is used by other services. This interface is trivial and it is expected that services extend it with their own public contract.

```
package net.larsan.norna.base;

import net.larsan.norna.*;

public interface ServiceHandle {

    public Status getStatus();

}
```

As shown above, the only mandatory method of a service public contract is a status accessor. This accessor MUST be implemented since it currently is the only way a participating service may know if the handle it is using is still valid. Future revisions of this interface may include a "Lease" object to better control handle validation.

SoftwareInfo

The software info interface provides the platform with static information account a particular service. This information is public, can be searched through the service registry, and includes the service Public ID.

```
package net.larsan.norna.base;

public interface SoftwareInfo {

    public String getPublicID();

    public String getSoftwareName();

    public String getOriginator();

    public String getRelease();

    public double getVersion();

    public String getDescription();

}
```

It can be noted that the information within the software information object is not necessary unique within the platform. It is used to uniquely identify a software release as opposed to an instance in the runtime.

2.6.5 Lifetime Interfaces

These are the lifetime interfaces a service MAY choose to implement.

Initializable

The initializable interface should be inplemented by a service to indicate that it need initiation information from the platform. This information will be presented in a "InitParameter"

Once initiated the service becomes eligible for destruction.

```
package net.larsan.norna.base;

import net.larsan.norna.*;

public interface Initializable {

    public void init(InitParameters param)
        throws UnavailableException;

    public void destroy();

}
```

Loadable

This interface should be implemented by services wishing to take part of the platform context. In so doing the service will also become aware of it's Namespace ID.

```
package net.larsan.norna.base;

import net.larsan.norna.Context;

public interface Loadable {

    public void load(Context context);

    public void unload();

}
```

Most interfaces in the Norna Framework are matched two and two. The "unload" call effectively balances the call to "load" and the service MUST NOT use the context after unloading.

Startable

The startable interface should be implemented by services that have a well defined lifetime process of their own and "interact with" as opposed to rely on "interaction from" other services. The startable interface is the only interface that may be repeated.

```
package net.larsan.norna.base;

import net.larsan.norna.*;

public interface Startable {

    public void start() throws UnavailableException;

    public void stop();

}
```

The startable interface is the only interface that may be repeated. A call to "stop" may legally be followed by a call to "start".

Restartable

This interface does not rely on the "Startable" interface. Despite its similarity in names, this interface indicates that a service want to be completely stopped, unloaded and destroyed. And the restarted. The platform will effectively kill all resources in regards to the service including dedicated class loaders and attempt to re-create, initiate, and start the service.

A service requests a restart through the "RestartListener" interface which is handled by the platform.

```
package net.larsan.norna.base;

import net.larsan.norna.RestartListener;

public interface Restartable {

    public void setRestartListener(RestartListener listener);

}
```

The platform SHOULD, but is not required to, honour a request to the restart listener.

ShutdownListener

The shutdown listener interface can be used by services that wishes to be notified in advance of platform shutdowns. The platform SHOULD make every attempt to honour this interface but might be forced to shutdown without doing so.

```
package net.larsan.norna.base;

public interface ShutdownListener {

    public void shutdownWarning(long ttl);

}
```

If this interface is honoured by the platform, the "ttl" argument will give the "time to live" before the platform will start closing. However, the service MUST NOT rely exclusively on a warning from this interface since the platform can be forced to exit at any time.

2.6.6 Framework Interfaces

The platform contains some interfaces that are posted towards it's services, most of which we have already mentioned.

The platform should make every possible attempt to shield itself from the services it is hosting. It is vital that no service can interrupt the framework.

Environment

The environment is a thin interface for the framework environment. The environment is available from the service registry.

```
package net.larsan.norna;

public interface Environment {

    public String getNamespace();

    public String getRootAddress();
```

```
    public int getRootPort();

}
```

The namespace is the root of the platform URL. The root address is the address part of the namespace and the port is the port of the namespace, which might be implicit.

ServiceRegistry

The service registry is for searching and controlling services. Each service is available to other services by two means, through the context or through the service registry. The service registry is available on the reserved namespace ID "/norna/registry".

We'll go through this interface a bit at the time.

```
package net.larsan.norna;

import java.util.Iterator;
import java.util.Properties;
import net.larsan.norna.base.ServiceHandle;
import net.larsan.norna.base.SoftwareInfo;
import java.util.regex.*;

public interface ServiceRegistry {

    public Environment getEnvironment();
```

The environment contains the root address and the namespace of the platform. The platform MAY choose to make the environment available to service outside the registry as well.

```
    public Iterator getByPublicID(String publicID);

    public ServiceHandle getByURL(String url);
```

The "getByPublicID" method returns an iterator of namespace ID of services with the specified public ID. It returns an iterator since a public ID is not necessary unique.

The "getByURL" method takes a full namespace URL or an namespace ID and return the service handle for that service or null if not found.

```
    public Iterator search(Properties attributes)
        throws PatternSyntaxException;
```

The search function takes a properties object that links short hand properties to regular expressions for searching. The properties available is taken from a service software info object and their shorthands are:

- "puid" – Service public ID

- "url" – Service namespace URL
- "name" – Service name
- "vendor' – Service vendor
- "description" – Service description
- "release" – Service release
- "version – Service version

The search will return an iteration over service namespace string URL.

```
public Iterator list();

public SoftwareInfo getSoftwareInfo(String url);
```

List all available services as string URL. And return the software info for a given service.

```
public void start(String url)
    throws NoSuchServiceException;

public void stop(String url)
    throws NoSuchServiceException;
```

Start and stop a service by it's namespace URL. The platform SHOULD guard this methods using "ServicePermission" objects.

```
public void addRegistryListener(RegistryListener l);

public void addRegistryListener(Properties filter,
                                RegistryListener l);

public void removeRegistryListener(RegistryListener l);

}
```

Add and remove service registry listeners. The filter functionality works as a search for a service would do, the shorthand of the service properties will be matched against a regular expression and only if it matches will the event be delivered to the listener.

2.7 Security

2.7.1 Model

The Norna security is based upon the highly successful Java 2 Security Model [2]. It uses permissions to guard methods during runtime.

[2] See: http://java.sub.com/security

Apart from the permissions of the running code the platforms SHOULD make sure the framework is secure from tampering by loading service in separate class loaders and verifying parameters passed from the services into the framework.

Other possible security breaches for the platform is events dispatched into a service from the registry. Such events MUST be delivered asynchronously to protect platform integrity.

2.7.2 Permissions

The framework defines three permissions to guard parts of its interfaces. These permissions will be set by the platform and MAY optionally be granted to separate services. The following rules apply:

- All permission actions and targets are case-sensitive.
- All permission uses multiple actions. These actions can be specified in comma separated lists.
- All permissions accepts a "*" character to indicate a blanket wild-card, either on actions or targets.

ContextPermission

The context permission checks access to the service context. The target of the permission is the namespace context the permission regards and the actions correspond to one or more of "bind", "lookup" or "unbind" in a comma separated list.

The context path is always relative to the namespace root.

The path may end with a hyphen to indicate sub-paths of the named context.

PackagePermission

The package permission is needed for service that wants to import or export packages. The permission target correspond to a package name and its action can be either "import" or "export", or both separated by comma.

The package name may end with a hyphen to indicate that it also covers all available sub-packages.

ServicePermission

This permission is used by the service registry. The permission target is a namespace ID or context and the action can be one or more of "get", "status", "start" or "stop" in a comma separated list.

The target may be an explicit namespace ID or a context path. It may end with a hyphen to indicate that it also covers sub-path of the context.