BEA Extension SDK for BEA WebLogic Network Gatekeeper™®

Developer Guide

Version 2.1
Revised: December 16, 2005

Contents

Using the Extension SDK templates

Introduction and Roadmap

The following sections describe the audience for and organization of this document:

- Document Scope and Audience
- Guide to this Document
- Terminology
- Related Documentation

Document Scope and Audience

The purpose of the document is to information on how to use the Extension SDK for BEA WebLogic Network Gatekeeper to create extensions to Network Gatekeeper.

The first part describes the different software modules from various perspectives:

- Actors
- How to use the utilities provided by the SLEE
- General sequence diagrams
- Frameworks for the different layers
- High availability aspects
- Specifics regarding call control plug-ins
- Specifics regarding messaging plug-ins

The second part provides information on the templates and build environment for creating extensions that is provided by the Extension SDK.

The following topics are covered:

- How to create a plug-in
- How to create an ESPA Service Capability module
- How to create a SESPA Service Capability module
- How to create a WESPA Service Capability module
- How to create a test application using the new interfaces
- Information on how to use the SLEE utility services

Intended audience is system integrator and field engineers with an interest in how to extend the functionality of the WebLogic Network Gatekeeper.

Prerequisites

In order to use the Extension SDK, Java and CORBA knowledge is essential.

It is also a prerequisite to know the architecture and to have hands on experience working with the WebLogic Network Gatekeeper.

Guide to this Document

The document contains the following chapters:

- Introduction and Roadmap, informs you about the structure and contents of this document, and other WebLogic Network Gatekeeper related documentation.
- Actors,outlines the different software layers and the actors involved.
- Interacting with the SLEE and the SLEE Utility Services, contains information about how to use the utilities provided by the SLEE.
- General sequence diagrams, contains generic sequence diagrams that outlines how the different layers interact with each other.
- Frameworks, explains the frameworks used in the different layers
- High availability, describes high availability aspects.

- Plug-ins that executes as a SLEE service and a web application, describes how to interact between a SLEE Service and a Web Service executing in Tomcat via the SLEE Common Loader. The focus is on network plug-ins.
- Call Control, contains specifics regarding call control plug-ins.
- Call user interaction, contains specifics regarding call user interaction plug-ins.
- SMS and MMS messaging, contains specifics regarding messaging plug-ins.
- Content based charging, contains specifics regarding content based charging plug-ins.
- Subscriber profile, contains specifics regarding messaging plug-ins.
- User Location, contains specifics regarding messaging plug-ins.
- Policy rules and Policy Utilities, explains how request data is used in Policy rules, and how the rules can be expanded using Policy Utility classes.
- Using the Extension SDK templates, contains information about the Extension SDK, the different software modules, the file structure and more.
- Creating an example network plug-in, explains how to create a plug-in based on the templates and build environment provided in the Extension SDK.
- Creating an example ESPA Service Capability module, explains how to create a service capability module based on the templates and build environment provided in the Extension SDK.
- Creating an example SESPA module, explains how to create a SESPA module based on the templates and build environment provided in the Extension SDK.
- Creating an example WESPA module, explains how to create a web service interface based on the templates and build environment provided in the Extension SDK.
- Creating an example application, explains how to create an application that uses the Web Services exposed by the WESPA module provided in the Extension SDK.
- Creating an example network simulator, explains how to create a simulator application that is tied to the network protocol plug-in.

Terminology

The following terms and acronyms are used in this document:

- API—Application Programming Interface

- Application—A telecom enabled computer application accessed either from a telephony terminal or a computer.
- Service Provider—An organization offering services provided by one or more applications to end users.
- AS—Application Server
- ATE—Application Test Environment
- CBC—Content Based Charging
- CORBA—Common Object Request Broker Architecture
- End User—Person that uses an application. An end user can be identical to a subscriber, for instance in a prepaid service. The end user can also be a non-subscriber, for instance in an automated mail-ordering application where the subscriber is the mail-order company and the end user is a customer to this company.
- Enterprise Operator —See Service Provider.
- ESPA—Extended and value added telecom web services APIs and service capabilities.
- HTML—Hypertext Markup Language
- IIOP—Internet Inter-ORB Protocol
- IN—Intelligent Network
- INAP—Intelligent Network Application Part
- IOR—Interoperable Object Reference
- IP—Internet Protocol
- JDBC—Java Database Connectivity, the Java API for database access.
- MAP—Mobile Application Part
- Mated Pair—Two physically distributed installations of WebLogic Network Gatekeeper nodes sharing a subset of data allowing for high availability between the nodes.
- MPP—Mobile Positioning Protocol
- NS—Network Simulator
- Operator—The Network Gatekeeper owner

- ORB—Object Request Broker
- OSA—Open Service Access
- PAP—Push Access Protocol
- Plug-in—A network plug-in the Network Gatekeeper to a network based service node or OSA/Parlay SCS through a specific protocol.
- SCF—Service Capability Function or Service Control Function
- SC—Service Capability
- Service—A network provided service capability.
- Service Capability—See Service
- SIP—Session Initiation Protocol
- SLEE—Service Logic Execution Environment
- SLEE Service—A software module that is designed to execute in the SLEE.
- SMPP—Short Message Peer-to-Peer Protocol
- SMS—Short Message Service
- SMSC—Short Message Service Centre
- SNMP—Simple Network Management Protocol
- SOAP—Simple Object Access Protocol
- SPA—Service Provider APIs
- SS7—Signalling System 7
- Subscriber—A person or organization that subscribes for an application. The subscriber is charged for the service usage. See End User.
- SQL—Structured Query Language
- TCP—Transmission Control Protocol
- User—An application accessing services through one or more APIs and has a user name and a password or a person working with OAM through the Network Gatekeeper management tool that has an administrative user name and password.

- USSD—Unstructured Supplementary Service Data
- VAS—Value Added Service
- VLAN—Virtual Local Area Network
- VPN—Virtual Private Network
- XML—Extended Markup Language

Related Documentation

WebLogic Network Gatekeeper documentation

This document is a part of WebLogic Network Gatekeeper documentation set. Other documents includes:

- Product Description - WebLogic Network Gatekeeper

  The product description describes functionality and architecture of the WebLogic Network Gatekeeper.

- User's Guide - WebLogic Network Gatekeeper

  The user's guide describes WebLogic Network Gatekeeper related operation and maintenance.

- Application Developer's Guide - Parlay X for WebLogic Network Gatekeeper

  The developer's guide describes how to design and implement applications using the Parlay X Web Services exposed by WebLogic Network Gatekeeper.

- User's Guide - WebLogic Network Gatekeeper Application Test environment

  The user's guide describes how to use WebLogic Network Gatekeeper ATE when it comes to application test.

- API Descriptions - Parlay X for WebLogic Network Gatekeeper

  The API descriptions describe WebLogic Network Gatekeeper Parlay X APIs available for developers and applications.

Actors

The following sections describe how to extend WebLogic Network Gatekeeper and the different actors involved when creating extensions:

- Extending WebLogic Network Gatekeeper
- Network plug-in
- Plug-in manager
- SC manager
- Service capability
- Web Services interface implementation

Extending WebLogic Network Gatekeeper

The WebLogic Network Gatekeeper provides an modular software architecture that allows for extensions to the traffic flow stack of the product, all the way from the northbound, application-facing, interface implementations to network protocol plug-ins. See the Product Description for information Architecture and functionality provided by the WebLogic Network Gatekeeper.

The base software modules for a traffic flow stack are:

- Web service interface implementations, that provides high level interfaces to applications.

- Stateless adapters, that acts as an adapter between the stateless Web Services interfaces and the stateful Service capabilies.
- Service capabilities, that provides a general abstraction of capabilities of a certain type, for example messaging, call control, and user location.
- Network protocol plug-ins, that handles the protocol-specifics of the underlying network node.

When extending Network Gatekeeper, several options are available:

1. Creating a whole new traffic path from the implementation of the northbound, application-facing, interfaces down to the network protocol plug-ins.
2. Creating a new implementation of the northbound, application-facing, interfaces, and map it to an existing stateless adapter (SESPA module).
3. Creating a new network protocol plug-in, and use the existing northbound, application-facing, interfaces and service capabilities.

Also, a combination of the two latter cases can be an option.

Which option to use depends on several factors.

If a custom northbound, application-facing, interface implementation is being created, it may be sufficient to create just the implementation of the interface and map it to an existing Stateless adapter or Service capability module as outlined in option 2. Things to consider when doing this is that CDRs generated will be identified to come from the Service Capability and SLA data and rules will be the ones offered by the Service Capability.

If the implementation of the northbound, application-facing, interfaces does not sit comfortably on an existing Stateless adapter, it may be necessary to create a whole new traffic path, as outlined in the first option.

If the northbound, application-facing, interfaces provided per default are sufficient, option 3 is suitable.

Since the software module executes in the execution environment offered by the SLEE, the software module needs to implement the interfaces provided by the SLEE and use SLEE utility services as outlined in Interacting with the SLEE and the SLEE Utility Services.

When creating new plug-ins and ESPA Service capabilities, it is also necessary to interact with the Plug-in manager and the SC manager.

Network plug-in

Traffic interfaces

The network protocol plug-in implements the protocol specifics, and acts as the telecom network-facing part of the traffic stack in WebLogic Network Gatekeeper.

A plug-in is directly associated with a Service capability, so the plug-in interfaces with one and only one plug-in interface, there are plug-in interfaces for:

- Call Control
- Charging (charging based on content)
- Messaging
- User location
- User status
- Subscriber profile
- User interaction (call and message based)

If a new service capability is introduced, it has a plug-in interface specific to that service capability.

A plug-in has a set of properties:

- PLUGIN_TYPE, which defines which service capability it is associated with. The property defines the plug-in type that the plug-in specifies when registering in the plug-in manager. Service capability implementations will use this type identifier when retrieving plug-ins for handling service requests. The type must match one of the allowed types in the plug-in manager service. Custom types are allowed, but must be registered in the plug-in manager.
- SUBTYPE, which defines a sub-type of an interface where relevant. Examples are GMS (messaging), USSD, or SMS. Subtype specifies a subtype of a specific resource interface. Note that one resource or SC may support multiple subtypes.
- TrAddressPlan, which defines which address plans the plug-in supports. Examples includes IP and E.164.

All these properties are taken into account by the Plug-in manager when the Service Capability is provided with a plug-in.

Note: Additional parameters, such as routing criteria and policy based routing settings are also taken into consideration when routing a request from a Service Capability to a plug-in. These parameters are configurable in contrast to the properties listed above.

Below is a table outlining the properties.

Table 2-1 Service Capabilities, plug-in types and plug-in subtypes

Service capability	Plug-in to be used by the Service capability	Subtype of plug-in
Call Control	CALL_CONTROL_TYPE	
Call user interaction	USER_INTERACTION_CALL_TYPE	
Charging (charging based on content)	CHARGING_TYPE	
Messaging	MESSAGING_TYPE (for SMS) MMS_TYPE (for MMS)	SMS MMS
Messaging user interaction	USER_INTERACTION_TYPE	SMS -when SMS is the bearer USSD -when USSD is the bearer GUI -when OSA Generic User interaction is the bearer
User location	USER_LOCATION_TYPE	
User status	USER_STATUS_TYPE	
Subscriber profile	SUBSCRIBER_PROFILE_TYPE	

All plug-ins extends the base module com.incomit.resources.defined in resource_common.idl.

The following modules are mapped to the service capabilities.

Table 2-2 Plug-in module and service capability

Module	Definition in	Corresponding Service capability
com.incomit.resources.callcontrol	CallControlResource_data.idl CallControlResource_IF.idl	Call Control
com.incomit.resources.charging	ChargingResource_data.idl ChargingResource_IF.idl	Charging
com.incomit.resources.messaging	messaging_mms_resource_if.idl messaging_resource_data.idl messaging_resource_if.idl	Messaging
com.incomit.resources.mm	MobilityResource_data.idl	User location and User Status
com.incomit.resources.mm.ul	UlResource_data.idl UlResource_IF.idl	User location
com.incomit.resources.mm.us	UsResource_data.idl UsResource_IF.idl	User status
com.incomit.resources.sp	sp_data.idl sp_interfaces.idl	Subscriber profile
com.incomit.resources.ui	UserInteractionResource_data.idl	Call user interaction and Messaging user interaction
com.incomit.resources.ui	UserInteractionCallResource_IF.idl	Call user interaction
com.incomit.resources.ui	UserInteractionResource_IF.idl	Messaging user interaction

All modules contains a set of interfaces to be implemented or used by the plug-in. Refer to the IDL definitions for the plug-in interfaces for detailed information. The IDL files are located in bea\wlng21\esdk\idl\plugin_if.

Plug-in states

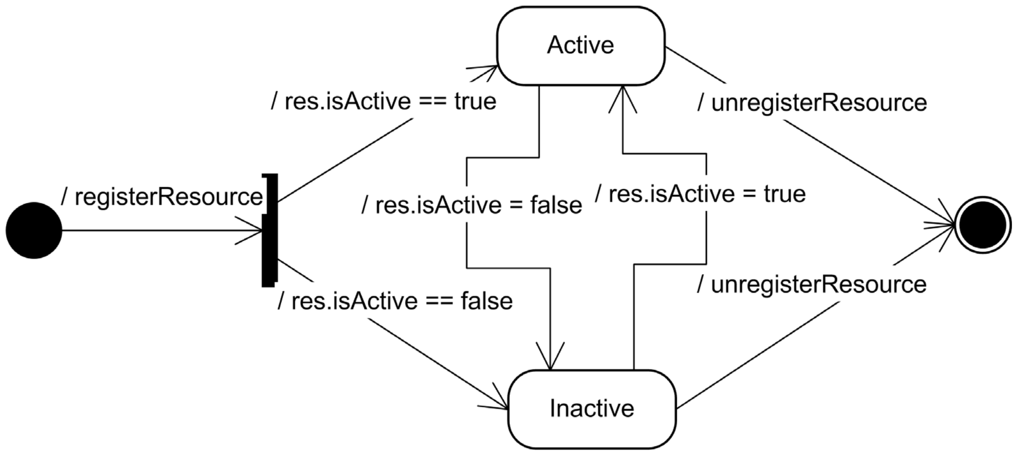
The following diagram shows the states of a plug-in as defined in the Plug-in Manager.

When the plug-in is in the Active state, the SCs are able to retrieve references to the plug-in and initiate new traffic. When the plug-in is in the Inactive state the SCs will not be able to retrieve any new plug-in references. The plug-in should set itself to inactive when it has lost contact with the underlying network node.

It is possible for the SCs to use existing references (for example call objects) to continue processing any active traffic.

The state of a plug-in does not prevent it from using the SCs to initiate network-triggered events. It only prevents the SCs to create application-initiated events.

Figure 2-1 Plug-in states



Different mechanisms are used when a plug-in changes between the active and inactive states. The state change from active to inactive is only checked when an SC requests to obtain a plug-in. In other words the plug-in have to make sure that the isActive method returns false when it is not active. When a plug-in moves from inactive state to active state it must explicitly invoke the resourceIsActive method. The plug-in must also make sure that the isActive method returns true when active.

Suspending a plug-in

It is possible to suspend a plug-in, which means that no new traffic will be sent to the plug-in. For example for call control this means that no new calls will be created, but all active calls will work as normal. This is for instance useful when making a graceful shutdown of a plug-in.

To suspend a plug-in it should simply return false when the isActive method is invoked. To resume from suspended state the plug-in should call resourceIsActive in the PluginRegistration interface. In order for a plug-in to be suspended it must implement ServiceDeploymentExt interface.

Switching plug-in

It is possible to use the states of a plug-in, see Plug-in states, to perform a HA switch when using an active-standby system. See details about this in section High availability.

Best practises for the plug-in interfaces

This chapter contains common details on the plug-in interfaces. Specific details for each service type (call control, user interaction, etc) are described in separate sections.

Usage of session id

To reduce the number of active CORBA objects, a plug-in may use the session id parameter. For example, when a new call is created (either network-triggered or application initiated) the plug-in sends a session id associated with that call. Whenever the proxy communicates with the plug-in it sends this id to the plug-in. This means that only one CORBA instance of the call object will be required. But in this case the plug-in must have some internal dispatching of invocations. Session id works in the same way with, for instance call legs of user interaction calls.

It is up to the developer of the plug-in to decide if session id or multiple objects are to be used. The call back interfaces that the SC implement are always newly created CORBA objects.

Always use timers (SLEETimer) to be able to detect and cleanup resources when no response is received. This is relevant for all layers, not only plug-in

Threading

For each method invocation that is made from the SC to the plug-in it is recommended that the plug-in creates a separate thread (or thread pool mechanism) that handles the invocation. This will allow the CORBA thread to return as soon as possible, which reduces the duration that the CORBA threads are blocked in the SC proxy. Most method calls are asynchronous (that is has a request and a response method), which makes this possible.

The SCs use a thread pool to implement this functionality, use the SLEE task manager as described in "SLEETaskManager" on page 3-13 and."SLEETask" on page 3-12.

Help classes for network plug-ins

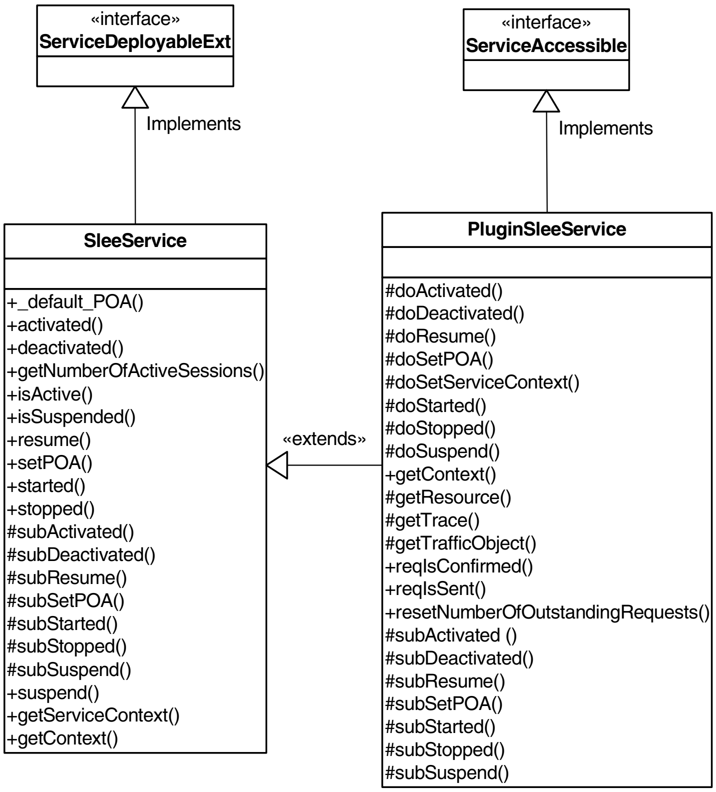
There is a set of help classes for plug-ins that simplifies the implementation. The template for the SLEE service part of a plug-in provided in the extension SDK module templates uses these classes.

The help classes for the network protocol plug-ins simplifies the interaction with the Service Deployable and Service Accessible interface by implementing the general parts of a SLEE Service in help classes.

In the package com.bea.wlcp.wlng.esdk there are two help classes for general SLEE Services. First there is an abstract help class, com.bea.wlcp.wlng.esdk.SleeService that implements the ServiceDeployableExt interface. The purpose of this class is to serve as a base class for general services executing in the SLEE. There is also an abstract help class, com.bea.wlcp.wlng.esdk.Context, that provides methods for getting and setting the POA and for getting the Service Context.

For network protocol plug-ins, there is an abstract base class, com.bea.wlcp.wlng.esdk.plugin.PluginSleeService, that implements the ServiceAccessible interface. It also implements the ServiceDeployableExt interface implicitly through the SLEEService class.

Figure 2-2 SleeService and PluginSleeService inheritance structure



PluginSleeService handles the registration of the plug-in in the plug-in manager. It extends the the abstract class com.bea.wlcp.wlng.esdk.SleeService. There is also an abstract class, com.bea.wlcp.wlng.esdk.plugin.PluginContext, that provides methods for getting and setting the resourceID, as well as declaring methods that should be implemented in the sub-context class.

When using these help classes, there is no need for the implementation classes for a plug-in to implement the ServiceAccessible and ServiceDeployableExt interfaces, instead they should extend the class PluginSleeService.

When a plug-in that extends the PluginSleeService class is started, the method doStarted(...) will be called.

When a plug-in that extends the PluginSleeService class is activated, the method doActivated(...) will be called. The help class PluginSleeService will then handle the registration of the plug-in in the plug-in manager.

When a plug-in that extends the PluginSleeService class is deactivated, the method doDeactivated(...) will be called.

The implementation of the traffic interfaces shall be implemented in a separate class. Whenever the method getTrafficObject() is called on the class the extends the PluginSleeService help class, the class that implements the traffic interfaces shall be instantiated, if the member-variable which holds the reference to the class that implements the traffic interface is not yet instantiated.

Plug-in manager

Accessing the Plug-in Manager

The Plug-in Manager is retrieved from the SLEEContext and a task is scheduled that will be performed in a separate thread when the plug-in manager has resolved a suitable plug-in based on the parameters provided. When scheduling the task, information on the type of plug-in to use, priority and so on are defined together with an object that implements the request that shall be forwarded to the plug-in. The object must implement SLEEResourceTask

In the example below this object is `task`. task implements the method dotask(...), and when the plug-in manager has retrieved a suitable plugin, the plug-in manager invokes the method doTask(...) on this object. The plug-in is provided as an argument to doTask(...). See MyServiceCapabilityManager_impl.java in \module_templates\espa_sc_module_impl\src\com\incomit\espa\my_espa_sc\ for an example.

Listing 2-1 Example of how to get a matching plug-in and to schedule a task

```
MyServiceCapabilityContext.getServiceContext().getSLEEContext().getResourceMan
ager().scheduleResourceTask(new TrProperty[0],
                                MyServiceCapabilityContext.PLUGIN_TYPE,

                                rAddress,

                                resourceContext,

                                0, // prio

                                MyServiceCapabilityContext.POLICY_SERVICE_GROUP,
                                1,
                                task,
                                MyServiceCapabilityContext.getServiceContext());
```

Plug-in manager interfaces

The following figure includes the interfaces related to the Plug-in Manager.

Figure 2-3 Plug-in manager interfaces

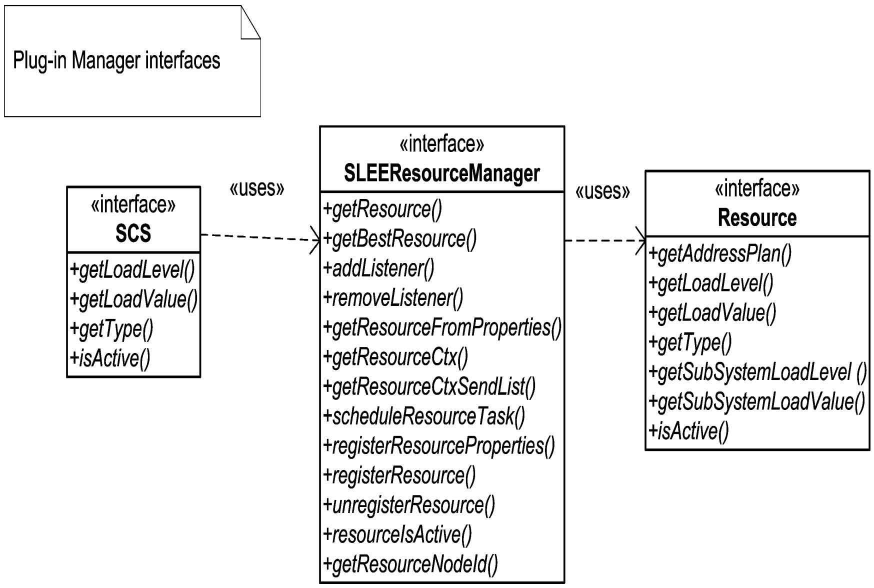


Table 2-3 Plug-in manager interfaces

Interface	Description
SLEEResourceManager	Initial object implemented by the Plug-in Manager.
Resource	Base interface that all plug-ins must implement.
ResourceListener	Call back interface used to notify that a plug-in has registered or de-registered itself in the Plug-in Manager
SCS	Implemented by the SCs. Closer descriptions can be found in SC manager.

SLEEResourceMgr

This interface is the initial object when accessing the Plug-in Manager.

Table 2-4 ResourceMgr

Method	Description
addListener	Adds a listener, interested in knowing if plug-ins have been added or removed.
removeListener	Removes a registered listener.
getBestResource	This method is used to get the resource having least load level which have been idle the longest time having the specified type and which have the route set up to handle this address and addresses belonging to the specified address plan. Use scheduleResourceTask instead.
getResourceFromProperties	Perform the same as getBestResource, with addition that only resources matching the specified properties are returned. Use scheduleResourceTask instead.
getResource	This method is used to get the resource having least load level which have been idle the longest time having the specified type and which have the route set up to handle this address and addresses belonging to the specified address plan. Use scheduleResourceTask instead.
getResourceCtx	Gets a plug-in.
getResourceCtxSendList	Gets a plug-in capable of handling sendlists

Table 2-4 ResourceMgr

Method	Description
scheduleResourceTask	Gets a plug-in and schedules a task to be performed. Asynchronous method for scheduling a resource task for executing a request towards a plug-in matching the specified criteria. If a plug-in can be allocated for the request the supplied resource task will be executed in a separate thread.
registerResourceProperties	Registers properties for a plug-in.
registerResource	Registers a plug-in in the plug-in manager.
unregisterResource	Unregisters a plug-in.
getResourceNodeId	Get the node ID for a plug-in.

Resource

The base interface that all plug-ins must implement. It is important that all CORBA objects that implement this interface are persistent, that is always use the same IOR. If this is not the case, plug-in routing information, as configured in the Plug-in Manager will not work. This interface is extended by Service Capability-specific parts, so there is one extension per SC type, call control, messaging, user location and so on.

Table 2-5 Resource

Method	Description
getAddressPlan	Get the supported address plans. For example E_164. The plug-in can support several address plans.
getLoadLevel	Get load level.
getLoadValue	Get load value in percent.
getType	Get type of plug-in. For example call control or messaging.
getSubSystemLoadLevel	Gets the load level of the underlying system for this resource.
getSubSystemLoadValue	Gets the load value of the underlying system for this resource.
isActive	Check if plug-in is active or not.

ResourceListener

The SCs can implement this interface. This listener interface will be notified when new plug-ins are registered in the Plug-in Manager. This will make it possible for the SC to add itself as a plug-in listener when new plug-ins are activated.

Table 2-6 ResourceListener

Method	Description
resourcesUpdated	Notify that a resource has been registered or de-registered.

Use cases

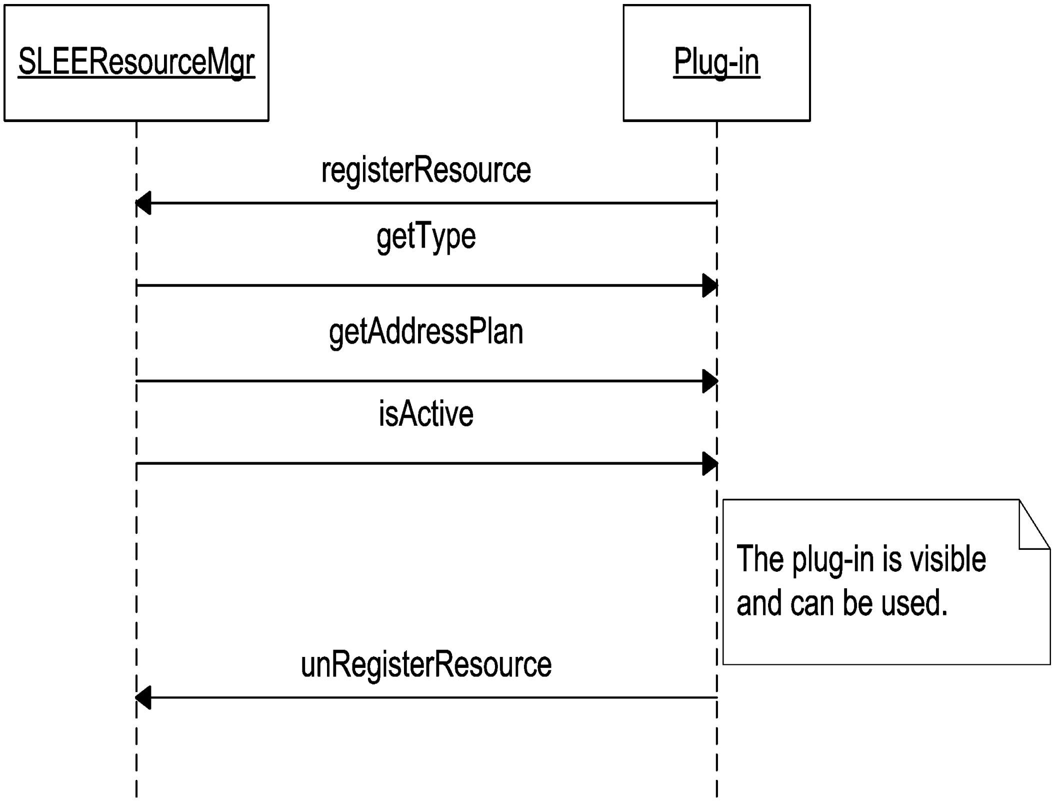
Registration and deregistration

The Resource interface is implemented by the plug-in.

If the plug-in returns false when the isActive method is invoked, it will not be accessible until it is activated.

If a plug-in supports more than one type it should use this registration process for each type.

Figure 2-4 Registration and de-registration of a plug-in

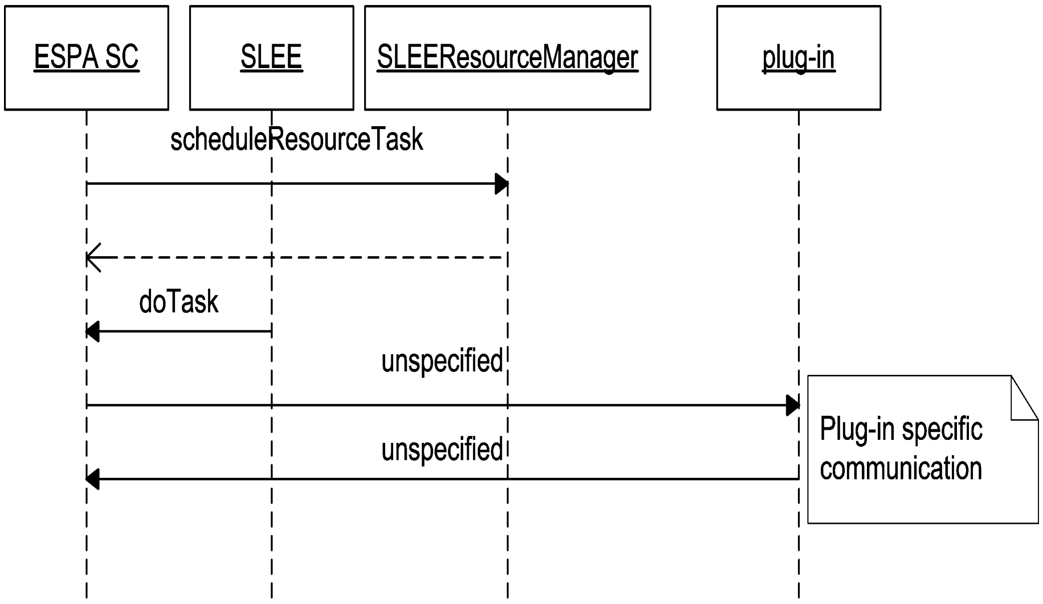


General usage (application-initiated events)

The following sequence diagram shows how application-initiated events are handled. An application-initiated event could for example be creation of a new call or sending an SMS.

The Plug-in Manager maintains a list of routing information for the plug-ins. In this list it is specified what address ranges a certain plug-in supports. The list is specified using the Management Tool.

Figure 2-5 General plug-in usage



Description of the sequence diagram:

- The SC asks the Plug-in Manager to schedule a task for the plug-in communication. This request will contain plug-in type, address plan, address, a set of properties and a class that the SLEE will perform doTask() on when a suitable plug-in is resolved. From this information the Plug-in Manager will locate all active plug-ins that matches with its internal routing information. Plug-in types, properties and address plans are defined in the plug-in ins. The properties is a name-value pair array used to correlate properties that the SC needs, and properties that the plug-ins supports. The following properties are used in the standard product:
  - NOTIFICATION_SUPPORT, whit the value TRUE or FALSE if notifications are supported by the plug-in.
  - SUBTYPE, with the values USSD or SMS. This is used to define if the messaging user interaction shall use USSD or plug-in.
- The Plug-in Manager will check if the plug-in is still active.
- The Load level of the plug-in will be requested. The plug-in must return its current load and if it is overloaded no new requests will be sent.

- The SLEE will invoke doTask() on the object provided when scheduling the task.
- Plug-in specific communication will commence.

SC manager

The SC Manager is similar to the Plug-in Manager in many ways. The difference is that it manages SCs instead of plug-ins. The main responsibility for the SC Manager is to deliver SC references to the plug-ins when dealing with network-triggered events. The SC Manager contains one part that is used by the SCs for registration an unregistration, and one part that is used by the plug-ins to obtain SCs.

Each SC that wants to be accessed by plug-ins needs to implement the SC base interface. The SC Manager supports registration and de-registration of SCs. The SCs are associated with a certain type that is used when locating the SCs. It is possible to create new SC types from the SLEE Manager.

In the case where several SLEEs are running, each instance will have its own SC Manager. In this case it does not matter which SC Manager is used, as they all share the same information.

It is not necessary for the plug-ins to use the SC Manager at all. It is the choice of the plug-in developer. The SC can register all available call back interfaces directly in the plug-in also. This means that the plug-in can handle the load balancing internally. If an SC is overloaded it will throw an overload exception and the plug-in should try another SC. This may be better for external, not executing in the SLEE, plug-ins for performance reasons as no extra CORBA invocations are needed. The use of the SC Manager is better suited for internal plug-ins as they interact with a pure java interface. It may on the other hand be more convenient to use the SC Manager. There is no need to use both these methods as the reference obtained from the SC Manager and the registered listener will be the same.

Accessing the SC Manager

The SC Manager is retrieved from the SLEEContext. See The SLEEContext is fetched from the ServiceContext, provided by the SLEE via the ServiceAccessible interface.

Listing 2-2 Getting the SC manager

```
m_scsManager = m_sc.getSLEEContext().getSCSManager();
```

Here, m_sc is the ServiceContext. The list of ESPA SCs are retrieved. In the example below all registered listeners of MESSAGING_TYPE are requested. Other ESPA SCs are requested using the same methodology, it is only the type that differs.

Listing 2-3 Example of how to get a list of ESPA Messaging SCs

```
scsArray = m_scsManager.getSLEESCSDiscovery().getSCSCtx(capabilityProperties,
                                                         SCS.MESSAGING_TYPE,
                                                         m_eventAny,
                                                         m_resourceID);
```

The code fragment also illustrates how a list of suitable SCs are fetched. The first in the list should be used since the SC manager provides load balancing, although any in the list can be used.

Interfaces

The figure below displays the interfaces related to the SC Manager.

Figure 2-6 SC Manager interfaces

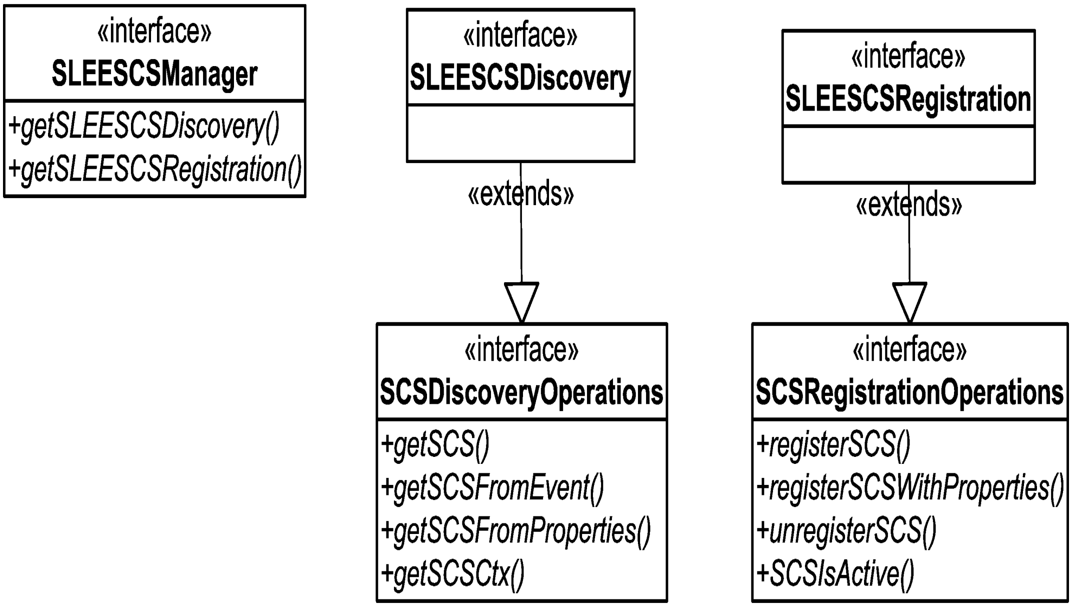


Table 2-7 SC manager interfaces

Interface	Description
SLEESCSMgr	Initial object implemented by the SC Manager.
SCSDiscovery	Interface that the plug-ins uses to obtain SC references.
SCSRegistration	Interface used to register an SC. This interface will not be closer described in this document.
SLEESCSDiscoveryOperations	This interface is implemented by the SCs proxies.
SCSRegistrationOperations	The base interface that all plug-ins must implement.

SLEESCSMgr

An object implementing the SCSMgr interface is the initial object in the SC Manager. This interface is used to access other SCSMgr instances and to retrieve the registration and discovery interfaces.

Table 2-8 SLEESCSMgr

Method	Description
getSCSRegistration	Retrieve the SCSRegistration object.
getSCSDiscovery	Retrieve the SCSDiscovery object.

SLEESCSRegistrationOperations

This is interface is used by the ESPA SC to registers into the SC manager.

Table 2-9 SLEESCSRegistrationOperations

Method	Description
registerSCS	Registers an SC. This method is to be used when only the type of the SC has significance.
registerSCSWithProperties	This method registers an SC using a set of properties. A property of type INSTANCE_ID and value the returned SCS id is always appended to the properties.
unregisterSCS	Unregisters an SC.
SCSIsActive	Called by SC to indicate that it is active.

SLEESCSDiscoveryOperations

This is interface is used by the plug-ins to retrieve references to SC instances. SCs retrieved with this interface will ensure that load balancing is applied to network-initiated traffic.

Table 2-10 SSLEECSDiscoveryOperations

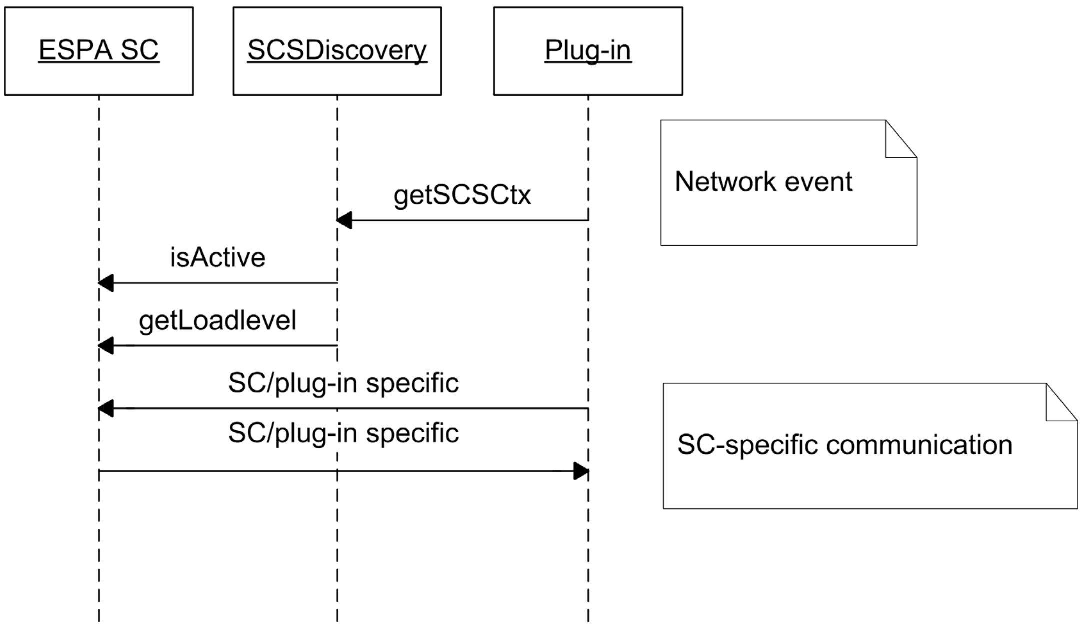
Method	Description
getSCS	Deprecated
getSCSCtx	Retrieve all SCs that have enabled a criteria matching the supplied parameters.
getSCSFromEvent	Deprecated
getSCSFromProperties	Deprecated

Use cases

Network-triggered event

The following sequence show how a network-triggered event is distributed to an SC using the SC manager.

Figure 2-7 Figure Network-triggered event



Service capability

The Service capabilities are responsible for implementing the Policy Enforcement Points, storing CDRs and to hold session information that is beyond the scope of the plug-ins.

Examples if this is user interaction sessions and call control sessions, which are of less transient nature than for example sending of SMSs or positioning requests.

The existing Service capabilities are stateful.

The service capabilities registers themselves with a certain type, that is used to bind a plug-in to a Service capability.

SESPA SC

The stateful adapters (or SESPA) are used to provide a stateless interfaces towards the service capabilities. SESPA presents a stateless interface to the Web Services implementation.

Per default there is a one to one mapping between Stateless adapters and Service capability modules.

If the standard Network Gatekeeper is extended with a Web Services or pure HTTP interface that fits on top of an existing Service Capability, the extended interface can be implemented on top of this interface.

In the standards product, the Parlay X implementation is using this interface.

Web Services interface implementation

The Web Services implementations are deployed in the Tomcat server embedded in the SLEE.

CHAPTER 3

Interacting with the SLEE and the SLEE Utility Services

This chapter describes how to interact with the SLEE and the utilities that the SLEE offers when creating applications.

These utilities can only be used when the applications are executing internally in the SLEE.

For a detailed description of all SLEE utility classes and methods, see the JavaDoc for the SLEE.

The following sections describe the SLEE and SLEE utility services:

- Basic SLEE interfaces
- SLEE utility interfaces
- SLEE utility classes
- OAM
- Using the database
- Using the alarm service
- Using the event service
- Using the charging service
- Using the time service
- Using the trace service

Basic SLEE interfaces

All services must implement the basic SLEE interfaces in order for them to executed in the SLEE, and being SLEE services.

ServiceAccessible

All accessible services running within the SLEE must have a class that implements this interface. The ServiceAccessible interface represents the object that will be installed in the SLEE service ORB as a service that can be accessed by other services. The class implementing this interface MUST also have an empty public constructor.

The ServiceContext object is provided by the SLEE using the ServiceAccessible interface.

The CORBA POA is supplied by the SLEE in setPOA(...), and the SLEE service must implement and return the same POA using the method public org.omg.PortableServer.POA _default_POA().

The SLEE service shall use this POA, or a child POA, when creating new CORBA objects.

The SLEE will call the following method on the object implementing this interface:

- setServiceContext(...) - Will be called by the SLEE to set the Service Context. An object representing the Service Context is provided. The Service Context is used for getting handles to other SLEE services, see "ServiceContext" on page 3-3.

ServiceDeployable

All services running within the SLEE must implement this interface, otherwise they can not be deployed. The SLEE will use the ServiceDeployable interface to notify the service of the current service deployment status.

The SLEE will call the following methods on the object implementing this interface:

- started() -will be called by SLEE when the service has been started.
- activated() -will be called by SLEE when the service has been activated.
- deactivated() -will be called by SLEE when a service is to be deactivated.
- stopped() -will be called by SLEE when a service is to be stopped.
- setServiceContext(...) - Will be called by the SLEE to set the Service Context. An object representing the Service Context is provided. The Service Context is used for getting handles to other SLEE services, see "ServiceContext" on page 3-3.

ServiceDeployableExt

Extension of ServiceDeployable interface that a service can implement if supporting suspend/resume for a service.

Extends the ServiceDeployable interface with the following operations:

- getNumberOfActiveSessions() -used for returning the number of active sessions the service holds.
- resume() -causes the service to resume its suspended state back to active state.
- suspend() -suspends the service.

ServiceManageable

All services running within the SLEE may have a class that implements this interface. The interface is used for providing extra management capabilities for a service. The ServiceManageable interface represents the object that will be installed in the name service as the OAM (Operation Administration and Management) object for a service. Note that the ServiceManageable interface extends ServiceCORBAServant. See "OAM" on page 3-16.

The SLEE will call the following method on the object implementing this interface:

- setServiceContext(...) - Will be called by the SLEE to set the Service Context. An object representing the ServiceContext is provided. The ServiceContext is used for getting handles to other SLEE services, see "ServiceContext" on page 3-3.

ServiceContext

The ServiceContext represents the context of a service. All service interfaces have a setServiceContext method that will be called from the SLEE before calling the method started() on the object implementing ServiceDeployable interface.

It is possible to get a handle to the following services through this interface:

- Charging service, see "ChargingService".
- Policy service, see "PolicyManager".
- SLEE Event Channel service, see "SLEEEventChannel".
- TraceLogService service, see "TraceLogService".
- ServiceInstanceHandler, see "ServiceInstanceHandler".

It is also possible to get references to the objects in the SLEE service implementing the following interfaces:

- ServiceAccessible object implemented by the SLEE service, see "ServiceAccessible".
- ServiceDeployable object implemented by the SLEE service, see "ServiceDeployable".
- ServiceManageable object implemented by the SLEE service, see "ServiceManageable".

It is also possible to get a handle to the SLEE Context service, which is used to get other utility services. See "Services fetched from the SLEEContext".

Via the ServiceContext it is also possible to:

- get the name of the jar file in which this service was installed.
- get the name of the service.
- get the state for the service

SLEEContext

SLEEContext represents the SLEE context for a service. It provides an initial object for retrieving other services. A SLEE Context object is provided by the SLEE via the Service Context object. A service can use the SLEE Context to retrieve handles to utility services provided by the SLEE.

It is possible to get a handle to the following classes through this interface:

- Alarm, see "Using the alarm service".
- Cyclic ID manager, see "SLEECyclicIDManager".
- Database Manager, see "SLEEDBManager".
- Event Log, see "Using the event service".
- Global Counter and the Global Counter Manager, see SLEEGlobalCounter and SLEEGlobalCounterManager.
- ID Manager, see "SLEEIDManager".
- Load Manager, see "SLEELoadManager".
- Plug-in Manager, see "SLEEResourceManager"
- SC Manager, see "SC manager" on page 2-17.

- Statistics Manager, see "SLEEStatisticsManager".
- Task Manager, see "SLEETaskManager".
- Time Manager, see "SLEETimeManager".
- Zombie Object Supervisor, see "SLEEZombieObjectSupervisor".

SLEE utility interfaces

A software module that executes in the SLEE utilizes a set of operations offered by the SLEE and the SLEE Utility services. Which SLEE utility interfaces to use depends on which SLEE utility services the software module needs.

Some services are available using the SLEE Context interface, see "Services fetched from the SLEEContext", others via the "Services fetched from the SLEEContext" interface. There are also a set of classes that can be used directly, see "SLEE utility classes".

Services fetched from the ServiceContext

ChargingService

The charging service provides methods for creating CDRs in the Network Gatekeeper database.

See "Using the charging service".

PolicyManager

The Policy Manager service provides methods for implementing a policy enforcement point.

See "Implement the Policy Enforcement Point".

SLEEEventChannel

Interface for broadcasting events and for register listeners of events originating from other SLEE services. Used for sending events (data) from a SLEE service to another, convenient when for example propagating changes of cached data to other instances of a service.

Below is pseudo code for broadcasting an event via the SLEEEventChannel.

Listing 3-1 Broadcasting an event

```
//m_sc holds the ServiceContext
```

```
String eventId = "eventID";
EventData eventData = new EventData();
org.omg.CORBA.Any event;
event = eventData;
m_sc.getSleeEventChannel().generateEvent(eventId, event);
```

Below is pseudo code for receiving the event.

Listing 3-2 Receiving an event -class that implements SLEEEventChannelListener

```
public void processEvent(String eventId,
                         org.omg.CORBA.Any event,
                         String source) {
   if (eventId == "eventID") { // Check the event ID
      EventData eventData = (EventData)event; //Cast to proper class
   }
   // source contains the instance ID of the SLEE service.
   // m_sc holds holds the ServiceContext
   if (source.equals(m_sc.getInstanceName()) {
      // The event originated from this instance.
   }
}
```

TraceLogService

Interface for the trace log service. A service can use the trace service to write service tracing information to a service specific trace file. Although the trace service can be disabled and enabled it is recommended that each service performs a check on the trace active flag before calling the logTrace method for performance reasons (reduces the number of allocated String objects at runtime). See "Using the trace service".

ServiceInstanceHandler

The Service instance handler can be used to locate multiple instances of a service. It provides IORs to other SLEE Services ServiceAccessible interface.

PolicyManager

The Policy Manager is used for retrieving the Policy service in order to create a Policy Enforcement Point. The policy manager is normally used in the ESPA SC Layer.

Below is an outline of how a Policy Enforcement Point is implemented. See MyServiceCapabilityManager_impl.java in module_templates\espa_sc_impl\src\com\acompany\espa\mysctype\ for details on how it is implemented.

The PolicyManager is fetched from the ServiceContext.

Listing 3-3 Fetching the PolicyManager

```
policyManager =
MyServiceCapabilityContext.getServiceContext().getPolicyManager();
```

Then a PolicyRequest object, that holds the data to be forwarded to the Policy Decision Point, is created. Among the data is application and service provider IDs for the application that the requests originates from.

Listing 3-4 Creating a PolicyRequest object

```
MyServiceCapabilityContext.getServiceContext().getSLEEContext().getTimeManager
().getTime();

     PolicyRequest policyRequest = new PolicyRequest_impl(

     MyServiceCapabilityContext.getServiceContext().getSLEEContext().getORB(),

     timeStamp);
```

```
    try {
        policyRequest.applicationID = m_ApplicationID;
        policyRequest.serviceProviderID = m_ServiceProviderID;
    } catch (com.incomit.espa.access.AccessException e) {
        ...
   }
    policyRequest.serviceName =
    MyServiceCapabilityContext.POLICY_SERVICE_NAME;
    policyRequest.serviceGroup =
    MyServiceCapabilityContext.POLICY_SERVICE_GROUP;
    policyRequest.methodName = "myMethod";
    policyRequest.serviceCode = serviceCode;
    policyRequest.requesterID = requesterID;
    policyRequest.transactionID = -1;
    policyRequest.noOfActiveSessions = -1;
    policyRequest.reqCounter = 0;
    AdditionalData adArray[] = new AdditionalData[2];
    AdditionalDataValue targetAddressValue = new AdditionalDataValue();
    AdditionalData adTargetAddressString = new AdditionalData();
    targetAddressValue.stringValue(address);
    adTargetAddressString.dataName = "targetAddress";
    adTargetAddressString.dataValue = targetAddressValue;
    adArray[0] = adTargetAddressString;
    AdditionalDataValue dataValue = new AdditionalDataValue();
    AdditionalData adDataString = new AdditionalData();
    dataValue.stringValue(data);
    adDataString.dataName = "data";
```

```
            adDataString.dataValue = dataValue;

            adArray[1] = adDataString;

            policyRequest.additionalParameters = adArray;
```

Then the PolicyRequest data is evaluated by the Policy decision Point.

Listing 3-5 Evaluate the Policy data

```
PolicyRequest modifiedRequest =
(PolicyRequest)policyManager.evaluate((PolicyRequest) policyRequest);
```

If the request is denied, a DenyException is thrown. If the request was accepted, the request data may have been modified, so the modified data is used when propagating the request instead of the original data.

Listing 3-6 Use the request data returned from the PolicyRequest

```
requestContext = new RequestContext(modifiedRequest.requesterID,

                                    modifiedRequest.serviceCode);

address = modifiedRequest.getAdditionalDataStringValue("targetAddress");

data = modifiedRequest.getAdditionalDataStringValue("data");
```

If the request was denied, a CDR shall be created with completion status Policy Rejected.

Services fetched from the SLEEContext

SLEEDBManager

Database manager for SLEE. The DB manager controls connections to the database. A SLEE service can use the DB manager to retrieve a DB connection. By default all users/slee services share the same database and database user identity. See "Using the database".

All SLEE services that use JDBC shall retrieve connections using the SLEEDBManager instead of using the driver directly

SLEELoadManager

Used to query the load of a SLEE. It reports the load as a number between 0 and 100. It is also possible to register listeners that report when the load has reach a certain level. Note that the load is reported to listener only every 5th second. It is recommended to use the synchronous method instead of registering listeners.

Listing 3-7 Getting the load on a SLEE Service

```
//m_sc holds the ServiceContext

int loadValue _= m_sc.getSLEEContext().getLoadManager().getLoadValue();
```

SLEELoadManagerListener

Load events are reported using this interface, when subscribed to using the SLEE LoadManager.

Events related to the load has reached a load level are reported via the SLEELoadManagerListener interface via the method processLoadEvent(LoadEvent loadEvent). The LoadEvent class holds information on which load level that was reached:

- NORMAL
- OVERLOADED
- SEVERE_OVERLOADED

The registration of the SLEELoadManagerListener interface is performed via the method addListener(...) in the SLEELoadManager interface.

SLEEResourceListener

When plug-ins are added, it is reported on this interface.

SLEEResourceManager

The Plug-in manager interface. See "Plug-in manager interfaces" on page 2-11.

SLEEResourceTask

This interface must be implemented by users of the SLEEResourceManager.scheduleResourceTask method. A resource task is scheduled by calling scheduleResourceTask with an instance to the class implementing SLEEResourceTask as an argument to the SLEEResourceManager.

This SLEEResourceTask interface is implemented by ESPA SCs, and the SLEE calls the method doTask(...) on the implementation of the interface. A reference to the allocated plug-in is provided as an argument to doTask(...).

SLEEStatisticsManager

This interface is used to add transaction statistics to the Statistics SLEE service. There are a set of pre-defined statistic types, and new ones can be added using this interface or by using OAM in the SLEE Statistics Service. All new Service Capabilities shall generate transaction statistics to allow for a transaction based price model. Examples on how to generate transactions statistics are provided in the template source code.

For extensions the statistics transaction type range must be: 1000 to 10 000.

Transaction statistics can be added for all traffic generated by all applications, or per service provider or per service provider and application.

serviceProviderID is put in the sourceEntity parameter and applicationID is put in the sourceApplication parameter.

Below is an example on how to generate statistics.

Listing 3-8 Update statistics counter

```
// m_sc holds the SLEEContext
// serviceProviderID contains the service provider ID that performed the request
// applicationID contains the application ID that performed the request
int transactionType = TRANSACTION_TYPE_NETWORK_TRIGGERED_EVENT_SUCCESS;
int transactions = 1; //number of transactions to add to the statistics counter
m_sc.getStatisticsManager().addTransactionStatistics(transactionType,
                                                     transactions,
                                                     serviceProviderID,
                                                     applicationID);
```

The transactionType must be added to the SLEE Statistics service using the Management Tool. That is, the value representing transactionType must be added using the Management method **addStatisticType** in the **SLEE_statistics** service.

SLEETask

SLEETasks shall be used instead of using Java threads directly. Java threads must never be created directly, they should always be created via SLEETasks. Since asynchronous communication between the software modules always is preferred, SLEETasks shall be used for remote procedure calls via CORBA.

The SLEETask interface should be implemented by users of the SLEETaskManager. A task is scheduled by calling scheduleSLEETask in the SLEETaskManager. As soon as one of the threads managed by SLEETaskManager becomes available doTask in SLEETask will be called.

SLEETasks shall be used instead of pure Java tasks. The SLEETaskManager provides mechanisms for pooling of threads. Use SLEETasks for processing of tasks that risks blocking of resources.

Listing 3-9 Pseudo code for a SLEETask

```
public class MyTask implements SLEETask {
   public MyTask() {
   }
   // Will be called by the SLEETaskManager
   public void doTask() {
            // Perform the task.
   }
}
```

SLEETaskManager

SLEE task manager should be used for scheduling a task by using a thread from the thread pool managed by the SLEETaskManager.

Listing 3-10 Pseudo code for scheduling a SLEETask

```
// m_sc holds the ServiceContext

m_taskMgr = m_sc.getSLEEContext().getTaskManager();

m_taskMgr.scheduleSLEETask(new MyTask)
```

SLEETimeManager

SLEE Time manager is responsible for managing system time and timer handling. Its possible to schedule timers and get system time.

SLEEZombieObjectListener

Listener interface for handling state changes of zombie objects. When the zombie object is reachable again the client listener will be notified. If the supervised object is not alive within AssumedDeadTimeout the listener will also be notified.

Instead of using this interface, use the SuperVisedListListener interface, see "SupervisedListListener" on page 3-16.

SLEEZombieObjectSupervisor

Class for supervising CORBA objects that are currently not reachable but may become available in the future. Clients can use this utility class to handle the checking of when such an object becomes active/reachable again. When the object is reachable the client listener will be notified. If the supervised object is not alive within the specified time-out (or getAssumeDeadTimeout() if time-out is not specified) the listener will also be notified.

Instead of using this class, use the SupervisedList class, see "SupervisedList" on page 3-15.

TaskChain

Used when tasks need to be executed in a given order.

This interface is returned when scheduling the first task in a task chain using SLEETaskManager.scheduleTaskChain. It can be used to add new tasks to the end of the chain.

SLEE utility classes

In com.incomit.slee there is a set of classes available. Below is a summary of the available classes. Refer to the Javadoc for the SLEE for detailed information.

Alarm

See "Using the alarm service".

LoadEvent

Definition of a load level event.

SLEECyclicIDManager

ID manager for retrieving a unique ID. The IDs will be unique for all SLEE instances using the same database.

IDs will be reused when IDs of an int minus the most significant byte has been exceeded.

Listing 3-11 Below is pseudo code for how to get a unique ID

```
//m_sc holds the ServiceContext

int ID = m_sc.getSLEEContext().getCyclicIDManager().getID()
```

SLEEDBTable

Database table class. Can be used to check if table exists and that it has the desired format and to create or replace the table. Used also for creating temporary tables.

SLEE services using the SLEE DB Manager to retrieve JDBC connections must use this class to create its tables, to get privileges to use the table. See "Using the database" on page 3-18.

SLEECyclicIDManager

ID manager for retrieving a unique ID. The IDs will be unique for all SLEE instances using the same database. The IDs are of type int.

IDs will be reused when IDs of a int minus the most significant byte has been exceeded.

SLEEIDManager

ID manager for retrieving a unique ID. The IDs will be unique for all SLEE instances using the same database. The IDs are of type long.

IDs will be reused when IDs of a long minus the most significant byte has been exceeded.

SLEEGlobalCounterManager

The SLEE global counter manager provides an interface for maintaining global counters. A global counter will be automatically updated across all SLEE instances in a system.

SLEEGlobalCounter

The SLEE global counter interface provides methods for incrementing and managing a global counter. Note that increments of a counter may not be distributed to all instances immediately. Updates will be performed globally every 10th increment of a specific counter or every 5th second for all counters that have been updated less than 10 times during the last 5 second period.

SupervisedList

List holding supervised CORBA objects, that if they are unreachable is moved to a zombie object supervisor list where they are "pinged" for a certain time before they are considered dead and entirely removed from the list.

Listing 3-12 Pseudo code for creating a supervised list.

```
// m_sc holds the ServiceContext.

//listener implements the SupervisedListListener interface

SupervisedList m_supervisedList;

long timeBeforeDead = 10000; // Time given in millseconds
```

```
m_supervisedList = new SupervisedList(m_sc.getSLEEContext(),
timeBeforeDead);
m_supervisedList.addListener(listener);
```

SupervisedListListener

Interface for receiving events about objects removed from a SupervisedList.

Listing 3-13 Pseudo code for listening to events from object s in a supervised list.

```
public class MyClass implements SupervisedListListener {
   MyClass() {
   }
   public void objectDead(SupervisedCorbaObject o) {
      // Object was removed due to zombie timeout.
      // Remove any references to the object.
   }
   public void objectZombie(SupervisedCorbaObject o) {
      // Object was declared a zombie and moved to zombie list.
   }
   public void objectLive(SupervisedCorbaObject o) {
      // Object was declared alive and put back into the list again
   }
}
```

OAM

All SLEE services that has parameters that needs to be configured in runtime must implement the ServiceManageable interface.

The OAM operations are defined in IDL, and Java stubs are generated from this interface. This means that the class that implements the OAM operations must extend the abstract class generated from the IDL definition. This class is named <name of interface in IDL>POA.

For example if the interface is named AServiceOAM, the implementation must extend AServiceOAMPOA. In the example below the implementation of the interface is in the class AServiceOAM_impl.

Listing 3-14 Example of declaration of the OAM class

```
public class AServiceOAM_impl extends AServiceOAMPOA implements
ServiceManageable
```

Also, the IDL file defining the OAM interface must be packaged into the jar file for the software module, in the root. There is a deployment descriptor in JAR file, which states the name of the IDL file for the OAM interface and the class that implements the interface.

Listing 3-15 Part of the Example of deployment descriptor

```
...

<SERVICE_MANAGEABLE>

        com.my_company.my_plugin.AServiceOAM_impl

        <SERVICE_MANAGEABLE_IDL>

            AServiceOAM.idl

        </SERVICE_MANAGEABLE_IDL>

</SERVICE_MANAGEABLE>

...
```

Additional information is also defined in the deployment descriptor. The build files in the templates are used for creating the deployment descriptors and to package them into the deployable Jar file.

Implementing OAM access control

To implement access control for OAM, there must be two methods for each OAM method. One that is exposed in the OAM interface, and another one that checks if the OAM user has appropriate privileges. The two methods are correlated by their name. So an OAM method with the signature <method name>, must have a corresponding method with the signature <method name>Allowed(...). In the latter method the privilege is checked. See example below.

Listing 3-16 Checking of an OAM method is permitted

```
public void setOverloadPercentage(int percentage)

....


public boolean setOverloadPercentageAllowed(int userLevel) {

       return(userLevel >= SLEEOAM.SEC_LEVEL_READ_WRITE);

    }
```

Using the database

The SLEE uses a DBMS that can be used by the application programmer. Support is built into the framework for accessing the database using JDBC.

The SLEE allocates database connections that can be accessed from the SLEE Context. From the SLEE Context you can retrieve a handle to the Database Manager, which is a singleton class that controls access to the database. By default all users and SLEE services share the same database. To get access to own tables, the tables must be created using the `createSLEEDBTable()` method. That is, the tables cannot be created using SQL syntax.

This example shows how to create a temporary table, called `mobile_users`, with two columns containing a user identity and its corresponding mobile subscription number. The table can be used to, for example, sending SMS messages to a group of registered users. The `createTemporaryTable()` method is called on service activation. The insertion of data through the `register()` method can be done from a class implementing the `ServiceAccessible` interface. To retrieve a list of all users that are currently registered, a convenience method called `getSubscribers()` is used. Finally, when the service is deactivated, the table is removed from the database by calling the `removeTemporaryTable()` method.

For the purpose of the example, we will create a class called `DbRegister.java`, that takes one argument in its constructor, a reference to the SLEE Service Context. The database manipulation is done through the database manager, `SLEEDBManager`.

Listing 3-17 DbRegister.java -initiation and constructor

```
package example.helloslee;
import com.incomit.slee.*;
import java.sql.*;
import java.sql.*;
public class DbRegister {
private ServiceContext itsServiceContext = null;
  private SLEEDBManager itsDbManager = null;
// Constructor
  public DbRegister (ServiceContext aServiceContext) {
    itsServiceContext = aServiceContext;
    SLEEContext sc = itsServiceContext.getSLEEContext();
    itsDbManager = sc.getDBManager();
}
```

To create the temporary table, we will use the method `createTemporaryTable()`, that takes no arguments. It uses the utility method `createSLEEDBTable()` that returns a `SLEEDBTable`, which then is used to add the necessary columns, before physically creating the table in the database. The arguments to `addColumn` specifies the column name, the data type, if the column is used in the primary key, and if `null` values are allowed.

Listing 3-18 DbRegister.java -method: createTemporaryTable

```
public void createTemporaryTable () {
  try {
     Connection conn =
            itsDbManager.getConnection();
     try {
            SLEEDBTable table =
            itsDbManager.createSLEEDBTable("mobile_users");
            table.addColumn("userName", "VARCHAR(20)", true, false);
            table.addColumn("msisdn", "VARCHAR(28)", false, true);
            if (!table.exists(conn)) {
                   table.create(conn);
            }
     }
     catch(Exception e){
            // Handle exception
     }
     finally {
            if (conn!=null) {
                   conn.close();
            }
     }
  } catch (SQLException e) {
    // Handle exception
  }
}
```

To insert a new row in the table, we use the method `register()`, that takes a user name and a mobile subscription number as arguments. It uses JDBC methods to create a statement and execute the update towards the database. It is working towards a pre-allocated connection resource that is served by the SLEE Database Manager.

Listing 3-19 DbRegister.java -method: register

```
 public void register(String user, String msisdn) {
  try {
      Connection conn =
      itsDbManager.getConnection();
      try {
             Statement stmt = conn.createStatement();
             try {
                    stmt.executeUpdate("INSERT INTO " +
                                       "mobile_users VALUES ('" +
                                        user + "', '" + msisdn + "')");
             catch (Exception e) {
                    //Handle exception
             }
             finally {
                    stmt.close();
             }
      catch (Exception e) {
                    //Handle exception
      }
      finally {
             if (conn!=null) {
                    conn.close();
```

```
                }
            }
        } catch (SQLException ex) {
            // Handle exception
        }
    }
```

As a convenience service, we provide a method that return all registered subscribers. The `get-Subscribers()` methods takes no arguments, and executes a SQL `SELECT` query towards the database. The Result Set is converted to a `Hashtable` and returned to the calling method.

Listing 3-20 DbRegister.java -method: getSubscribers

```
  public Hashtable getSubscribers () {
      Connection conn = itsDbManager.getConnection();
      Statement stmt;
      ResultSet rs;

       try {
              stmt = conn.createStatement();
              Hashtable table = new Hashtable();
              String query = "SELECT userName, msisdn " +
                             "FROM mobile_users";
              rs = stmt.executeQuery(query);
              while (rs.next()) {
                     table.put(rs.getString(1), rs.getString(2));
              }
              return table;
```

```
        }
        catch (SQLException e) {
        // Handle exception
        }
        finally {
        if (rs!=null) {
               rs.close();
        }
        if (stmt!=null){
               stmt.close();
        }
        if (conn!=null) {
               conn.close();
        }
    }
  }
```

Finally, when the table is no longer needed, it is removed from the database. For this the method `removeTemporaryTable()` is used. It also uses a connection resource from the SLEE Database Manager, and a SQL `DROP TABLE` command in its JDBC execute update statement.

Listing 3-21 DbRegister.java -method: removeTemporaryTable

```
  public void removeTemporaryTable () {
    try {
        Connection conn =
        itsDbManager.getConnection();
        Statement stmt;
        try {
               stmt = conn.createStatement();
               stmt.executeUpdate("DROP TABLE mobile_users");
        }
        catch (Exeption e) {
               // Handle exception
        }
        finally {
               if (stmt!=null) {
                      stmt.close();
               }
               if (conn!=null) {
                      conn.close();
               }
        }
     } catch (SQLException e) {
       // Handle exception
     }
  }
}
```

Using the alarm service

Alarms in the SLEE are handled by a dedicated alarm service. All alarms reported from an application will be reported to an alarm log in the database. There is only one table in the database for logging alarms, which means that all applications using the alarm service share the same table in the database.

Alarms are critical events that need corrective action. If an application raises too many alarms, it is taken out of service. The maximum number of allowed critical alarms for an application can be configured in the deployment descriptor.

In this example we will explore how to set up a reference to the alarm service, and how to generate alarms that will be stored in the alarm log.
An alarm record contains the following information:

Field	**Description**
Alarm Number	An alarm identifier, for example the type of the alarm. For extensions, the alarm number range must be: 500 000 to 999 999.

Severity	The severity of the alarm. Pre-defined constants are defined in the `AlarmService` interface, and should be used:
	`AlarmService.MINOR`
	`AlarmService.WARNING`
	`AlarmService.MAJOR`
	`AlarmService.CRITICAL`

Service Instance Name	The source of the alarm, that is, the unique name for this service. Retrieved by using `getInstanceName()` on the `ServiceContext` object.
Time and Date	The time-stamp will be generated by the system when receiving an alarm
Additional Information	Any other information that can be used to inform about the alarm. For example the stack trace can be included, if available.

The method `fireAlarm()` is used to report an alarm, and has the following signature:

```
public void fireAlarm(byte[] source, int severity, int identifier, byte[]
info)
```

The following code shows an example how to set up and use the alarm service. It is in the form of an utility class that uses the Service Context to retrieve a handle to the alarm service. The method `raiseAlarm()` fires a simple alarm that will be written to the system alarm table in the database. The method takes one argument of the type `Throwable`, and will include the stringified error message in the alarm log.

Listing 3-22 Alarm.java

```
package example.helloslee;

import com.incomit.slee.*;
import com.incomit.slee.alarm.*;

public class Alarm {

  private ServiceContext itsServiceContext;
  private AlarmService itsAlarmService;

  private int alarmNumber = 4711;

  public Alarm (ServiceContext aServiceContext) {
    itsServiceContext = aServiceContext;
    SLEEContext sc = itsServiceContext.getSLEEContext();
    itsAlarmService =  sc.getAlarmService();
  }

  public void raiseAlarm (Throwable e) {
    itsAlarmService.fireAlarm(itsServiceContext
                               .getInstanceName().getBytes(),
                              AlarmService.MAJOR,
                              alarmNumber,
                                    e.toString().getBytes());
  }
}
```

Using the event service

Events in the SLEE are handled by a dedicated Event Log service. All events generated by an application will be logged to an event log in the database. There is only one table in the database for logging events, which means that all applications using the Event Log Service share the same table in the database.

Events are expected events of importance to the operator.

In this example we will explore how to set up a reference to the Event Log Service, and how to generate events that will be stored in the event log.

An event record contains the following information:

Field	Description
Event Number	An event identifier, for example the type of the event For extensions, the event number range must be: 500 000 to 999 999.

Field	Description

Level	The level of the event. Pre-defined constants are defined in the `EventLogService` interface, and should be used:
	`EventLogService.LOW`
	`EventLogService.MEDIUM`
	`EventLogService.HIGH`

Service Instance Name	The source of the event, that is, the unique name for this service. Retrieved by using `getInstanceName()` on the `ServiceContext` object.
Time and Date	The time-stamp will be generated by the system when receiving an event
Additional Information	Any other information that can be used to inform about the event.

The method `logEvent()` is used to report an event, and has the following signature:

```
public void logEvent(byte[] source,int identifier,int level,byte[] info)
```

The following code shows an example how to set up and use the Event Log Service. It is in the form of an utility class that uses the Service Context to retrieve a handle to the Event Log Service. The method `storeEvent()` logs a simple event that will be written to the system event log table in the database.

Listing 3-23 Event.java

```
package example.helloslee;


import com.incomit.slee.*;

import com.incomit.slee.event.*;


public class Event {
  private ServiceContext itsServiceContext;
  private EventLogService itsEventService;

  private int eventNumber = 42;
  private String message = "Logged message";

  public Event (ServiceContext aServiceContext) {
    itsServiceContext = aServiceContext;
    SLEEContext sc = itsServiceContext.getSLEEContext();
    itsEventService =  sc.getEventLogService();
  }

  public void storeEvent () {
    itsEventService.logEvent(itsServiceContext
                              .getInstanceName().getBytes(),
                             EventLogService.MEDIUM,
                             eventNumber,
                             message.getBytes());
  }
}
```

Using the charging service

The charging service works in a similar way as the event service, the difference is the information that will be stored in the database. Each SLEE service will have a dedicated instance of the charging service. The charging service is retrieved via the Service Context.

The signature of the `logChargingInfo()` method looks like this:

```
public void logChargingInfo(ChargingInfo info)
```

The `ChargingInfo` class is a container with fields and corresponding `set`-methods for all charging-specific data. Before applying the `logChargingInfo()` method, all mandatory parameters must be set. Parameters that are not set will cause a `null` value to be written to the Charging Log. The methods available for the `ChargingInfo` class are shown in the table below. All fields of type `long` that represents a time is in milliseconds. A start or stop time is represented as the number of milliseconds since January 1, 1970 00:00:00.000 GMT.

Method	Parameter	Description
`addAdditionalInfo`	`(String xmlTag, String xmlValue)`	Adds user defined additional charging parameters. These parameters will be saved in the `additional_info` field of the charging database as XML elements. This method can be called more than once to add more parameters.
`clear`	`()`	Clears all data and restores the default values in the ChargingInfo instance
`setAmountOfUsage`	`(long amountOfUsage)`	Sets the used amount. Used when the charging is not time dependent, for example, flat rate services.
`setCompletionStatus`	`(int status)`	Sets transaction completion status. Indicates if the transaction was completed or not. If the transaction is divided into parts, the completion status also indicates if all transaction parts have been sent.The class defines constants that should be used: `ChargingInfo.COMPLETION_STATUS_COMPLETED` `ChargingInfo.COMPLETION_STATUS_PARTIAL` `ChargingInfo.COMPLETION_STATUS_FAILED` `ChargingInfo.COMPLETION_STATUS_POLICY_REJECTED`

Method	Parameter	Description
`setConnectTime`	`(long connectTime)`	Sets a timestamp telling when the destination party responded
`setDestinationParty`	`(String destinationParty)`	Sets the destination party's address
`setDurationOfUsage`	`(long durationOfUsage)`	Sets the total time the service used network resources
`setEndOfUsage`	`(long endOfUsage)`	Sets a timestamp telling when the service stopped using network resources
`setOriginatingParty`	`(String originatingParty)`	Sets the originating party's address
`setServiceName`	`(String serviceName)`	Sets the name of the used service
`setSessionID`	`(long sessionId)`	Sets the session ID. The session ID is the connection between related charging transactions.
`setTransactionPartNumber`	`(int transactionPartNumber)`	Sets the transaction part number. Used if the transaction is divided into different parts. Increment the number by one for each transaction part.
`setStartOfUsage`	`(long startOfUsage)`	Sets a timestamp telling when the service started using network resources
`setUserID`	`(String userId)`	Sets the ID of the application that has used the service

The following example will show how to obtain a reference to the charging service and create a Charging Data Record in the charging table in the database. The class is in the form of a helper, that is created at the start of service usage for a specific subscriber. When the subscriber stops using the service, the `createChargingRecord()` method is called to produce the charging record. The class uses the `timeManager` utility service from the `com.incomit.time package` to retrieve the system time at start and stop of service usage.

Listing 3-24 ChargingHelper.java

```
package example.helloslee;


import com.incomit.slee.*;

import com.incomit.slee.time.*;

import com.incomit.slee.charging.*;


public class ChargingHelper {


  private ServiceContext itsServiceContext = null;

  private ChargingService itsChargingService = null;

  private SLEETimeManager itsTimeManager = null;

  private ChargingInfo itsInfo;


  public ChargingHelper (ServiceContext aServiceContext,

                         int aSessionId,

                         String aUserId) {

    itsServiceContext = aServiceContext;

    itsChargingService = itsServiceContext.getChargingService();

    SLEEContext sc = itsServiceContext.getSLEEContext();

    itsTimeManager = sc.getTimeManager();

    itsInfo = itsChargingService.createChargingInfo();

    itsInfo.setSessionID(aSessionId);

    itsInfo.setServiceName(itsServiceContext.getName());
    itsInfo.setUserID(aUserId);
    itsInfo.setStartOfUsage(itsTimeManager.getTime());
    itsInfo.addAdditionalInfo("additionalinfo", "testvalue");
  }

  public void createChargingRecord () {
    itsInfo.setEndOfUsage(itsTimeManager.getTime());
    itsInfo.setCompletionStatus(ChargingInfo
                              .COMPLETION_STATUS_COMPLETED);
    itsInfo.addAdditionalInfo("moreinfo", "someothervalue");

    try {
      itsChargingService.logChargingInfo(info);
    } catch(ChargingException ce) {
      //Handle Exception
    }
  }
}
```

Fields in the `ChargingInfo` object that is not set will use default or `null` values. The call to `setCompletionStatus()` is not necessary, since status `COMPLETED` will be used by default.

Using the time service

The time service was introduced in the previous section. Except from using the Time Manager class to retrieve the system time, it also contains functionality to handle timers.

With the use of timers, the application programmer can protect programs from hanging and avoid dead-lock situations. For example, when requesting resource over a high latency network, the programmer can include a timer with a certain treshold value. If the network resource is unavailable, the timer will expire, and the program can continue, taking any actions necessary to report the faulty resource etc. If the resource is available, the timer is reset and execution continues normally.

With periodic timers, events can be generated repeatedly, with regular intervals. For example, an application can read the contents of a database table containing subscription numbers, and use the result to submit a group SMS.

Also, it is recommended to create a new object reference for each timer to be used. If the same reference is used and you want to cancel a timer, any of the timers using the same object reference could be cancelled.

Listing 3-25 Timer.java

```
package example.helloslee;


import com.incomit.slee.*;

import com.incomit.slee.time.*;


public class Timer {


  private ServiceContext itsServiceContext = null;

  private SLEETimeManager itsTimeManager = null;

  private String timerRef;


Object synchObject = new Object();


  public Timer (ServiceContext aServiceContext,

                String reference)

  {

    itsServiceContext = aServiceContext;

    SLEEContext sc = itsServiceContext.getSLEEContext();
    itsTimeManager = sc.getTimeManager();
    timerRef = reference;
  }

  public void removeTimer () {
    itsTimeManager.cancelTimer(timerRef);
  }

  public void startTimer () {
    itsTimeManager.scheduleTimer(false,
                                 10 * 1000, // 10 seconds
                                 new MyTimerListener(this),
                                 timerRef);
  }
}
```

Listing 3-26 Timer.java -method: MyTimerListener

```
class MyTimerListener implements SLEETimerListener {

  private Timer timer = null;

  MyTimerListener(Timer timer) {
    this.timer = timer;
  }

  public void processTimer(Object reference) {
    synchronized (timer.synchObject) {
    timer.synchObject.notify();
    }
  }
}
```

Listing 3-27 PeriodicTimer.java

```
package example.helloslee;

import com.incomit.slee.*;
import com.incomit.slee.time.*;
import com.incomit.slee.event.*;

public class PeriodicTimer {

  private ServiceContext itsServiceContext = null;
  private SLEETimeManager itsTimeManager = null;
  private EventLogService itsEventService;

  private String timerRef = "periodic timer";

  public static final long PERIOD = 2000;

  Object synchObject = new Object();
```

Listing 3-28 PeriodicTimer.java (continued)

```
public PeriodicTimer (ServiceContext aServiceContext) {
    itsServiceContext = aServiceContext;
    SLEEContext sc = itsServiceContext.getSLEEContext();
    itsEventService =  sc.getEventLogService();
    itsTimeManager = sc.getTimeManager();
  }

  public void doPeriodicTask (int cnt) {
    String message = "This is periodic task " + cnt;
    itsEventService.logEvent(itsServiceContext
                             .getInstanceName().getBytes(),
                             EventLogService.LOW,
                             cnt,
                             message.getBytes());
  }

  public void removeTimer () {
    itsTimeManager.cancelTimer(timerRef);
  }

  public void startTimer () {
    itsTimeManager.scheduleTimer(true,
                                 period,
                                 new MyTimerListener(this, 10),
                                 timerRef);
    try {
      synchronized (synchObject) {
        synchObject.wait();
      }
    }
    catch (java.lang.InterruptedException ie) {
      // Handle exception
    }
  }
}
```

Listing 3-29 PeriodicTimer.java (continued)

```
public class MyTimerListener implements SLEETimerListener {

private int counter = 0;
private int noOfEvents;
private PeriodicTimer timer;

  MyTimerListener(PeriodicTimer timer, int noOfEvents) {
    this.timer = timer;
    this.noOfEvents = noOfEvents;
  }

  public void processTimer(Object reference) {
    if (counter < noOfEvents) {
      timer.doPeriodicTask(counter);
      counter++;
      }
    else {
      synchronized (timer.synchObject) {
        timer.synchObject.notify();
        //do stuff
      }
    }
  }
}
```

Using the trace service

The trace service enables to trace execution of a program. It can be used, for example, when a service is suspected to be erroneous. The trace output is written to file. For performance reasons it is recommended that the application performs a check on the trace active flag before calling the `logTrace` method. The trace file that is generated is service specific.

All services using the trace service will have an associated buffer class. This class will buffer all trace messages from the service until the size of the buffer reach the specified size setting and then flush the buffer to a file.

The information that gets written to the trace log is controlled by using trace filter groups. It is possible to turn on and off different pre-defined trace groups at runtime by supplying a new filter value. The following filters are available:

Filter	Description	Value
`METHOD_IN`	Log trace at entry of method.	1
`METHOD_OUT`	Log trace at exit of method.	2
`USERDEF_1`	User defined trace (DEBUG)	4
`USERDEF_2`	User defined trace (INFO)	8
`USERDEF_3`	User defined trace (WARNING)	16

Filter	Description	Value
`USERDEF_4`	User defined trace (ERROR)	32
`USERDEF_5`	User defined trace	64
`USERDEF_6`	User defined trace (RACE) This trace level shall be used for measuring the time from when entering a certain code segment to a existing ac certain code segment. Normally a code segment is from when entering a module to exiting a module. The elapsed time shall be provided in the trace.	128
`RAW_DATA`	Log any data	256
`EXCEPTION_LOG`	Writes trace information at exceptions that breaks the execution flow.	512
`TRAFFIC FLOW`	Writes trace information when traffic related requests (both application and network initiated) are received by and sent from the service.	1024

To specify which groups to include in the trace log, the values for the different filters are added. For example, if you want to include the entry and exit of a method, as well as information in user defined trace group 4, you should specify a filter value of 35:

```
1 + 2 + 32, or METHOD_IN + METHOD_OUT + USERDEF_4
```

It is not recommended, for performance reasons, to enter a filter value equal to 0 to disable trace logging. Instead the trace for that service should be deactivated.

The code example below shows how to set up a reference to the trace service. It then defines a set of methods that shows how to use the trace methods available in the API. The methods, along with initializations, will be executed from the `trace()` method, which is defined at the end of the example.

Listing 3-30 Tracer.java

```
package example.helloslee;


import com.incomit.slee.*;
import com.incomit.slee.trace.*;

public class Tracer {

  private ServiceContext itsServiceContext;
  private TraceLogService itsTraceService;

  public Tracer (ServiceContext aServiceContext) {
    itsServiceContext = aServiceContext;
    itsTraceService = itsServiceContext.getTraceService();
  }
```

The call to the `ordinaryTrace()` method will cause the trace to attempt a trace of the method entry and exit. It uses the API methods `logTraceIntoMethod()` and `logTraceOutOfMethod()` with the following signatures:

```
public void logTraceIntoMethod(java.lang.String className,
                               java.lang.String methodName)
public void logTraceOutOfMethod(java.lang.String className,
                                java.lang.String methodName)
```

The parameter `className` denotes the current, defining class, and `methodName` is the name of the method in which the statement is included. The calls to `logTraceIntoMethod()` and `logTraceOutOfMethod()` will always cause its information to be included in its corresponding trace group.

Listing 3-31 Tracer.java (continued) -Example of method in and method out trace

```
public void methodInmethodOut() {
      if (itsTraceService.isGroupTraceActive(TraceLogService.METHOD_IN)) {
             itsTraceService.logTraceIntoMethod("Tracer",
                                                "methodInmethodOut");


            }
            // Perform method-specific functionality
          if (itsTraceService.isGroupTraceActive(TraceLogService.METHOD_OUT)) {
                   itsTraceService.logTraceOutOfMethod("Tracer",
                                                "methodInmethodOut");
            }


      }
```

The call to the `userdef1Trace()` method will cause the trace to attempt a trace on user defined level 1. It uses the API methods `logTrace()` with the following signature:

```
public void logTrace(java.lang.String className,
                     java.lang.String methodName,
                     int traceGroup,
                     java.lang.String info)
```

The information from `logTrace()` can be included in any trace group (the value of `traceGroup`). `logTrace()` also allows the inclusion of an arbitrary string in the generated trace log message.

Listing 3-32 Tracer.java (continued) -Example of userdef 1 trace

```
public void userdef1Trace () {
       if (itsTraceService.isGroupTraceActive(TraceLogService.USERDEF_1))
{
              for (int i = 0; i < 10; i++) {
                     itsTraceService.logTrace("Tracer",
                                                 "userdef1Trace",
                                        TraceLogService.USERDEF_1,
                                        "Trace Log Record " + i);
              }
       }
}
```

The call to the `traceFaultyMethod()` will generate an arithmetic exception, which will be logged in the trace log. To facilitate the tracing of exceptions, the API offers the `logTraceException()` method. It has the following signature:

```
public void logTraceException(java.lang.String className,
                              java.lang.String methodName,
                              int traceGroup,
                              java.lang.String info,
                              java.lang.Throwable exception)
```

The `className` and `methodName` are the same as for the `logTrace()` method. The trace log message will be included in the trace group corresponding to the value of the `traceGroup` parameter. Information about the exception, the exception stack trace from the `exception` object, will also be included in the trace log message, along with an arbitrary string (`info`).

Listing 3-33 Tracer.java (continued) -example of exception trace

```
 public void traceFaultyMethod () {


      try {
        int a = 42;
        int b = 0;
        int c = a / b;
      } catch(ArithmeticException ae) {
       if (itsTraceService.isExceptionTraceActive()) {
            itsTraceService.logTraceException("Tracer",
                                            "traceFaultyMethod",
                                            TraceLogService.EXCEPTION_LOG,
                                            Division by zero!",
                                            ae);
       }
      }


    }
  }
```

The `traceRawData()` method below shows how to trace any data in the form of a byte array. For this, the API offers the `logTrace()` method with the following signature:

```
public void logTrace(byte[] buf,
                     int off,
                     int len)
```

The `buf` parameter is a byte array, `off` indicates the starting position in this buffer, and `len` how many bytes from the buffer that will be included in the trace log message. The call to `logTrace` below will cause a byte array representing the string "`rawdata 1 2 3`" being included in the trace log.

Listing 3-34 Trace.java (continued) -example of using raw data trace

```
 public void traceRawData () {
    if (itsTraceService.isGroupTraceActive(TraceLogService.RAW_DATA)) {
       String rawData = "Testing rawdata 1 2 3";
       itsTraceService.logTrace(rawData.getBytes(),
                               8,
                               rawData.length());
    }
  }
```

When exceptions occur, but there are alternative execution paths can be used, so the request still can be executed, the method logTraceException(...) should be used according to the example below. `itsTraceService` is the trace service.

Listing 3-35 Example of use of logTraceException

```
if (itsTraceService.isExceptionTraceActive()) {
                m_ts.logTraceException(CLASSNAME,
                                       METHODNAME,
                                       TraceLogService.EXCEPTION_LOG,
                                       e.getMessage(),
                                       e);
            }
```

To trace the traffic execution flow, the method logTrafficFlowTrace(...) shall be used. It shall only be used for the traffic interfaces. A traffic flow context should be provided, with information on which Service Provider and application that performed the request. In the example below, `m_trafficFlowContext` contains the Service Provider ID and application ID of the requester in the format `m_trafficFlowContext = applicationId + "\\" + serviceProviderId;`

Listing 3-36 .Example of use of logTrafficFlowTrace

```
if (m_ts.isTrafficFlowTraceActive()) {
       Object[] params = {listener, data, address,
                          serviceCode,
                              requesterID};
        m_ts.logTrafficFlowTrace(CLASSNAME,
                                          METHODNAME,
                                          m_trafficFlowContext,
                                          params);
}
```

General sequence diagrams

Below are a set of generalized sequence diagrams describing the call flow through the Network Gatekeeper. The flows are described from an end-to-end perspective, from a northbound, application-facing, Web Service interfaces to a plug-in.

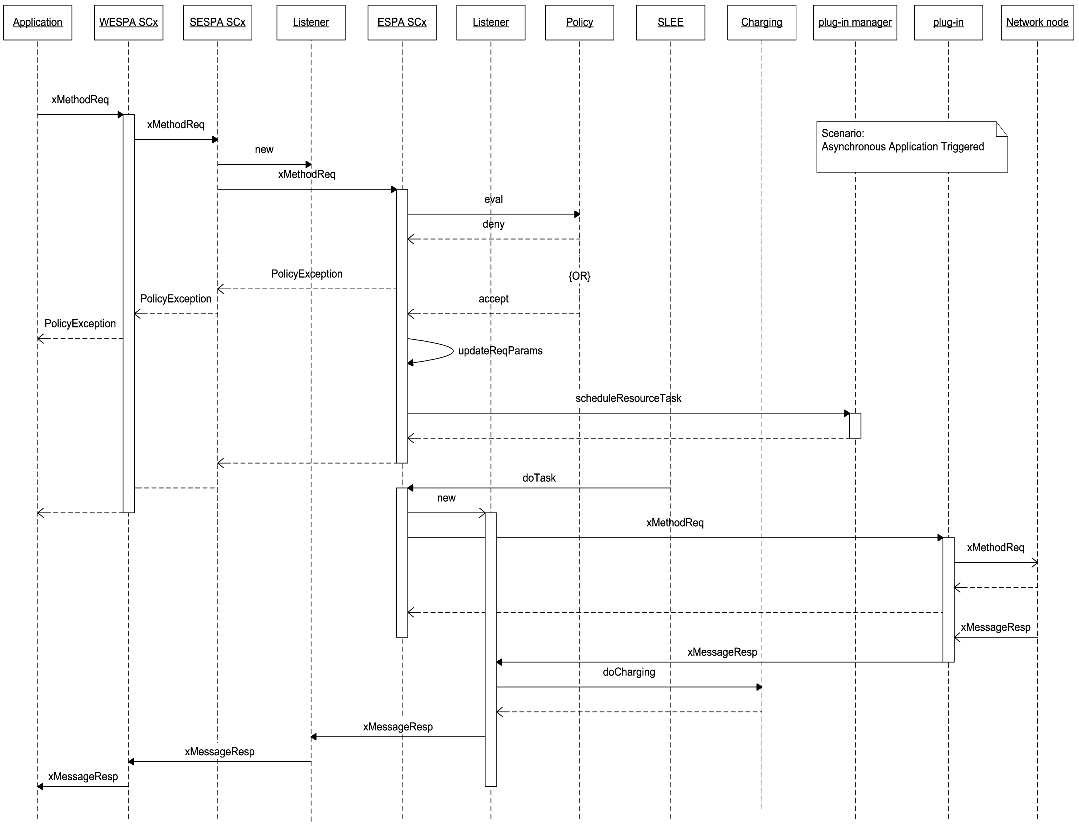
The following sections contain sequence diagrams:

- Asynchronous application-initiated
- Synchronous application-initiated
- Network-triggered

Asynchronous application-initiated

Below is a sequence showing the flow of an arbitrary method request through the Network Gatekeeper. The method call is performed asynchronous from the point-of-view of the application, and the request is performed asynchronous trough the Network Gatekeeper. Requests must always be asynchronous inside Network Gatekeeper, otherwise resources will be reserved for too long from an end-to-end perspective.

Figure 4-1 Asynchronous application-initiated sequence



1. An application performs a method request (SOAP) towards the Web Services implementations (WESPA SC). The implementation of the WESPA SC is deployed in the Embedded Tomcat server.

2. The methods request is propagated to the stateless adapter (SESPA SC). This is a normal Java request, since SESPA has registered its interface, all classes, and all objects in the SLEE Common Loader.

3. A listener class is instantiated. This object is provided as a call-back object when the method request is propagated to the ESPA SC. This method call is using CORBA.

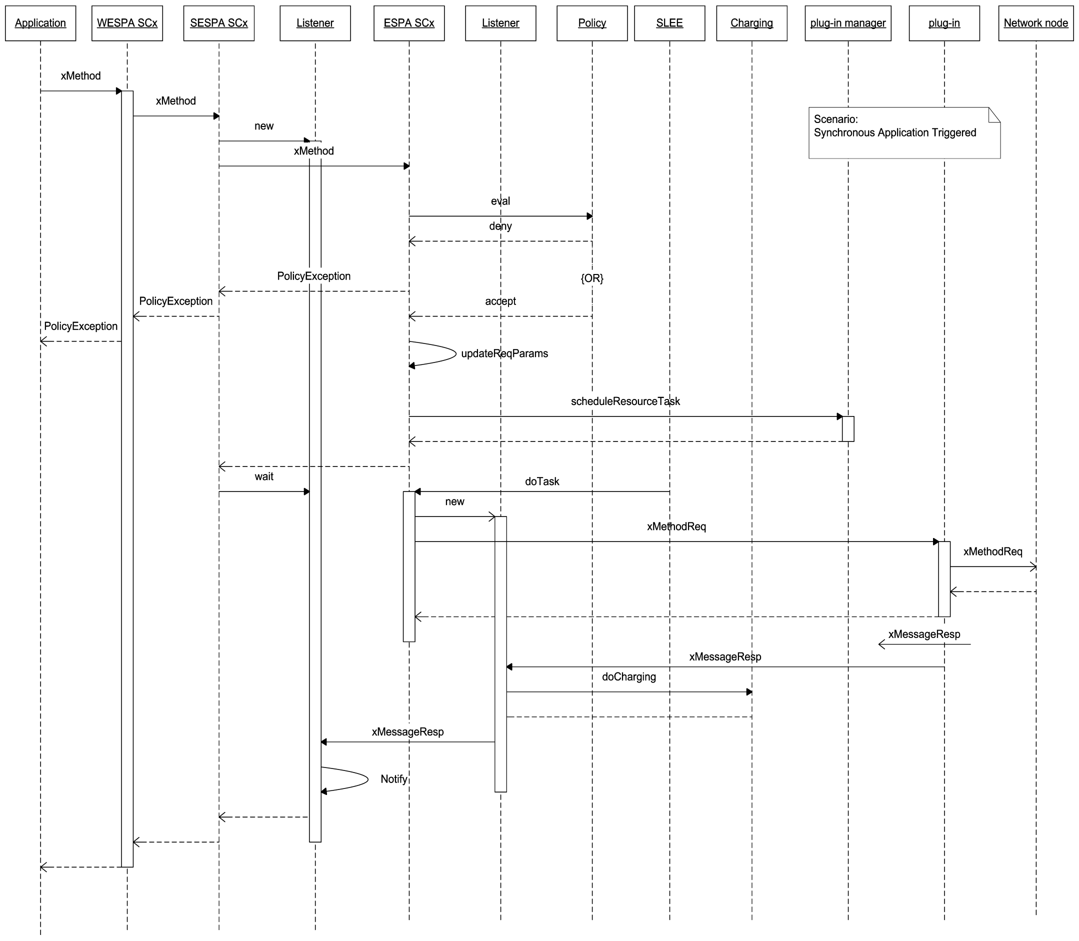
4. In the ESPA SC, a request to evaluate (eval) the request is performed towards the Policy service. This request is called a Policy Enforcement Point (PEP). Two scenarios are possible after policy has evaluated the request:
   a. The method request is denied by the Policy Service, and a PolicyException shall be thrown towards the SESPA SC. The exception will be propagated to WESPA SC and finally to the application.
   b. The method request is accepted by the Policy Service. The parameters feed into the evaluation requests are returned. These may be altered by the Policy Service so the original request parameters shall be updated with the ones received from the policy evaluation.
5. A resource task is scheduled by calling scheduleResourceTask on the plug-in manager.
6. ScheduleResourceTask returns immediately and the execution thread returns back to the SESPA SC, the WESPA SC, and finally to the application.
7. When the Plug-in manager has found a suitable plug-in and a free thread is available from the pool of threads. The SLEE Tread Pool Manager performs doTask(...) on the call-back listener object provided in the call to scheduleResourceTask.
8. When doTask is called, a listener object is created.
9. The request is propagated to the plug-in, with the newly created listener call-back object sent as an argument.
10. The plug-in performs the protocol specifics related to the request. Illustrated here is an asynchronous request, so the operation toward the network node is returned immediately, but the result of the operation will be received later on.
11. When the response to the request reaches the plug-in, it propagates the response to the ESPA SC listener.
12. When the SC has received the response it creates a CDR using the SLEE Charging service.
13. The response is propagated from the ESPA SC listener to the SESPA listener.
14. The SESPA listener propagates the response to the WESPA SC, which propagates the response to the application.

Synchronous application-initiated

Below is a sequence showing the flow of an arbitrary method request through the Network Gatekeeper. The method call is performed synchronous from the point-of-view of the application, although the request is not performed synchronous trough the Network Gatekeeper. Requests must always be asynchronous inside Network Gatekeeper, otherwise resources will be reserved for too long from an end-to-end perspective.

The synchronization is performed in the SESPA layer.

Figure 4-2 Synchronous application-initiated sequence



1. An application performs a method request (SOAP) towards the Web Services implementations (WESPA SC). The implementation of the WESPA SC is deployed in the Embedded Tomcat server.
2. The methods request is propagated to the stateless adapter (SESPA SC). This is a normal Java request. Since SESPA register its interface, all classes, and all objects in the SLEE Common Loader.
3. A listener class is instantiated. This object is provided as a call-back object when the method request is propagated to the ESPA SC. This method call is using CORBA.
4. In the ESPA SC, a request to evaluate (eval) the request is performed towards the Policy service. This request is called a Policy Enforcement Point (PEP). Two scenarios are after policy has evaluated the request:
   a. The method request is denied by the Policy Service, and a PolicyException shall be thrown towards the SESPA SC. The exception will be propagated to WESPA SC and finally to the application.
   b. The method request is accepted by the Policy Service. The parameters feed into the evaluation requests are returned. These may be altered by the Policy Service so the original request parameters shall be updated with the ones received from the policy evaluation.
5. A resource task is scheduled by calling scheduleResourceTask on there plug-in manager.
6. ScheduleResourceTask returns immediately and the execution thread returns back to the SESPA SC.Where SESPA waits for the response to the request to arrive to the listener.
7. When the Plug-in manager has found a suitable plug-in and a free thread is available from the pool of threads. doTask is performed on the call-back listener object provided in the call to scheduleResourceTask.
8. When doTask is called, a listener object is created.
9. The request is propagated to the plug-in, with the newly created listener call-back object sent as an argument.
10. The plug-in performs the protocol specifics related to the request. Illustrated here is an asynchronous request, so the operation toward the network node is returned immediately, but the result of the operation will be received later on.
11. When the response to the request reaches the plug-in, it propagates the response to the ESP A SC listener.

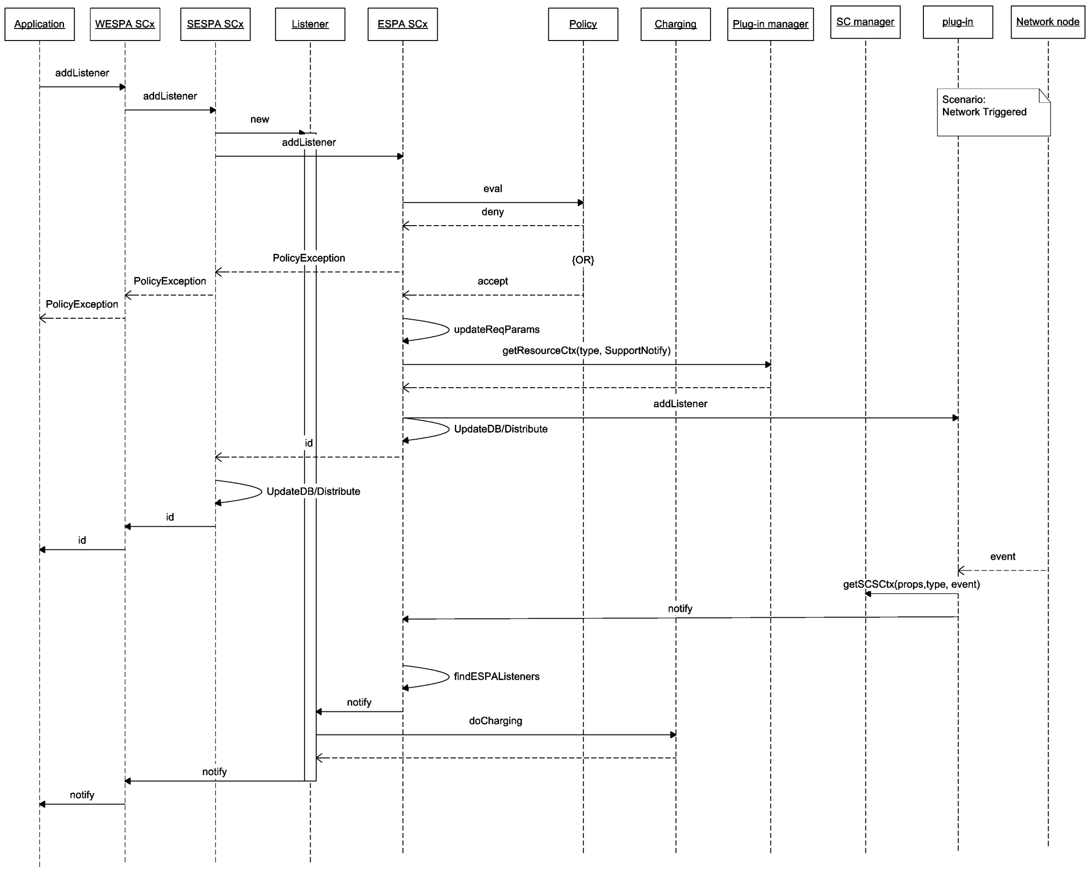
12. When the SC has received the response it creates a CDR using the SLEE Charging service.
13. The response is propagated from the ESPA SC listener to the SESPA listener.
14. The SESPA listener performs a notify on the listener, and thus returning releasing the object previously set in state wait. The response is finally returned to the application through the SESPA SC and the WESPA SC.

Network-triggered

Below is a sequence showing the flow of an arbitrary method network triggered request through the Network Gatekeeper.

Before receiving and forwarding requests originating from the network to an application, the application must register to notifications for the event in question.

Figure 4-3 Network-triggered sequence



Registering the listener

1. An application performs a method request (SOAP) towards the Web Services implementations (WESPA SC). The implementation of the WESPA SC is deployed in the Embedded Tomcat server. The method request (addListener) provides an URL to an end-point, where the application has implemented the Web Service that is used by Network Gatekeeper not notify the application about network-initiated events.

2. The request to add the listener is propagated to the stateless adapter (SESPA SC). This is a regular Java request. Since SESPA register its interface, all classes, and all objects in the SLEE Common Loader.
3. A listener class is instantiated. This object is provided as a call-back object when the method request is propagated to the ESPA SC. This method call is using CORBA.
4. In the ESPA SC, a request to evaluate (eval) the request is performed towards the Policy service. This request is called a Policy Enforcement Point (PEP). Two scenarios are after policy has evaluated the request:
    a. The method request is denied by the Policy Service, and a PolicyException shall be thrown towards the SESPA SC. The exception will be propagated to WESPA SC and finally to the application.
    b. The method request is accepted by the Policy Service. The parameters feed into the evaluation requests are returned. These may be altered by the Policy Service so the original request parameters shall be updated with the ones received from the policy evaluation.
5. The ESPA SC call getResourceCtx on the ResourceDiscovery interface in the plug-in manager. The type of plug-in requested is provided together with a property that asks for if the plug-in is capable of handling network initiated traffic. A plug-in is returned.
6. The ESPA SC updates the database with the reference and distributes the reference to other ESPA SCs.
7. An ID for the listener is returns to the SESPA SC, which also updates the database with the references and distributes the references to other SESPA SCs.
8. The ID is returned to the Application. This ID is used to when removing the listener.
9. When a plug-in has been chosen, the plug-in manger invokes addListener on the plug-in, which starts to listen on network-initiated events. As an alternative there can be one listener in the ESPA SC, and the plug-in distributes all incoming events to this listener. The registration of the listener should be performed when the ESPA SC becomes activated.

Handling incoming events

10. An network-initiated event is routed to the plug-in.

11. The plug-in queries the SC manager for information on which ESPA SCs that has registered for notifications on the event, using the method getSCSCtx on the SCSDiscovery interface. The parameters provided are matched to the parameters given when the listener was created.
12. ScheduleResourceTask returns immediately and the execution thread returns back to the SESPA SC.Where SESPA waits for the response to the request to arrive to the listener.
13. A list of matching ESPA SCs is returned, and the plug-in invokes a method on the ESPA SC, in this case the method called is notify. Which methods to call is specific to the SC.
14. The ESPA SC tries to find all registered listeners in SESPA.
15. When a listener is found, the method request is propagated to the SESPA SC listener which performs the charging specifics.
16. The method request is propagated to the WESPA SC, which will perform a request to the application.

Frameworks

The following sections contain descriptions of the frameworks used by extensions to WebLogic Network Gatekeeper:

- Interacting with the SLEE
- Web Services framework
- Stateless adapter framework
- Service capability framework
- Plug-in framework

All software modules executing as SLEE services interacts with the SLEE. The Web Services framework is used when a software module implements or uses a a Web Service, in both these cases the software module executes as a web application in Embedded Tomcat. The stateless adapter framework is used by the SESPA modules, while the Service capability framework is used by the ESPA modules. The plug-in framework is used by the plug-ins for interacting with the plug-in manager.

Interacting with the SLEE

The framework for interacting with the SLEE is described in "Interacting with the SLEE and the SLEE Utility Services".

Web Services framework

Retrieving the login ticket from the SOAP Header

The login ticket represents a login from an application. The login ticket is provided when the application logs in. The ticket shall be provided in the SOAP header of every request the application makes towards Network Gatekeeper.

The login ticket is provided in the WSSE part of the SOAP header as illustrated below. In this header, the application login user ID is also provided.

Listing 5-1 WSSE header in SOAP Header

```
<wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext">

   <wsse:UsernameToken
xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext">

    <wsse:Username xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext">

     domain_user@app_domain_1.default_provider

    </wsse:Username>

    <wsse:Password xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
Type="wsse:PasswordText">

     app:73183493944772289

    </wsse:Password>

   </wsse:UsernameToken>

  </wsse:Security>
```

The login user ID is found in the <wsse:Username> tag, in the form, <Application Instance group>@<Application Account ID>.<Service Provider ID>. The login ticket is found in the <wsse:Password> tag in the form app:<login Ticket>.

The SOAPHeaderHandler is a utility class in com.bea.wespa.util.SOAPHeaderHandler. The login ticket is provided as a parameter in each request from the WESPA SC to the SESPA SC, see example below.

Listing 5-2 Calling a SESPA method

```
returnValue = m_serviceCapability.myMethod(
                      m_SOAPHeaderHandler.getCurrentSessionTicket(),
                      endpoint,
                      data,
                      address,
                      serviceCode,
                      requesterID);
```

In the example, `m_SOAPHeaderHandler` is instantiated from the SOAP header handler utility.

See "Stateless adapter framework" for a description on how SESPA implements the method and resolves a an ESPA Manager form the login ticket.

Interworking with the stateless adapters (SLEE common loader)

In order for a WESPA SC to use an object in SESPA it is necessary to use the SLEE Common Loader. This section will outline how to use an object in SESPA, while "Stateless adapter framework" describes how the class is registered and the object is loaded.

A WESPA SC must know under which name the object is registered in the SLEE Common loader.

Listing 5-3 Obtain a reference to the interfaces to a Stateless Adapter (SESPA SC)

```
try {
   if (m_serviceCapability == null) {
      Object obj = SleeCommonLoader.getInstance().getObject(
                   OBJ_SESPA_MY_SERVICE_CAPABILITY);
      m_serviceCapability =
      (com.incomit.sespa.myservicecapability.MyServiceCapability) obj;
```

```
    }
} catch (Throwable t) {
    ....
}
```

The SLEE Common Loader is queried for the interface object registered under the name `OBJ_SESPA_MY_SERVICE_CAPABILITY`. The returned object is casted to the correct class.

Since the SESPA SC uses methods in the interface exposed by the WESPA SC, WESPA must also register the interface classes and add the interface objects into the to the SLEE Common loader.

Stateless adapter framework

Interworking with a WESPA SC (SLEE common loader)

The implementation of the Web Services and the software modules executing within the SLEE cannot get hold on each others classes directly. In order to make them reach each other, the SLEE Common Loader is used. The SLEE Common Loader provides a registration and lookup mechanism as described below.

Listing 5-4 Registering a class in the SLEE Common loader

```
m_sc = serviceContext;
String[] ifClasses = {
    "com.incomit.sespa.myservicecapability.MyServiceCapability",
    "com.incomit.sespa.myservicecapability.MyServiceCapabilityListener"
    };
try {
    for (int i=0; i<ifClasses.length; i++) {
    SleeCommonLoader.getInstance().registerClass(m_sc.getJarName(),
                                                          ifClasses[i]);
    }
} catch (SleeCommonLoaderException ex) {
    ....
}
```

In the example above, the SESPA SC implementation registers its interfaces classes so that they can be used by the WESPA SC implementation. First, the name of the classes are defined in a list, and they are registered in the SLEE Common Loader together with the name of the JAR-file they are found in. In this case, the name of the file is resolved using the object m_sc, which is of type ServiceContext. Refer to "Services fetched from the SLEEContext" on page 3-9 for a description of ServiceContext.

The next step is to instantiate an object and then add it to the SLEE Common loader, allowing the WESPA SC to use the objects. This is illustrated below.

Listing 5-5 Add object to the SLEE common class loader, allowing WESPA to access the object.

```
try {
    m_myScImpl = new MyServiceCapabilityImpl(m_sc, m_dbHelper);
} catch (Exception ex) {
    String errorMsg = "Failed to create MyServiceCapabilityImpl. Reason: "
                      + ex.getMessage();
    throw new ServiceDeploymentException(errorMsg,
                                         errorMsg,
                                         m_sc.getName(),
                                         0);
}
try {

    SleeCommonLoader.getInstance().addObject(OBJ_SESPA_MY_SERVICE_CAPABILITY,
                                             m_myScImpl.getProxy());
} catch (SleeCommonLoaderException ex) {
   ...
}
```

The class is instantiated, and the object is added to the SLEE Common Loader. The object is the object implementing the interface given when registering the class (`com.incomit.sespa.myservicecapability.MyServiceCapability`). When the object is added, a name for it is provided (`OBJ_SESPA_MY_SERVICE_CAPABILITY`) which is used for looking up of the object by the WESPA SC. The name must be unique, and it is recommended to use the actual class name as a part of the name. Instead of the object itself, a proxy representing the object is provided. This is due to high availability reasons, and is further explained in "High availability" on page 6-1.

See "Web Services framework" on page 5-2 for information how the object is used by the WESPA SC implementation.

Below is outlined how the object is removed from the SLEE Common Loader.

Listing 5-6 Removing an object from the SLEE Common Loader

```
try {
   SleeCommonLoader.getInstance().removeObject(OBJ_SESPA_MY_SERVICE_CAPABILITY
);
} catch (SleeCommonLoaderException ex) {
   String errorMsg = "Failed to deregister SESPA MyServiceCapability
                     service from SleeCommonLoader. Reason: "
                     + ex.getMessage();
   throw new ServiceDeploymentException(errorMsg,
                                        errorMsg,
                                        m_sc.getName(),
                                        0);
}
```

The object is removed from the SLEE Common Loader, using the name for lookup. When removing the object, all resources used should be cleaned up, and the affected applications should be notified. Objects should be cleared when SLEE calls deactivate(...) on the implementation of the service.

Getting an ESPA session based on the loginticket

The ESPA SC provides the SESPA SC with an application session and a Manager on which the ESPA SC methods are invoked.

In general there is a three step process:

- The SESPA SC logs in to ESPA Access.
- SESPA SC gets an application session that is correlated with the login ticket.
- On the object representing the application session, the ESPA Manager, the SESPA SC can execute the methods provided by the ESPA SC.

There are utility classed in SESPA which assist in retrieving the session and the Manager based on the login ticket.

Below is an example that illustrated how to retrieve the application session and a manager.

Listing 5-7 Getting an application session and a manager

```
ApplicationSession appSession = m_dbHelper.getApplicationSession(loginTicket);

MyServiceCapabilityManager espaManager = getEspaManager(loginTicket);

result = espaManager.myMethodWait(data,

                                  address,

                                  waitTimeoutSeconds,

                                  serviceCode,

                                  requesterID,

                                  appSession);
```

First, the application session is fetched from the database where the correlation between the login ticket and the application session object is stored. This is done using the SESPA utility class com.incomit.sespa.util.DbHelper. m_dbHelper is an object instantiated from this class.

Then the ESPA Manager for the SC is fetched based on the login ticket. The ESPA Manager represents the ESPA SC, and the methods implemented in the ESPA SC can be invoked on this object as illustrated above. Below getEspaManager(...) is illustrated

Listing 5-8 getEspaManager(...)

```
public MyServiceCapabilityManager getEspaManager(String loginTicket)
   throws GeneralException{
   MyServiceCapabilityManager manager = null;
   try {
      ESPAManager espaManager = null;
      espaManager = m_dbHelper.getEspaManager(loginTicket,
                               MyServiceCapabilityManager.SERVICE_NAME);
      manager = MyServiceCapabilityManagerHelper.narrow(espaManager);
   } catch(AccessException ex) {
      ...
   }
   return manager;
}
```

The code fragment illustrates how the ESPA manager is fetched based on the login ticket and the service name defined in `MyServiceCapabilityManager.SERVICE_NAME`. Where ServiceName is defined in the IDL file for the ESPA SC interface.

The manager is then narrowed to the actual class.

Service capability framework

For information on how to interworking with the SC Manager, see "SC manager" on page 2-17 and "Plug-in manager interfaces" on page 2-11.

Plug-in framework

For information on how to interworking with the Plugin manager, see "Plug-in manager interfaces" on page 2-11.

High availability

The following sections contain descriptions of high-availability aspects for extensions to WebLogic Network Gatekeeper:

Introduction

HA is important for both incoming and outgoing traffic. This is handled mostly by the Plug-in and SC Managers. HA and load balancing between the underlying platform and Network Gatekeeper are managed by the SC Manager for northbound traffic, and the plug-in manager for southbound traffic. Recovery and distribution mechanisms only apply to newly created sessions since ongoing sessions will maintain the established reference between Network Gatekeeper and the underlying platform since there is no redundancy on a session level.

Network Gatekeeper would normally run on a cluster of servers running in parallel to support high availability. The number of servers required for the different configurations ranges normally from two to eight. This means for instance that all SCs will be run on all machines and contain exactly the same information, but have different active sessions. Each server running SCs will also execute a plug-in manager and an SC Manager, all synchronized and to be treated as equals. This also means that if one server crashes, applications may continue to use the system uninterrupted. All active sessions on the faulty machine will be lost.

Plug-in Manager and SC Manager

Each Network Gatekeeper server will have its own instance of the Plug-in Manager and SC Manager. All these instances share the same information, which means that it makes no difference which instance is used. Once one manager is obtained references to all other managers can be acquired. This is performed with the getAllResourceManagers method in the ResourceManager interface and getAllSCSManagers method in the SCSMgr interface.

An external plug-in could poll the Plug-in Manager and the SC Manager at regular intervals to see if additional managers have started.

SC Manager

The SC Manager can be used by all plug-ins capable of detecting network triggered sessions to obtain references to SCs.

Each Network Gatekeeper SLEE executes an instance of the SC Manager. All instances within one Network Gatekeeper node are synchronized and are to be treated as equals. Upon startup of an Network Gatekeeper SLEE, all SCs dealing with network triggered sessions will register their callback interfaces in the SC Manager executing in the same SLEE, and the change is propagated between all SC Manager instances.

Plug-ins using SC Manager

If no SC is found to be active or if all are under severe overload, the SC Manager will raise a SCSMgmtException to the getSCS method call. Under such a condition the plug-in should abort the dialogue since no suitable SC is available in the Network Gatekeeper cluster.

An SC returned by the SC Manager has always been checked and found working, however something might have happened to it during the time it takes the plug-in to invoke the reportNotification method. Under such a condition the plug-in could either choose to use the getSCS method again or to abort the dialogue.

There is a pinging mechanism between the SC Manager and the SCs. If an SC is found to be not reachable, it is put in a “zombie list” maintained by the SC Manager. All entries in the zombie list are checked periodically by the SC Manager, and zombies that are found working after some time will be put back in the list of active again. This mechanism deals with the case were network connectivity is lost for some time between Network Gatekeeper hosts.

In the case of inactivity, the plug-in could check the SC Managers for existence periodically by invoking __non_existant() on the SCSManager. However this may work differently with different ORBs but in our case it should be fine since we use the same ORB.

Failure on notification reporting

If the plug-in gets a CORBA exception on reportNotification towards an SC, it must consider the type of exception.

There is once condition that should trigger the plug-in to try with another SC by either invoking SCSDiscovery.getSCS or shifting to another SC if it uses the directly registered callbacks. This condition is if the received exception is a org.omg.CORBA.SystemException and completed status indicates org.omg.CORBA.CompletionStatus.COMPLETED_NO. There is only one way to guarantee that a broken network connection will not result in a lost relationship between Network Gatekeeper and the underlying platform on that SC, and that is putting the SC in a zombie list and perform regular isActive checks on the SC.

Listing 6-1 Examining the type of exception

```
if ( ex instanceof org.omg.CORBA.SystemException ) {

   org.omg.CORBA.SystemException coSyEx = (org.omg.CORBA.SystemException) ex;

   if ( coSyEx.completed == org.omg.CORBA.CompletionStatus.COMPLETED_NO)

      retry = true;

}
```

If the completion status indicates COMPLETED_YES or COMPLETED_MAYBE the plug-in cannot know for sure whether the notification has been handled or not and it should therefore treat the call normally. The plug-in should either start an activity supervision timer on the call that will expire after a certain time if no action is performed on the call from Network Gatekeeper, or it could rely on supervision timers in the MSC that will cause a TC_ABORT from the MSC after some time.

Incoming traffic

This section describes HA regarding the incoming traffic, that is traffic from the telecom network.

When a network-triggered event that should be sent to an SC is received, the SC Manager can be used. This manager will always return an active SC at the time of the request.

Despite this, the SC may crash immediately after the SC was received. In this case, the plug-in will have to retrieve a new SC using the SC Manager. Now, the SC will detect the error and return another working SC instance, if one exists.

The plug-in can also use the call back interfaces that the SCs register directly in the plug-in. If the plug-in detects an error that is not transient, the faulty listeners should be removed. The SC will register as a listener again once it is activated/restarted. On transient errors the plug-in should keep the call back interface and try to reuse it with subsequent calls. On several repeated errors the interface may be discarded even in this case.

If no SCs are available or an error (for example a CORBA system exception) is encountered in an active session then the plug-in must take its own default action and also destroy all objects related to that session.

Outgoing traffic

This section describes HA regarding the outgoing traffic, that is traffic from the SCs to the plug-ins.

The outgoing traffic works in a similar way as the incoming. In this case the Plug-in Manager is responsible to deliver plug-ins to the SCs. If the Plug-in manager detects an error in a plug-in (for example a CORBA system exceptions), it will remove this plug-in.

In the same way that several SCs may run in parallel, plug-ins can also run in parallel to allow the service to be used uninterrupted.

SESPA and ESPA

If a SESPA SC looses contact with the ESPA SC it currently uses, there is an automatic HA switch performed for the ESPA session object and the ESPA Manager object. This is achieved by SESPA who registers a proxy to the SESPA object that implements the SESPA interface in the SLEE Common Loader. See “Stateless adapter framework” on page 5-4.

Below is an example on how the SESPA SC registers the proxy object in this HA handler.

Listing 6-2 Registering an object in the HA Handler

```
m_haHandler = HAHandler.createInstance(m_sc,
                                       this,
                                        null);
```

```
m_haProxy = (com.incomit.sespa.myservicecapability.MyServiceCapability)
m_haHandler.getHAProxy(
com.incomit.sespa.myservicecapability.MyServiceCapability.class);
```

The HA handler is fetched and the Service Context and the object implementing the SESPA SC are provided as arguments. An additional parameter, a custom recovery manager, can also be provided. This is discussed later in this section.

When the HA handler has been retrieved, the class object representing the SESPA interface is provided to the HA handler and a HA proxy object is returned. This HA Proxy object is added to the SLEE Common loader as described in "Stateless adapter framework" on page 5-4.

When HA switch is performed between SESPA and ESPA, this is transparent for the SESPA, so the objects representing the ESPA session and ESPA manager can be used after a HA switch. If the SESPA implementation has objects that are created using either the session or manager object, these are not automatically restored. The SESPA SC must implement this recovery functionality. The object that performs the recovery, the service specific recovery manager, is provided as a parameter to the createInstance(...) method. The method recoversession(...) is called on the service specific recovery manager after the recovery manager has restored the ESPA Session and Manager objects.

An example of when to use service specific recovery manager is when a SESPA implementation on top of ESPA Messaging keeps track of opened mailboxes and reopens them after a HA switch.

CHAPTER 7

Plug-ins that executes as a SLEE service and a web application

The following sections contain descriptions of plug-ins that uses or exposes Web Services:

- Introduction
- Interaction between the web application part of a plug-in and the SLEE service part of a plug-in

Introduction

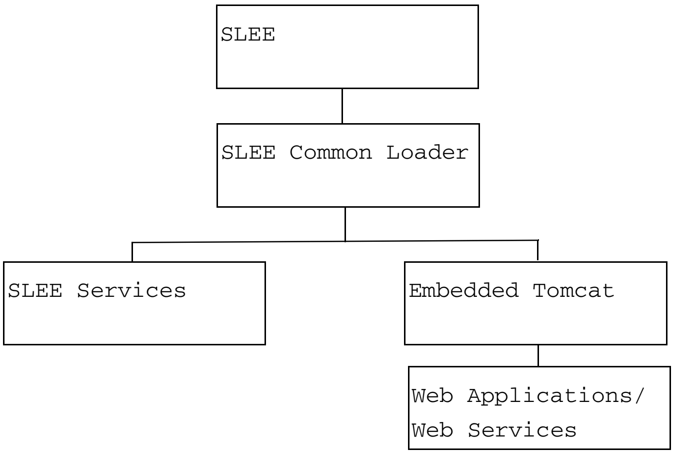
There are a lot of network nodes that exposes Web Services (SOAP/HTML) or other protocols with HTPP as a bearer. When plug-ins communicate with these network elements they can utilize the Tomcat servlet engine provided by the Network Gatekeeper. This section describes how a plug-in, or any SLEE service can interact with the Tomcat.

The plug-in must be divided into two parts, one part that executes as a regular SLEE service and one part that executes as a web application in Tomcat. The Tomcat is itself deployed as a SLEE service, Embedded_Tomcat.

When creating a server part for incoming HTTP requests, a web application in the form of a servlet or a Web Service needs to be created and deployed in Embedded_Tomcat. When creating a client part for outgoing requests, the client part should also be created as a servlet or a Web Services client, and deployed in Embedded_Tomcat. The reason for creating and deploying a servlet or a Web Service client in Embedded_Tomcat is that the necessary Axis classes are a part of the Embedded-Tomcats classpath and not a part of the SLEEs classpath.

Because of the classloader hierarchy in the Network Gatekeeper, the SLEE service part of the plug-in needs to register itself into the SLEE Common loader and the interface needs to be retrieved by the part of the plug-in that executes in Embedded Tomcat.

Figure 7-1 Classloader hierarchy



Interaction between the web application part of a plug-in and the SLEE service part of a plug-in

Interface class registration

It is the responsibility of the SLEE service part of the plug-in to register all interface classes associated with the communication between the SLEE Service part of the plug-in and the web application part of the plug-in. It is important that these classes are registered into SLEE Common Loader before any of them are used within the plug-in. If this is not the case a ClassCastException will occur when the SLEE Service part and the web application part communicate.

When the plug-in goes into Started state, the jar file that contains the interface definition of the implementation together with the fully qualified classname of the interface is registered in the SLEE Common Loader. In the example below, m_sc is the service context provided by the SLEE.

Listing 7-1 Registering the class interfaces

```
String mySOAPReciever_if = ”com.acme.MySOAPReciever_if”;

String JarName = m_sc.getJarName;

SleeCommonLoader.getInstance().registerClass(JarName, mySOAPReciever_if);

String mySOAPSender_if = ”com.acme.MySOAPSender_if”;

String JarName = m_sc.getJarName;

SleeCommonLoader.getInstance().registerClass(JarName, mySOAPSender_if);
```

Incoming requests

For request that originates in the network, a web application or Web Service part of a plug-in needs to be implemented. This part of the plug-in needs to communicate with the part of the plug-in that executes as a SLEE service.

In the SLEE service part of the plug-in

As the SLEE service gets into state Activated, the plug-in instantiates the implementation of the part of the plug-in that handles incoming request.

The object is added to the SLEE Common Loader. An ID is provided when registering the object. This ID is by the web application part of the plug-in to bind the implementation to the interface.

Note: It is important that the object is added in the SLEE Common Loader immediately after it has been instantiated.

Listing 7-2 Add the implementation of the interface to the SLEE Common Loader

```
soapReceiver = new MySOAPReciever_impl(TheContext.getServiceContext(), this);

SleeCommonLoader.getInstance().addObject(MY_SLEESERVICE_OBJECT_ID,
soapReceiver);
```

In the web application part of the plug-in

The web application part of the plug-in gets the object that implements the SLEE service part of the plug-in from the SLEE Common Loader. The object is accessed via the ID that the object was registered in the SLEE Common Loader by the SLEE service part of the plug-in, see Listing 7-2. The object returned is then casted to the correct class. The object shall be retrieved for each request.

Listing 7-3 Fetch the interface from the SLEE Common Loader

```
Object obj = SleeCommonLoader.getInstance().getObject(MY_SLEESERVICEOBJECT_ID);

MySOAPReciever_if mySOAPReciever_if = (MySOAPReciever_if) obj;
```

Since the web application part of the plug-in is fetching the object from the SLEE Common loader, the object must exist, and it must have been registered in the SLEE Common Loader. This means that the web application part of the plug-in must start before the part that executes as a SLEE Service. This is done automatically since the regular SLEE services always are started prior to Tomcat.

Outgoing requests

For request that originates from the Network Gatekeeper, a HTPP client or Web Services client needs to be implemented. This part of the plug-in needs to communicate with the part of the plug-in that executes as a SLEE service.

In the web application part of the plug-in

The part of the plug-in that executes in Embedded_Tomcat should be responsible for constructing and performing the HTPP request. If it is a SOAP request it should be responsible for creating the SOAP message with the help of the AXIS classes available for applications running in Emdedded_Tomcat service.

When the web application being instantiated, it should add itself to the SLEE Common Loader according to Listing 7-4.

Listing 7-4 Add the implementation of the interface to the SLEE Common Loader

```
soapSender = new MySOAPSender_impl(TheContext.getServiceContext(), this);

SleeCommonLoader.getInstance().addObject(MY_SERVLET_OBJECT_ID, soapSender);
```

It is important that the web application part of the plug-in implementation is registered into the SLEE Common Loader before the SLEE Service part of the implementation tries to use the interface. One way to achieve this is to add a method in the interface used by the web application part (and implemented in the SLEE Service part of the plug-in) that notifies the SLEE service part of the plug-in that is has started, and make the web application part of the plug-in to call this method when it goes into state Active.

Since the web application parts always starts after the SLEE services, this will make sure that the SLEE part of the plug-in does not try to use interface prior to that the implementation has been instantiated.

In the SLEE service part of the plug-in

For outgoing requests, the SLEE service part of the plug-in uses the interface class registered by the web application part of the plug-in.

This means that it should fetch the object via the SLEE Common Loader and cast it to the correct class. The object shall be retrieved for each request.

Listing 7-5 Fetch the interface from the SLEE Common Loader

```
Object obj = SleeCommonLoader.getInstance().getObject(MY_SERVLET_OBJECT_ID);

MySOAPSender_if mySOAPSender_if = (MySOAPSender_if) obj;
```

This also means that the SLEE service part of the plug-in must wait until the part executing in Embedded_tomcat has registered the implementing class in the SLEE Common loader.

Call Control

The following sections contain descriptions of plug-ins of call control type:

- Network plug-in
- Use cases

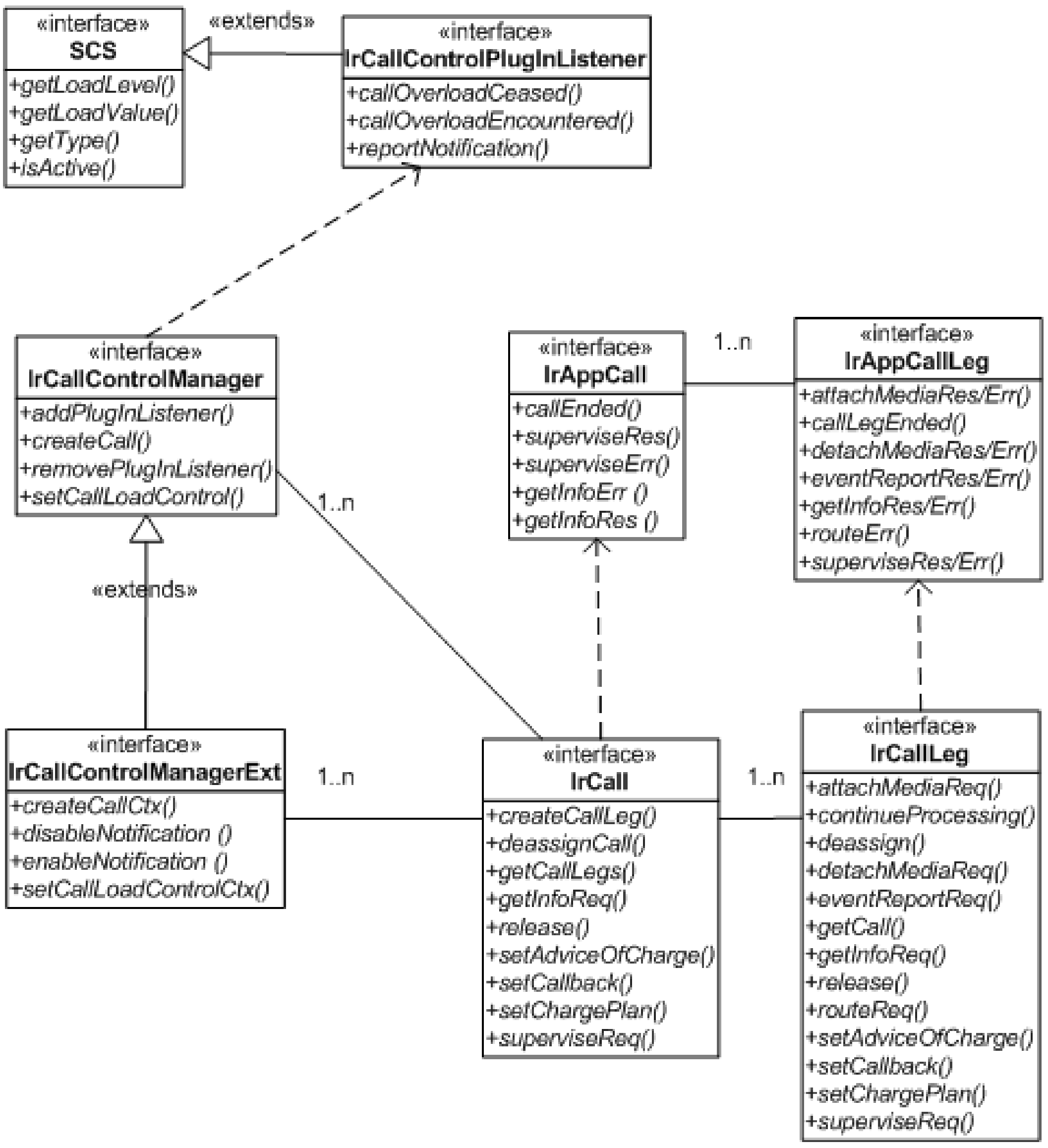
Network plug-in

Interfaces

All call control interfaces are defined in the package com.incomit.resources.callcontrol.

The call control interfaces are similar to the Parlay 3.2 call control interfaces.

Figure 8-1 Call control interface



Interfaces that should be implemented by the plug-in are listed in the table below.

Table 8-1 Interfaces that shall be implemented by a plug-in for Call control

Interface	Description
CallControlResource	Initial object obtained from Plug-in Manager.
IrCallControlManager	Deprecated.
IrCallControlManagerExt	The extended resource multi-party call control manager interface provides the management functions to the multi-party call control plug-ins.
IrCall	The logical representation of a call.
IrCallLeg	The logical representation of a call leg.

Interfaces that are implemented by the Call Control SC.

Table 8-2 Interfaces that are implemented by the Call Control SC, used by a Call control plug-in.

Interface	Description
IrCallControlPlugInListener	Call back interface for receiving network initiated calls, overload events and so on.
IrAppCall	Interface that is used by plug-in to receiving events related to calls.
IrAppCallLeg	Interface for receiving events related to a call leg.

CallControlResource

CallControlResource inherits from the Resource interface and adds an additional method that is used to obtain the call control manager.

Table 8-3 CallControlResource

Method	Description
getCallControlManager	Retrieve call control manager.

IrCallControlManager - Deprecated -

IrCallControlManager is deprecated.

IrCallControlManagerExt

IrCallControlManagerExt is used to create calls, handle load control, and to register plug-in listeners. Only one instance of this type is required.

Table 8-4 IrCallControlManagerExt

Method	Description
createCallCtx	Create a new call object.
disableNotification	Disable a call notification.
enableNotification	Enables a call notification for network initiated traffic.
setCallLoadControlCtx	Used for call gapping.

IrCall

IrCall interface represents a call. Each active call will have one object instance implementing this interface, or if the plug-in uses the session id only one instance is required.

Table 8-5 IrCall

Method	**Description**
createCallLeg	Create a new call leg.
deassignCall	De-assign the call without disconnect.
getCallLegs	Retrieve a list of all call legs belonging to this call.
getInfoReq	Request to receive call information when the call is disconnected.
release	Release this call and disconnect all call legs.
setAdviceOfCharge	Set advice of charge information.
setCallback	Change the call back object used for this call.
setChargePlan	Set the call charge plan.
superviseReq	Supervise a call, that is set granted connect time for this call.

IrCallLeg

The IrCallLeg interface represents a call leg. For each leg in a call there must be an instance implementing this interface, or if the plug-in uses the session id only one instance is required.

Table 8-6 IrCallLeg

Method	Description
attachMediaReq	Attach a detached call leg.
continueProcessing	Resume call processing.
deassign	Release control of leg. The call will remain, but all requested call info and events are disabled.
detachMediaReq	Detach this leg from the call. That is no connection will exist with other call legs.
eventReportReq	Request to event reports.
getCall	Retrieve the call object that this leg belongs to.
getInfoReq	Request to receive call information when the call leg is disconnected.
release	Disconnect this call leg.
routeReq	Route this call leg.
setAdviceOfCharge	Set advice of charge information.
setCallback	Change the call back object used for this call.
setChargePlan	Set the call charge plan for this leg.
superviseReq	Set granted connect time for this call leg.

IrCallControlPlugInListener

IrCallControlPlugInListener listens for events originating in the plug-in. There may be several listeners registered for each plug-in.

This interface inherits from the SCS interface. It makes it possible to narrow a SC object retrieved from the SC Manager to an IrCallControlPlugInListener.

Table 8-7 IrCallControlPlugInListener

Method	Description
callOverloadCeased	Report that overload has ceased.
callOverloadEncountered	Report an overload in the plug-in.
reportNotification	Send notification about a network-triggered call.

IrAppCall

IrAppCall is used to receive events related to a specific call.

Table 8-8 IrAppCall

Method	Description
callEnded	Notification that the call has ended.
superviseRes	Response to a previous call to superviseReq.
superviseErr	Response to a previous call to superviseErr.
getInfoErr	This asynchronous method reports that the original request was erroneous, or resulted in an error condition.
getInfoRes	This asynchronous method reports time information of the finished call or call attempt as well as release cause depending on which information has been requested by getInfoReq.

IrAppCallLeg

IrAppCallLeg is used to receive events related to a specific call leg.

Table 8-9 IrAppCallLeg

Method	Description
attachMediaRes/Err	Response to a previous call to attachMediaReq.
callLegEnded	Inform that call leg has ended.

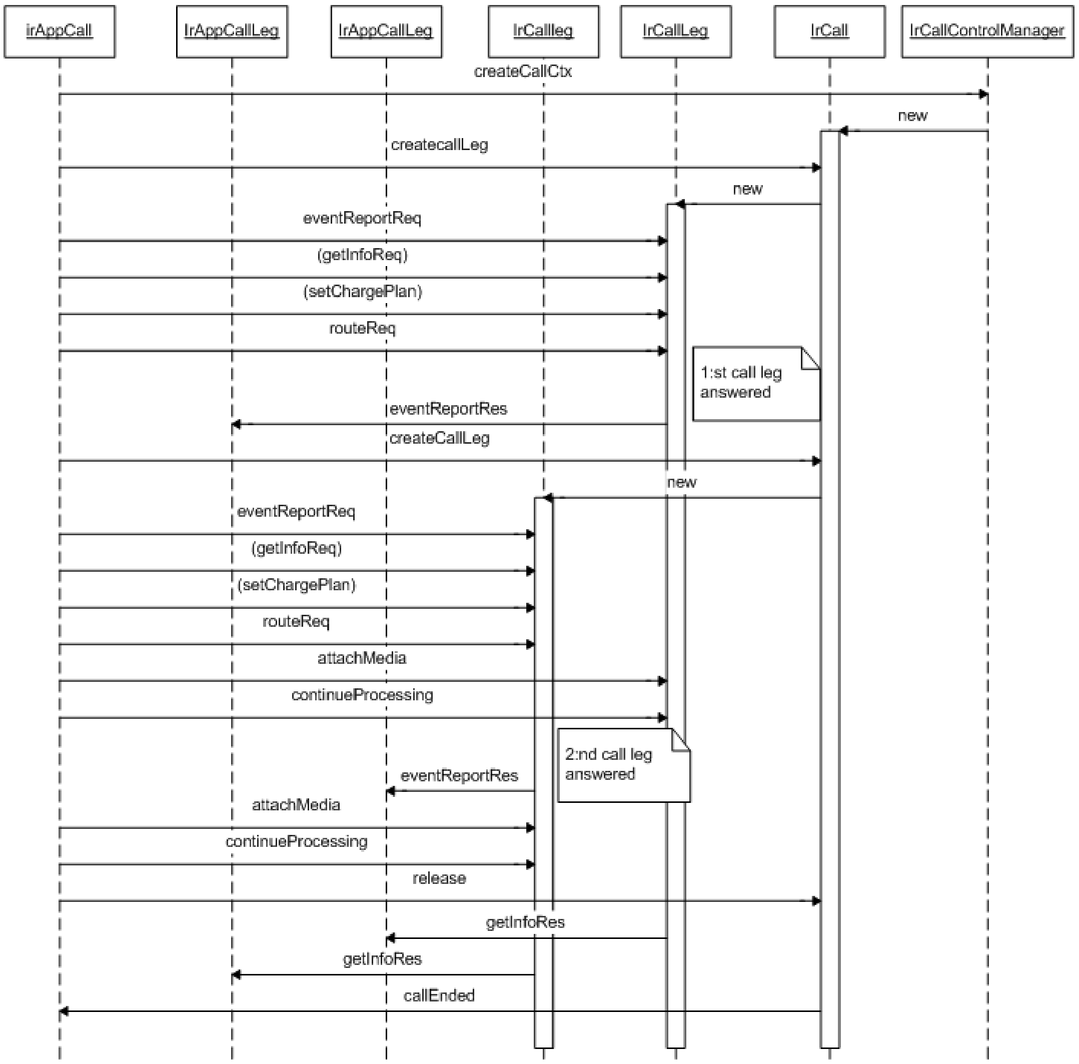
Method	Description
detachMediaRes/Err	Response to a previous call to detachMediaReq.
eventReportRes/Err	Response to a previous call to eventReportReq.
getInfoRes/Err	Response to a previous call to getInfoReq.
routeErr	Report an error when routing call leg.
superviseRes/Err	Response to a previous call to superviseReq.

Use cases

Application-initiated two-party call

The following sequence diagram show a basic two-party call. How the SC obtains the call control manager is not shown here.

Figure 8-2 Application-initiated two-party call



Details about the sequence diagram:

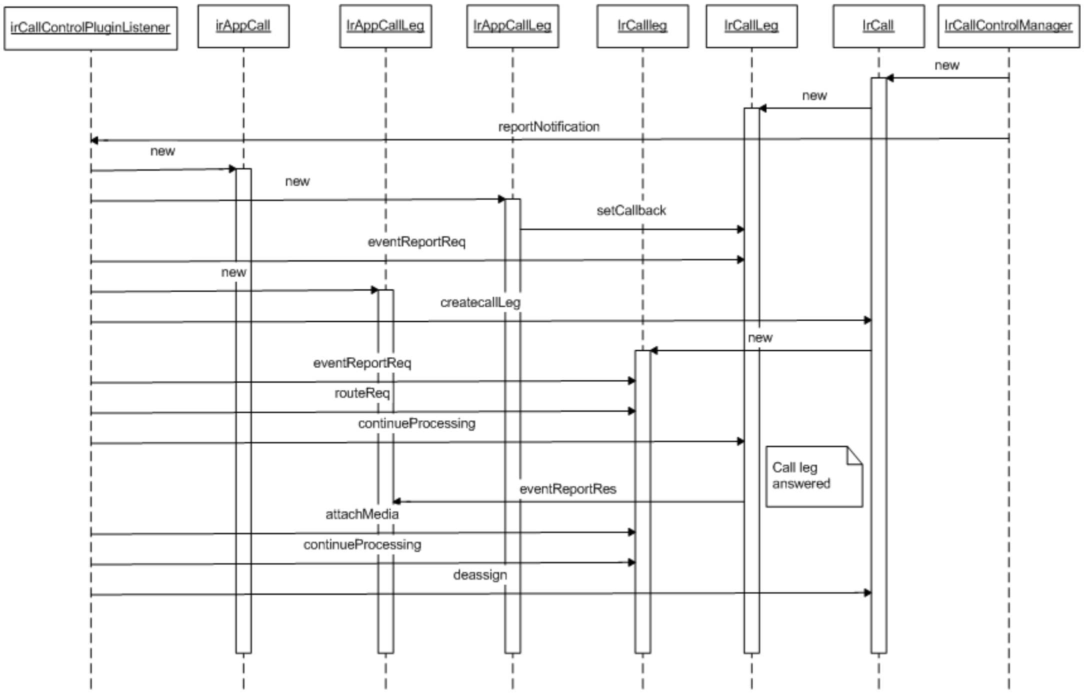
- The client application requests to create a new call and this object is created and returned.
- A call leg is created for the first party.

- Event reports and call information is requested. Charge plan is set and the call is routed. Preferably, the plug-in would buffer these calls and send everything when routeReq is called. getInfoReq and setChargePlan are not mandatory.
- After a while when the call leg is answered the answer event is sent to the application with eventReportRes. Other events could also be sent here, such as busy or no answer.
- The application decides to connect another call leg to this call.
- The same methods may be invoked as when creating the first call leg. The first call leg is in interrupted state, so call processing is on hold until continueProcessing is invoked on this leg.
- attachMediaReq is invoked on the first call leg. Note that all call legs created by an application are detached initially.
- continueProcessing is invoked. At this time, any buffered events in the previous steps should be sent.
- The second leg is answered and eventReportRes is called.
- The leg is attached and continueProcessing invoked.
- The application decides to release the call.
- Call information is sent for each call leg where information was requested.
- The end of the call is notified.

Network-triggered call

This example shows how a basic network-triggered call may be handled. For each new call the plug-in should use the SC Manager to obtain a listener interface or use one of the registered plug-in listeners.

Figure 8-3 Network-triggered call



Details about the sequence diagram:

- The plug-in receives a network triggered call event and creates the call and call leg object that is sent to the plug-in listener. The call is now in interrupted state.
- The application creates the callback interfaces and notifies the plug-in about these.
- The application requests to receive event notification for the first call leg.
- A new call leg is created.
- First event reports are requested for this leg. Then the call leg is routed, but no operations are sent until continueProcessing is invoked.
- The second call leg is answered.
- The second call leg is attached to the call and continueProcessing is invoked.

- The call is de-assigned. This will release all objects related to this call, but the call will remain active in the network.

Call user interaction

The following sections contain descriptions of plug-ins of call user interaction type:

- Network plug-in
- Use cases for Call user interaction

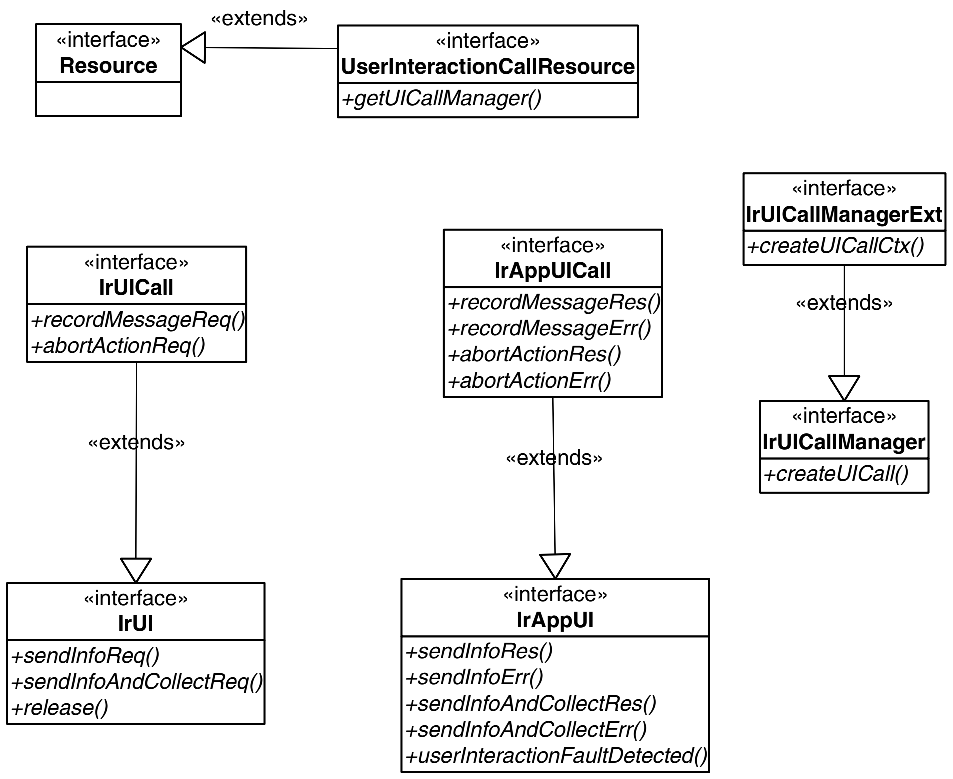
Network plug-in

Call user interaction interfaces

All Call user interaction interfaces are defined in the files UserInteractionCallResource_IF.idl and UserInteractionResource_IF.idl located in bea\wlng21\esdk\idl\plugin_if\ui

There are two interface definition files, where general user interaction interfaces are defined in UserInteractionResource_IF.idl and functionality related only to Call user interaction is found in UserInteractionCallResource_IF.idl.

Figure 9-1 Call user interaction interfaces



Interfaces that should be implemented by the plug-in are listed in the table below.

Table 9-1 Interfaces that shall be implemented by a plug-in for Subscriber profile

Interface	Description
Resource	Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.
UserInteractionCallResource	Manager object that is used to create and release sessions towards the Call user interaction plug-in.

Interface	Description
IrUI	Provides functions for sending information, typically prompt messages and to order collection of information from the end user.The information is typically collected from IVRs capable of collecting user input in the form of DTMF. It also provides functions for ending, or releasing, a user information session.
IrUICall	Provides functions for ordering recording of messages and to abort a request.
IrUICallManager	Deprecated.
IrUICallManagerExt	Provides functions for creating objects representing a Call user interaction session.

Interfaces that are implemented by the ESPA Call user interaction SC.

Table 9-2 Interfaces that are implemented by the ESPA Call user interaction SC, used by a Call user Interaction plug-in.

Interface	Description
IrAppUI	Call back interface for receiving responses to operations performed via the IrUI interface.
IrAppUICall	Call back interface for receiving responses to operations performed via the IrUICall interface.

UserInteractionCallResource

UserInteractionCallResource inherits from the Resource interface and adds an additional method that is used to obtain the Call user interaction manager.

Table 9-3 UserInteractionCallResource

Method	Description
getUICallManager	Retrieve a Call user interaction manager. The returned object shall be narrowed from a IrUICallManager to a IrUICallManagerExt object since IrUICallManager is deprecated.

IrUICallManager -deprecated-

IrUICallManager is deprecated.

IrUICallManagerExt

IrUICallManagerExt is used to create Call user interaction sessions. Only one instance of this type is required.

Table 9-4 IrUICallManagerExt

Method	Description
createUICallCtx	Create a new Call user interaction object.

IrUI

The User Interaction Service Interface provides functions to send information to, or gather information from a user.

Table 9-5 IrUI

Method	Description
sendInfoReq	Plays an announcement or sends other type of information to the end user.
sendInfoAndCollectReq	Plays an announcement or sends other type of information to the end user and collects input from the end user.
release	Releases an user interaction session. Releases all resources associated with the user interaction session and terminates the ongoing user interaction session.

IrUICall

The User Interaction Call Service Interface provides functions for recording messages from an end user and to abort user interaction operations.

Table 9-6 IrUICall

Method	Description
recordMessageReq	Records a message from the end user.
abortActionReq	Aborts a previously ordered operation.

IrAppUI

Table 9-7 IrAppUI

Method	Description
sendInfoRes	Result of a successful sendInfoReq request.
sendInfoErr	Result of a failed sendInfoRes request.
sendInfoAndCollectRes	Result of a successful sendInfoAndCollectReq request.
sendInfoAndCollectErr	Result of a failed sendInfoAndCollectReq request.
userInteractionFaultDetected	Indicates that a fault has been detected in the user interaction.

IrAppUICall

Table 9-8 IrAppUICall

Method	Description
recordMessageRes	Result of a successful recordMessageReq request.
recordMessageErr	Result of a failed recordMessageReq request.
abortActionRes	Result of a successful request to abort a user interaction operation.
abortActionErr	Result of a failed request to abort a user interaction operation.

Use cases for Call user interaction

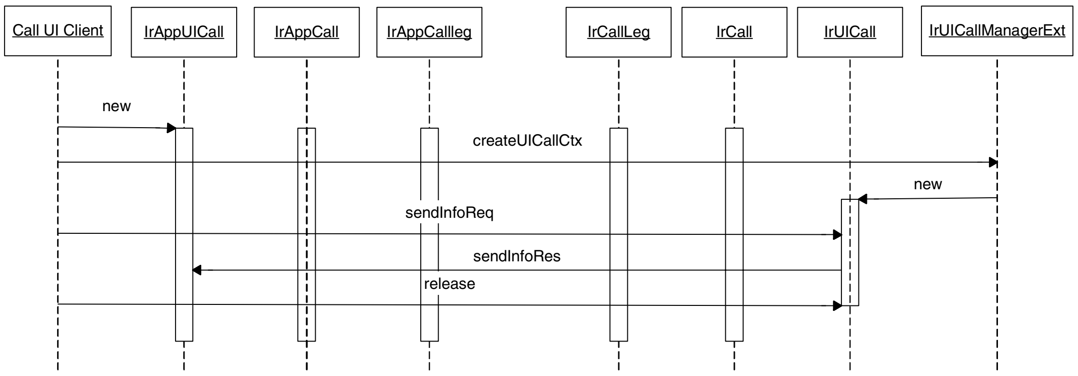
A Call user interaction plug-in only supports operations originating from an application. Network triggered is not supported.

A Call user interaction session is established on an existing call. The call is created with an identifier that identifies an already created call. The call can be either initiated by an application or from the network.

Application-initiated usage of a Call user interaction plug-in

The following sequence diagram show a basic interaction between a user of Call user interaction plug-in and the Plug-in.

Figure 9-2 Application-initiated usage of Call user interaction plug-in.



Details about the sequence diagram:

- Prior to this sequence the Call user interaction client has received a plug-in. See "General usage (application-initiated events)" on page 2-15. The client shall also have access to a IrAppCall object representing an ongoing call.
- The Call user interaction client calls createUICallCtx on the implementation of the IrUICallManagerExt interface. This interface is implemented in the plug-in. An identifier for the Call user interaction session, along with the interface is returned to the client.
- The client request to send information via, sendInfoReq, to one or more participants (represented by call legs) in the call.

- The result of the sendInfoReq request is returned asynchronously to the IrAppUICall interface implemented by the client via sendInfoRes. If the plug-in experiences an error when sending the info to the participant in the call, the method sendInfoErr is invoked instead.
- Additional operation s can be performed towards the IrUICall interface, all operations belonging to the user interaction session. To end the session, the client calls release on the IrUICall interface.

It is not illustrated here how to end the call session that was used for the Call user interaction session.

SMS and MMS messaging

The following sections contain descriptions of plug-ins of SMS and MMS type:

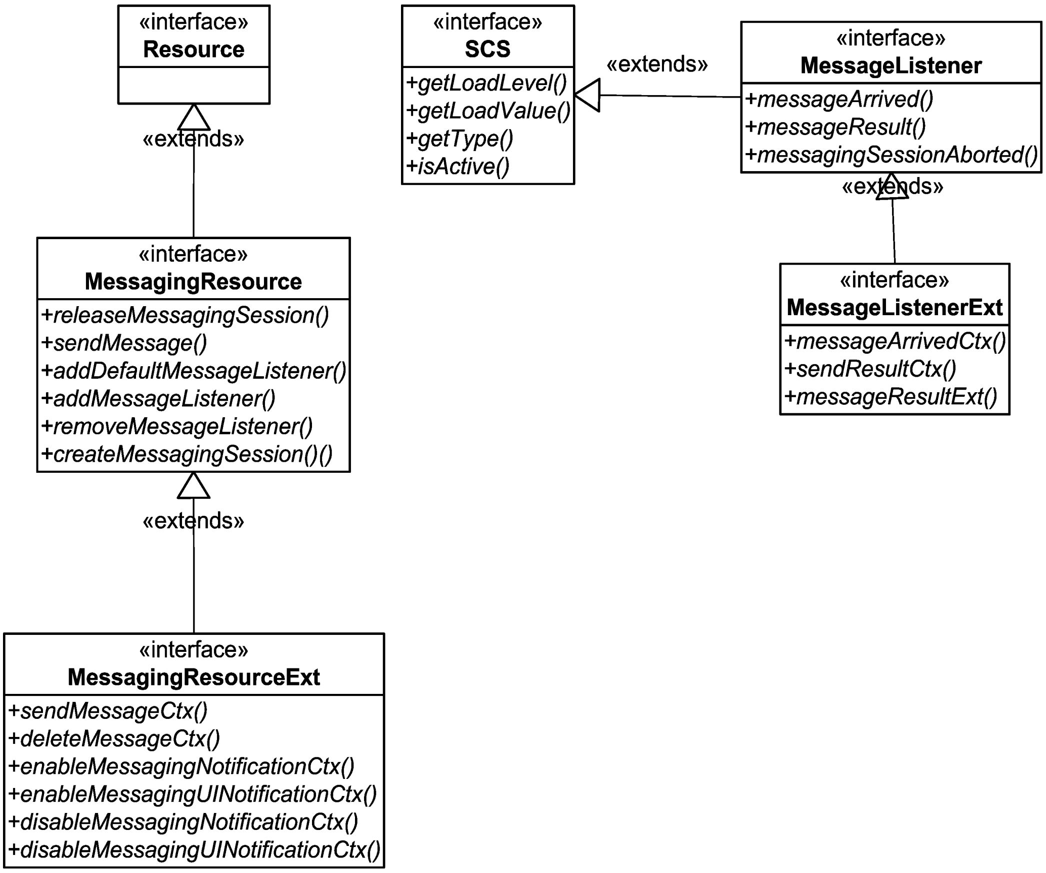
- Network plug-in
- Use cases for SMS

Network plug-in

SMS Interfaces

All SMS messaging interfaces are defined in the package com.incomit.resources.messaging.

Figure 10-1 SMS interface



Interfaces that should be implemented by the plug-in are listed in the table below.

Table 10-1 Interfaces that shall be implemented by a plug-in for SMS messaging

Interface	Description
Resource	Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.
MessagingResource	Manager object that is used to create and release messaging sessions. Used for message based user interaction.
MessagingResourceExt	The extended SMS messaging plug-in interface provides functions for sending and deleting messages, and to enable and disable notification listeners.

Interfaces that are implemented by the Messaging SC.

Table 10-2 Interfaces that are implemented by the Messaging SC, used by a SMS messaging plug-in.

Interface	Description
MessageListener	Call back interface for handling network initiated messages and delivery receipts for sent messages. Delivery receipts are returned when the message has been delivered to the terminal.
MessageListenerExt	Extended call back interface for handling network initiated messages. and delivery receipts for sent messages. Delivery receipts are returned when the message has been delivered to the terminal.

MessagingResource

MessagingResource inherits from the Resource interface and contains some datatypes and adds an method for using messaging sessions.

Table 10-3 MessagingResource

Method	Description
createMessagingSession	Create a message based user interaction session.
releaseMessagingSession	Release a previously created messaging session. Used for message based user interaction.

Method	Description
sendMessage	Deprecated.
addDefaultMessageListener	Deprecated.
addMessageListener	Deprecated.
removeMessageListener	Deprecated.

MessagingResourceExt

MessagingResourceExt is an extended version of the MessagingResource interface and should be used instead of this.

Table 10-4 MessagingResourceExt

Method	Description
sendMessageCtx	Sends a message.
deleteMessageCtx	Deletes a message from the underlying storage.
enableMessagingNotificationCtx	Enable message notification for a specified set of notification criteria to the Messaging SC. Not used by existing messaging plug-ins, since all incoming messages are routed automatically to a mailbox.
enableMessagingUINotificationCtx	Enable message notification for a specified set of notification criteria to the Messaging user interaction SC.
disableMessagingNotificationCtx	Disable messaging notification for a specific notification assignment ID.
disableMessagingUINotificationCtx	Disable messaging notification for a specific notification assignment ID.

MessageListener

MessageListener listens for events originating in the plug-in. There may be several listeners registered for each plug-in.

This interface inherits from the SCS interface. It makes it possible to narrow a SC object retrieved from the SC Manager to an MessageListener.

Table 10-5 MessageListener

Method	Description
messageArrived	Deprecated.
messageResult	Results of previous send message operations are reported using this interface.
messagingSessionAborted	Indications that a previously created messaging session has been aborted are reported using this method.

MessageListenerExt

MessageListenerExt listens for events originating in the plug-in. There may be several listeners registered for each plug-in. This interface extends the MessageListener interface and should be used instead of that.

This interface inherits from the SCS interface. It makes it possible to narrow a SC object retrieved from the SC Manager to an MessageListener.

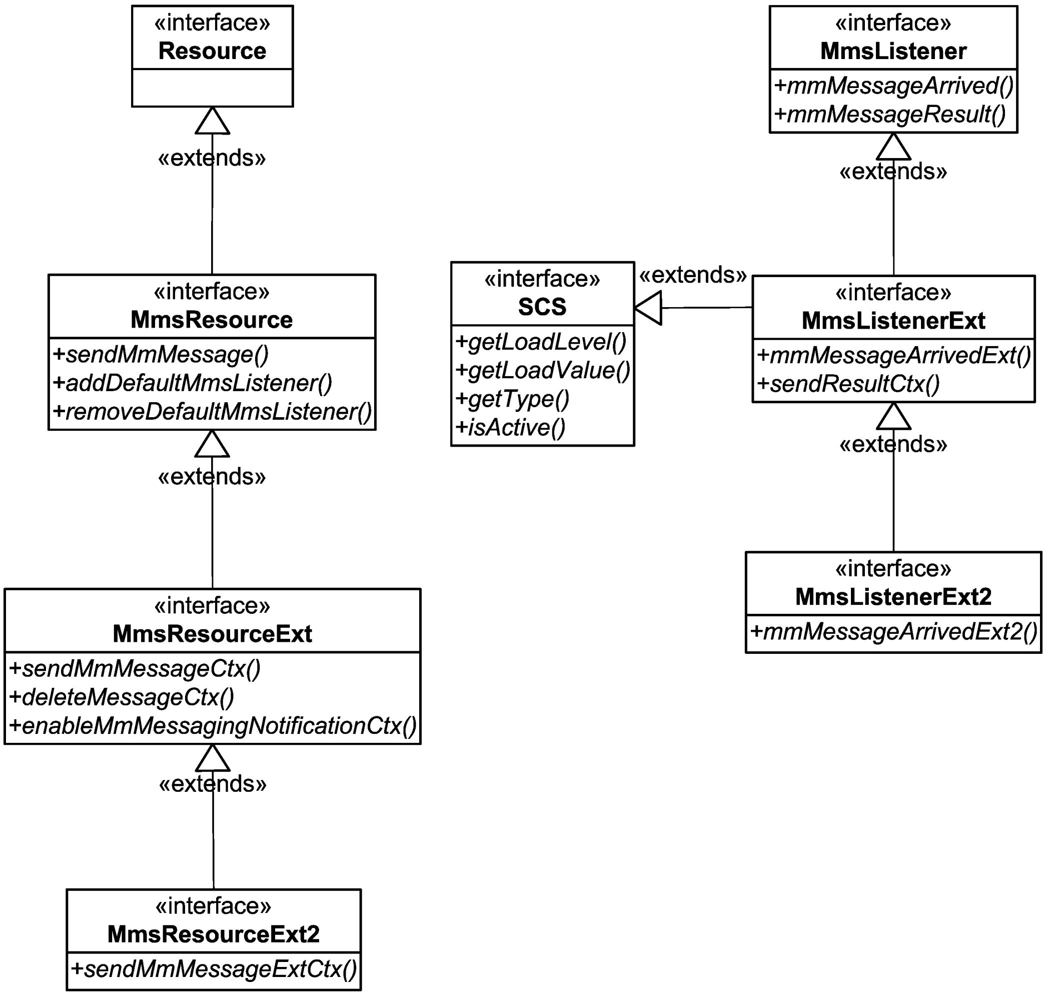
Table 10-6 MessageListenerExt

Method	Description
messageArrivedCtx	Will be used for notifying a listener that a network initiated message has arrived.
sendResultCtx	Will be used for notifying a listener that a application initiated message has been delivered to the terminal.
messageResultExt	A listener will be notified if a message was sent by the underlying plug-in. Holds a status code with more detailed information on the status of the message.

MMS Interfaces

All MMS messaging interfaces are defined in the package com.incomit.resources.messaging.

Figure 10-2 MMS interface



Interfaces that should be implemented by the plug-in are listed in the table below.

Table 10-7 Interfaces that shall be implemented by a plug-in for MMS messaging

Interface	Description
Resource	Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.
MmsResource	Used for sending MM Messages.
MmsResourceExt	The extended MMS messaging interface provides functions for sending and deleting messages, and to enable and notifications.
MmsResourceExt2	The extensions to the extended MMS messaging interface provides functions for sending messages.

Interfaces that are implemented by the Messaging SC.

Table 10-8 Interfaces that are implemented by the Messaging SC, used by a MMS messaging plug-in.

Interface	Description
MmsListener	Call back interface for handling network initiated MMS messages.
MmsListenerExt	Extended call back interface for handling network initiated MMS messages. Also provides functionality for receiving notifications about a sent MMS.
MmsListenerExt2	Extended call back interface for handling network initiated MMS messages.

MmsResource

MmsResource is the base interface for sending MMS messages.

Table 10-9 MmsResource

Method	Description
sendMmMessage	Sends an MMS message.
addDefaultMmsListener	Deprecated.
removeDefaultMmsListener	Deprecated.

MmsResourceExt

MmsResourceExt is extended interface to for sending MMS messages.

Table 10-10 MmsResourceExt

Method	Description
sendMmMessageCtx	Sends an MMS message. Results of messages sent using this method are delivered to any service capability instance registered using the SC Manager.
deleteMessageCtx	Delete an MMS message from the underlying storage
enableMmMessagingNotificationCtx	Enable message notification for a specified set of notification criteria.

MmsResourceExt2

MmsResourceExt2 is extended interface to for sending MMS messages.

Table 10-11 MmsResourceExt2

Method	Description
sendMmMessageExtCtx	Sends an MMS message. Results of messages sent using this method are delivered to any service capability instance registered using the SC Manager.

MmsListener

MmsListener listens for events originating in the plug-in. There may be several listeners registered for each plug-in.

This interface inherits from the SCS interface. It makes it possible to narrow a SCS object retrieved from the SC Manager to an MmsListener.

Table 10-12 MmsListener

Method	Description
mmMessageArrived	Deprecated.
mmMessageResult	Results of previous send message operations are reported using this interface.

MmsListenerExt

MessageListenerExt listens for events originating in the plug-in. There may be several listeners registered for each plug-in. This interface extends the MmsListener interface and should be used instead of that.

This interface inherits from the SCS interface.It makes it possible to narrow a SC object retrieved from the SC Manager to a MmsListenerExt. It also extends the MmsListener interface.

Table 10-13 MmsListenerExt

Method	Description
mmMessageArrivedExt	Will be used for notifying a listener that a network initiated message has arrived.
sendResultCtx	Will be used for notifying a listener that a application initiated message has been sent to the underlying network.

MmsListenerExt2

This interface shall be implemented to be able to receive messages from an MMS plug-in. It also extends the MmsListenerExt interface.

Table 10-14 MmsListenerExt2

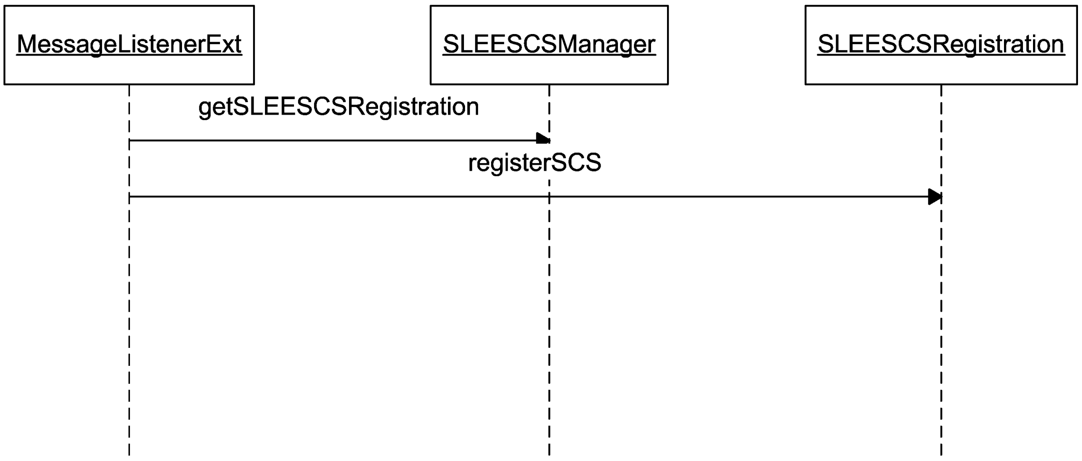
Method	Description
mmMessageArrivedExt2	Will be used for notifying a listener that a network initiated message has arrived.

Use cases for SMS

ESPA Service Capability registers SC

The following sequence diagram shows how the ESPA Messaging SC registers itself in the SC Manager. When creating a messaging plug-in, it is not necessary to perform this task, but it is included to illustrate that the SC must register itself and that the plug-in use the registered information.

Figure 10-3 ESPA Service Capability registers in the SC Manager



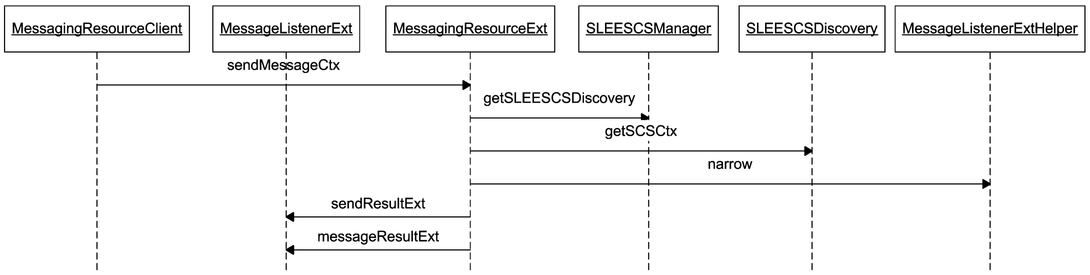
Details about the sequence diagram:

- First, the MessageListenerExt implementation fetches the SLEESCSRegistration interface from the SLEESCSManager.
- It then registers the MessageListenerExt interface with the SC Manager (on SLEESCSRegistration). The class implementing MessageListenerExt is used by the plug-in to notify the ESPA Messaging SC on events related to the outcome of sendMessageCtx operations, see Application-initiated send message, and messageArrivedCtx, see Network-triggered messages, operations.

Application-initiated send message

The following sequence diagram show a basic send message interaction between the Messaging SC and the Plug-in. Note that a plug-in shall uses the SC manager to get an ESPA SC to deliver the delivery receipts, via MessageListenrExt interface. This is due to that it may take long time before a delivery request is sent back to the plug-in, so ESPA Messaging SC itself is acting as a listener for delivery reports.

Figure 10-4 Application-initiated send message



Details about the sequence diagram:

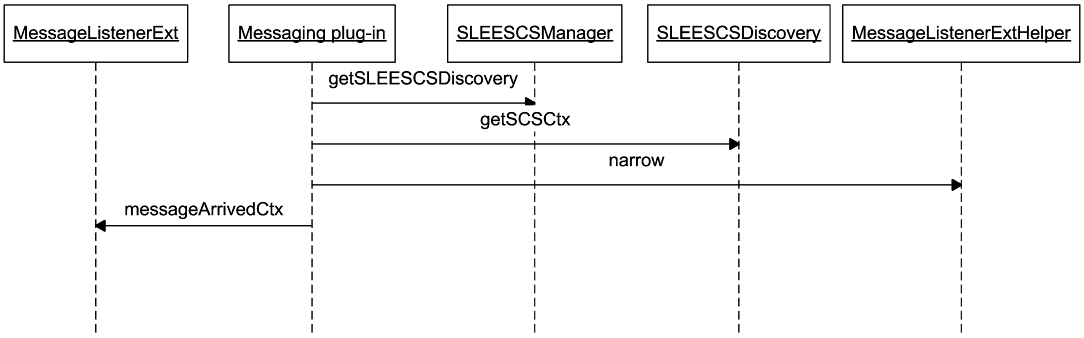
- Prior to this sequence the ESPA Messaging SC has received a plug-in. See "General usage (application-initiated events)" on page 2-15.
- The MessagingResourceClient application performs a SendMessageCtx call to The implementation of the MessagingResourceExt interface. This interface is implemented in the plug-in.
- The plug-in fetched the ESPA SC to report the result of the sendMessageCtx call. The ESPA SC listener interface, the implementation of the MessagelistenerExt interface, is fetched by performing a getSCSCtx operation on the SLEESCSDiscovery interface.
- The MessagelistenerExt interface returned is narrowed to the appropriate object by calling narrow on MessageListenerHelper. This class is auto generated when generating Java stubs from the MessageListenerExt IDL interface.
- When sendMessageCtx is performed on the plug-in's implementation of the MessagelistenerExt interface, it is the plug-ins responsibility to convert the request to a protocol-specific request and send the request to the network element.

- If the sendMessageCtx operation contains more than one destination address, that is it contains a sendlist, the plug-in calls sendResultExt on the ESPA SC as soon as the network node has received the send message request, and acknowledged it. The ESPA Messaging SC will create a CDR with completion status partial.
- When the destination terminal(s) have received the message, and acknowledged it to the plug-in, via the network node, the plug-in calls sendResultExt on the ESPA SC. The ESPA Messaging SC will then create a CDR.

Network-triggered messages

This example shows how a basic network-triggered message may be handled. For each new incoming message the plug-in should use the SC Manager to obtain a listener interface. Prior to this sequence, an application must have registered for notifications on network-initiated messages. It is not necessary for ESPA enable specific listeners for incoming traffic, since the ESPA messaging SC listens to all incoming traffic and distributes the message to a mailbox. Application register listeners that listens for events, such as new message arrived events, on the mailbox.

Figure 10-5 Network-triggered message



Details about the sequence diagram:

- The plug-in receives a network triggered message from the underlying telecom network.
- The plug-in fetches an appropriate ESPA SC based on a set of properties and a type identifier given in getSCSCtx(...). The type identifier is SCS.MESSAGING_TYPE in for ESPA Messaging (defined in resource_common.idl).

- A list of ESPA Messaging SCs are returned. The SLEESCSDiscovery interface provides load balancing, so it is recommended to use the first in the list.
- The MessagelistenerExt interface returned is narrowed to the appropriate object by calling narrow on MessageListenerHelper. This class is auto generated when generating Java stubs from the MessageListenerExt IDL interface.
- messageArrivedCtx is invoked by the plug-in on the MessageListenerExt interface implemented by the ESPA Messaging SC.

Content based charging

The following sections contain descriptions of plug-ins of Content based charging type:

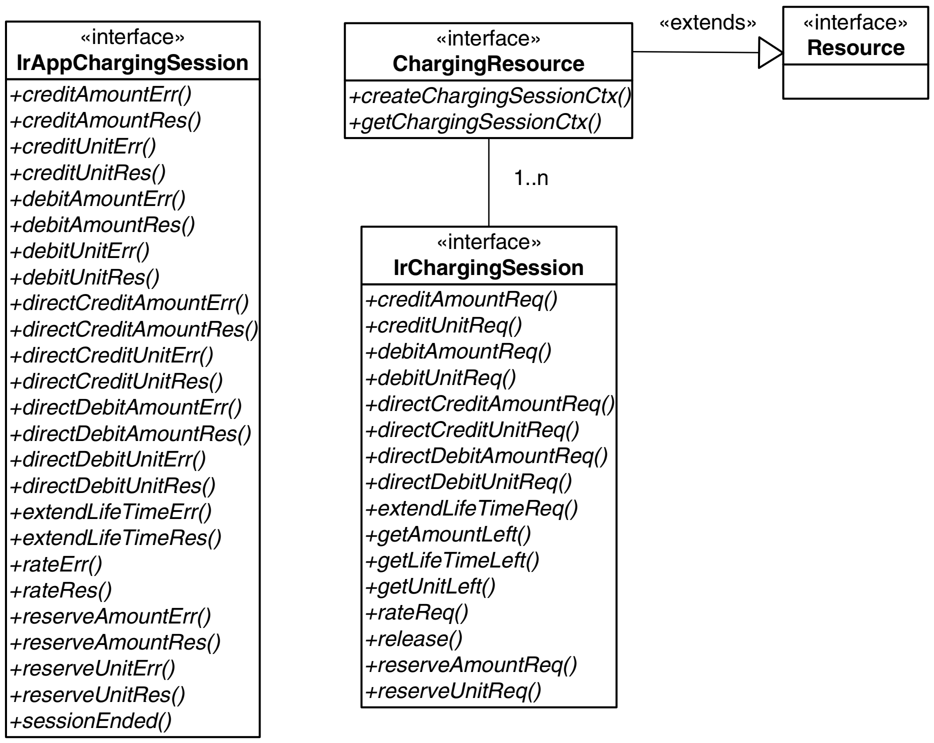
- Network plug-in
- Use cases for Content based charging

Network plug-in

Content based charging interfaces

All subscriber profile interfaces are defined in the file bea\wlng21\esdk\idl\plugin_if\charging\ChargingResource_IF.idl

Figure 11-1 Content based charging interface



Interfaces that should be implemented by the plug-in are listed in the table below.

Table 11-1 Interfaces that shall be implemented by a plug-in for Content based charging

Interface	Description
Resource	Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.
ChargingResource	Manager object that is used to create sessions towards a Content based charging plug-in.
IrChargingSession	The Charging session interface provides functions to reserve, debit and credit user accounts.It also provides functions for rating requstes.There are separate functions that operate on units, volumes and amounts.

Interfaces that are implemented by the ESPA Content based charging SC.

Table 11-2 Interfaces that are implemented by the ESPA Content based charging SC, used by a Content based charging plug-in.

Interface	Description
IrAppChargingSession	Call back interface for receiving responses to operations performed via the IrChargingSession interface.

ChargingResource

Table 11-3 ChargingResource

Method	Description
createChargingSessionCtx	Creates a Content based charging session.
getChargingSessionCtx	Retrieves an already created instance of a charging session.

IrChargingSession

Table 11-4 IrChargingSession

Method	Description
creditAmountReq	Credits an amount towards the reservation associated with the session.
creditUnitReq	Credits one or more units towards the reservation associated with the session.
debitAmountReq	Debits an amount towards the reservation associated with the session.
debitUnitReq	Debits one or more units towards the reservation associated with the session.
directCreditAmountReq	Credits an amount directly without affecting the reservation.
directCreditUnitReq	Credits one or more units directly without affecting the reservation.
directDebitAmountReq	Debits an amount directly without affecting the reservation.

Method	Description
directDebitUnitReq	Debits one or more units directly without affecting the reservation.
extendLifeTimeReq	Request to extend the lifetime of a reservation.
getAmountLeft	Request to get the remaining amount in a reservation.
getLifeTimeLeft	Request to get the remaining lifetime of a reservation.
getUnitLeft	Request to get the remaining units of a reservation.
rateReq	Request to rate a request.
release	Releases a Content based charging session and releases any reserved amount or units left in a reservation.
reserveAmountReq	Reserves an amount from an account
reserveUnitReq	Reserves one or more units from an account.

IrAppChargingSession

Table 11-5 IrAppChargingSession

Method	Description
creditAmountErr	Result of a failed creditAmountReq request. No amount was credited.
creditAmountRes	Result of a successful creditAmountReq request. Contains the information requested.
creditUnitErr	Result of a failed creditUnitReq request. No units were credited.
creditUnitRes	Result of a successful creditUnitReq request.
debitAmountErr	Result of a failed debitAmountReq request. No amount was debited.
debitAmountRes	Result of a successful debitAmountReq request.
debitUnitErr	Result of a failed debitUnitReq request. No units were debited.
debitUnitRes	Result of a successful debitUnitReq request.
directCreditAmountErr	Result of a failed directCreditAmountReq request. No amount was debited.

Method	Description
directCreditAmountRes	Result of a successful directCreditAmountReq request.
directCreditUnitErr	Result of a failed directCreditUnitReq request. No units were credited.
directCreditUnitRes	Result of a successful directCreditUnitReq request.
directDebitAmountErr	Result of a failed directDebitAmountReq request. No amount was credited.
directDebitAmountRes	Result of a successful directDebitAmountReq request.
directDebitUnitErr	Result of a failed directDebitUnitReq request. No units were debited.
directDebitUnitRes	Result of a successful directDebitUnitReq request.
extendLifeTimeErr	Result of a failed extendLifeTimeReq request. The lifetime was not extended.
extendLifeTimeRes	Result of a successful extendLifeTimeReq request.
rateErr	Result of a failed rateReq request. The rating was not performed.
rateRes	Result of a successful rateReq request.
reserveAmountErr	Result of a failed reserveAmountReq request. No amount was reserved.
reserveAmountRes	Result of a successful reserveAmountReq request.
reserveUnitErr	Result of a failed reserveAmountReq request. No units were reserved.
reserveUnitRes	Result of a successful reserveAmountReq request.
sessionEnded	The charging session was ended by the plug-in.

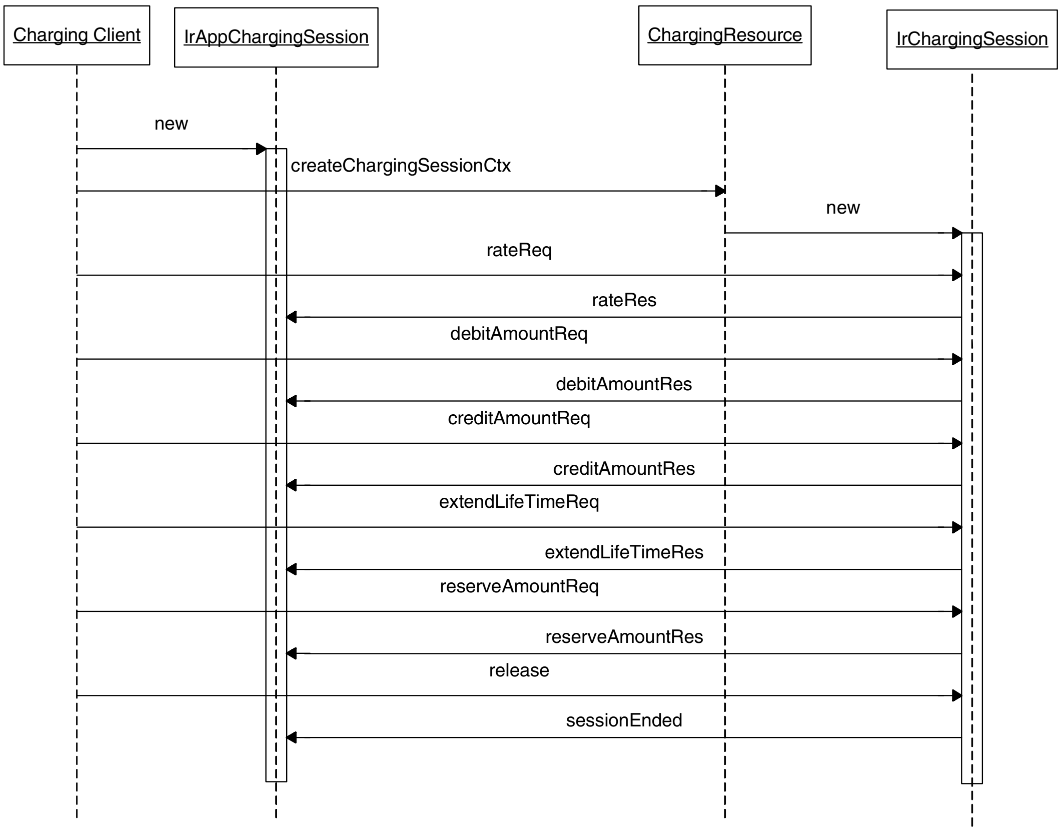
Use cases for Content based charging

A Content based charging plug-in only supports operations originating from an application. Network triggered is not supported.

Application-initiated usage of a Content based charging plug-in

The following sequence diagram show a basic interaction between a user of content based charging plug-in and the plug-in.

Figure 11-2 Application-initiated usage of Content based charging plug-in.



Details about the sequence diagram:

- Prior to this sequence the Charging client (for example the content based charging ESPA SC) has received a plug-in. See "General usage (application-initiated events)" on page 2-15.

- The Charging client application performs a createChargingSessionCtx to the implementation of the ChargingResource interface. This interface is implemented in the plug-in.
- The plug-in creates an instance of the IrChargingSession interface and return it to the client. An identifier for the session, along with the interface is returned to the client. A sequence number is also supplied. This sequence number shall be increased for each new request, in order to make sure that the requests are performed in the correct order.
- The Content based charging client orders the plug-in to rate the request. The plug-in returns the rating via rateRes, so the client knows how much to reserve. if the rating failed, rateErr shall be returned.
- The reservation is performed via reserveAmountReq.
- A successful reservation is reported via reserveAmountRes.
- An amount tis debited from the reservation using debitAmountReq.
- The successful debit is reported via debitAmountRes.
- The debited amount was too large, so the account is credited using creditAmountReq, and the successful credit is reported using creditAmountRes.
- The lifetime of the reservation is extended via extendLifeTimeReq, and the successful request is reported via extendLifeTimeRes.
- When all apportions in the session have been performed, the session is released.

Subscriber profile

The following sections contain descriptions of plug-ins of Subscriber profile type:

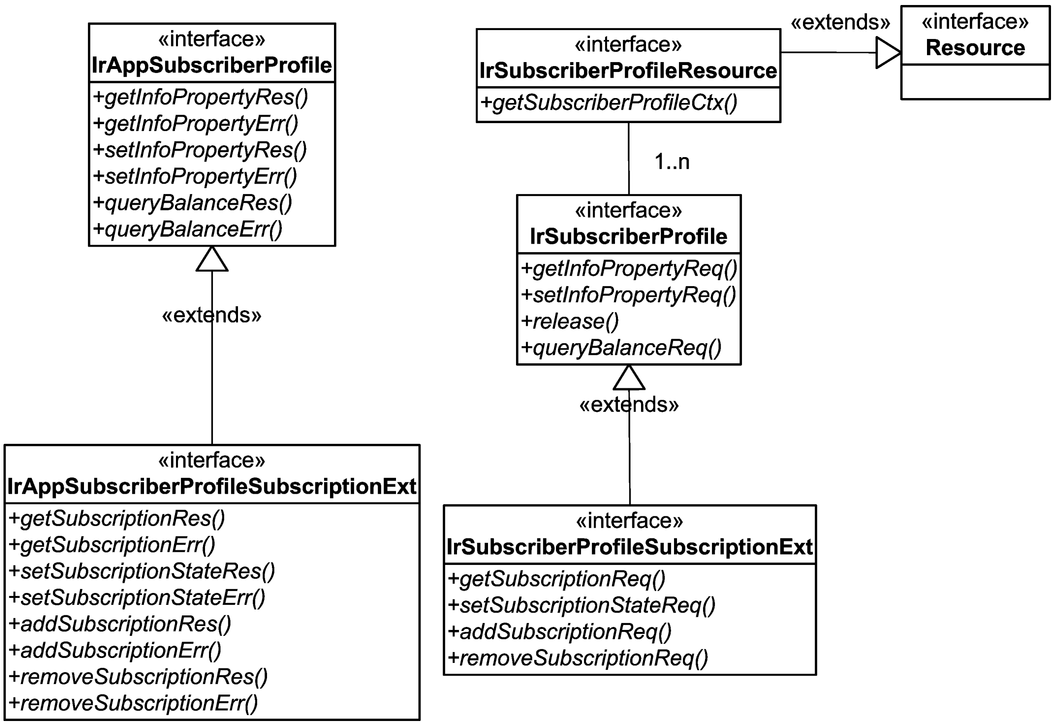
- Network plug-in
- Use cases for Subscriber profile

Network plug-in

Subscriber profile interfaces

All subscriber profile interfaces are defined in the file bea\wlng21\esdk\idl\plugin_if\sp\sp_interfaces.idl

Figure 12-1 Subscriber profile interface



Interfaces that should be implemented by the plug-in are listed in the table below.

Table 12-1 Interfaces that shall be implemented by a plug-in for Subscriber profile

Interface	Description
Resource	Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.
IrSubscriberProfile	Manager object that is used to create and release sessions towards the subscriber profile plug-in, and to get and set subscriber data.
IrSubscriberProfileSubscriptionExt	The extended subscriber profile interface provides functions for handling subscriptions. The ESPA Subscriber profile SC does not operate on this interface, but other -custom- clients to a subscriber profile plug-in may use this. Examples of this can be policy utility classes.

Interfaces that are implemented by the ESPA Subscriber profile SC.

Table 12-2 Interfaces that are implemented by the Subscriber profile SC, used by a subscriber profile plug-in.

Interface	Description
IrAppSubscriberProfile	Call back interface for receiving responses to operations performed via the IrAppSubscriberProfile interface.

Interfaces that could be implemented by a client to the Subscriber profile plug-in.

Table 12-3 Interfaces that could be implemented by a client to the Subscriber profile plug-in.

Interface	Description
IrAppSubscriberProfileSubscriptionExt	Call back interface for receiving responses to operations performed via the IrSubscriberProfileSubscriptionExt interface.

IrSubscriberProfileResource

Table 12-4 IrSubscriberProfileResource

Method	Description
getSubscriberProfileCtx	Request a subscriber profile. Creates a session.

IrSubscriberProfile

Table 12-5 IrSubscriberProfile

Method	Description
getSubscriberId	Gets a subscriber ID based on the session id.
getInfoPropertyReq	Request to get one or more properties for the subscriber.
setInfoPropertyReq	Request to set one or more properties for the subscriber.

Method	Description
release	Releases a session.
queryBalanceReq	Request to get the available balance for a subscriber.

IrSubscriberProfileSubscriptionExt

Table 12-6 IrSubscriberProfileSubscriptionExt

Method	Description
getSubscriptionReq	Request to fetch data about a subscription.
setSubscriptionStateReq	Request to set data about a subscription.
addSubscriptionReq	Request to add a subscription.
removeSubscriptionReq	Request to remove a subscription.

IrAppSubscriberProfile

Table 12-7 IrAppSubscriberProfile

Method	Description
getInfoPropertyRes	Result of a successful getInfoPropertyReq request. Contains the information requested.
getInfoPropertyErr	Result of a failed getInfoPropertyReq request.
setInfoPropertyRes	Verification of a successful setInfoPropertyReq request.
setInfoPropertyErr	Result of a failed setInfoPropertyReq request.

IrAppSubscriberProfileSubscriptionExt

Table 12-8 IrAppSubscriberProfileSubscriptionExt

Method	Description
getSubscriptionRes	Result of a successful getSubscriptionReq request.
getSubscriptionErr	Result of a failed getSubscriptionReq request.
setSubscriptionStateRes	Result of a successful setSubscriptionStateReq request.
setSubscriptionStateErr	Result of a failed setSubscriptionStateReq request.
addSubscriptionRes	Result of a successful addSubscriptionReq request.
addSubscriptionErr	Result of a failed addSubscriptionReq request.
removeSubscriptionRes	Result of a successful removeSubscriptionReq request.
removeSubscriptionErr	Result of a failed removeSubscriptionReq request.

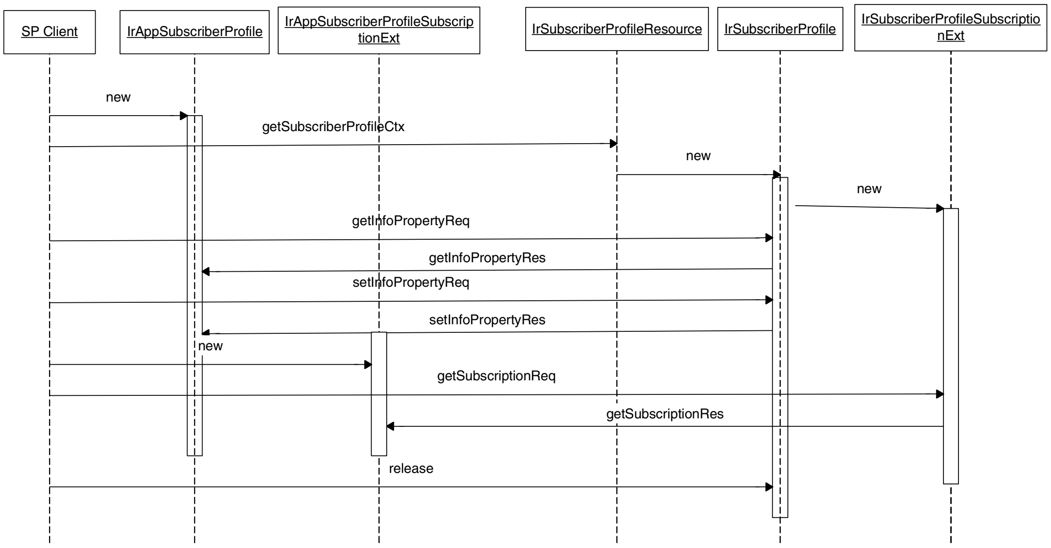
Use cases for Subscriber profile

A subscriber profile plug-in only supports operations originating from an application. Network triggered is not supported.

Application-initiated usage of a Subscriber profile plug-in

The following sequence diagram show a basic interaction between a user of subscriber profile plug-in and the Plug-in.

Figure 12-2 Application-initiated usage of Subscriber profile plug-in.



Details about the sequence diagram:

- Prior to this sequence the Subscriber profile client (for example the subscriber profile ESPA SC) has received a plug-in. See "General usage (application-initiated events)" on page 2-15.
- The Subscriber profile client application performs a getSubscriberProfileCtx to the implementation of the IrSubscriberProfile interface. This interface is implemented in the plug-in. An identifier for the session, along with the interface is returned to the client.
- The client request to get information about the subscriber via getInfopropertyReg. The client defines which properties of a subscriber to be fetched from the plug-in and the result is returned asynchronously to the IrAppSubscriberProfile interface implemented by the Subscriber profile plug-in client.
- The requested data is provided by the plug-in via the call getInfoPropertyRes. If the plug-in experiences an error when getting the requested properties, the method getInfoPropertyErr is invoked instead. The data is given as name-value pairs.
- The Subscriber profile plug-in sets data in the plug-in via setInfoPropertyReq. The data is provided as name-value pairs.

- The result of the setInfoPropertyReq request is returned asynchronously to the IrAppSubscriberProfile interface implemented by the Subscriber profile plug-in client via setInfoPropertyRes. If the plug-in experiences an error when setting the requested properties, the method setInfoPropertyErr is invoked instead.
- Subscription information can also be fetched and defined using the IrSubscriberProfileSubscriptionExt interface.
- The Subscriber profile client retrieves information on which subscriptions the subscriber has by calling getSubscriptionReq.
- The response is sent back asynchronously from the plug-in using getSubscriptionRes on the IrAppSubscriberProfileSubscriptionExt interface implemented by the Subscriber profile client. If the plug-in experienced problems, getSubscriptionErr is invoked instead.
- When the interaction with the subscriber profile plug-in is finished, the Subscriber profile client invokes release, and the session is destroyed. It is the responsibility of the plug-in to remove all objects.

User Location

The following sections contain descriptions of plug-ins of User location type:

- Network plug-in
- Use cases for user location

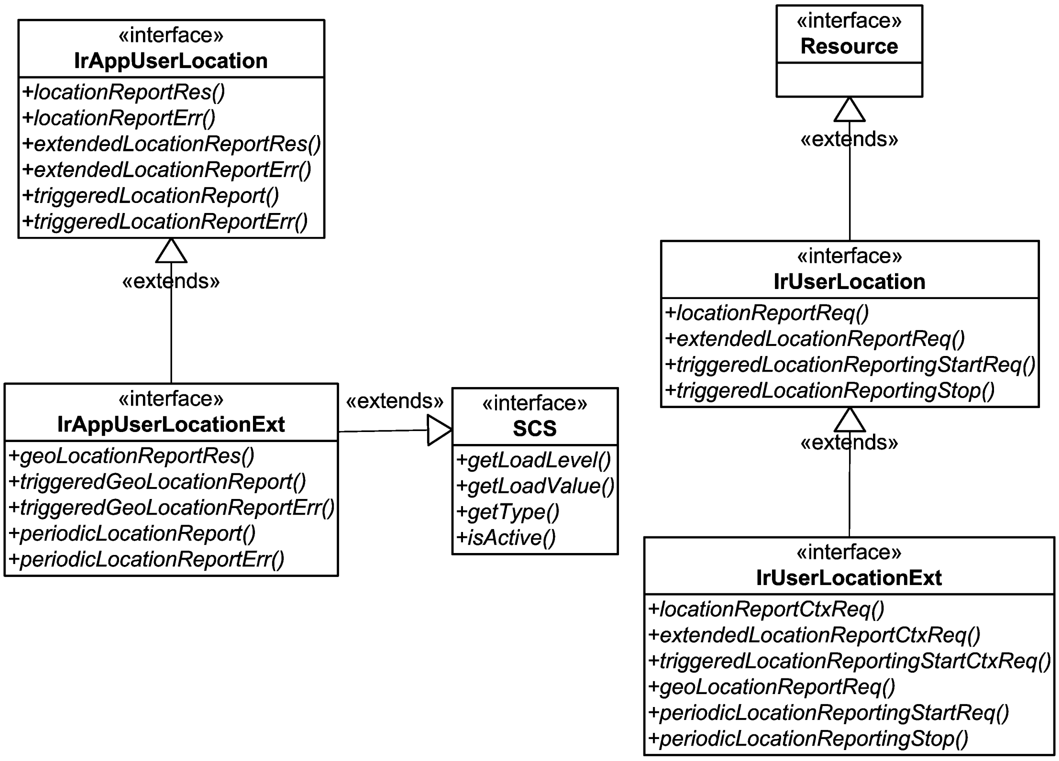
Network plug-in

User location interfaces

All User location interfaces are defined in the file UlResource_IF.idl in bea\wlng21\esdk\idl\plugin_if\mobility.

The interfaces use definitions in the files UlResource_data.idl and MobilityResource_data.idl.

Figure 13-1 User location plugin interface


«interface»
IrAppUserLocation
+locationReportRes()
+locationReportErr()
+extendedLocationReportRes()
+extendedLocationReportErr()
+triggeredLocationReport()
+triggeredLocationReportErr()
«extends»
«interface»
IrAppUserLocationExt
+geoLocationReportRes()
+triggeredGeoLocationReport()
+triggeredGeoLocationReportErr()
+periodicLocationReport()
+periodicLocationReportErr()
«extends»
«interface»
SCS
+getLoadLevel()
+getLoadValue()
+getType()
+isActive()
«interface»
Resource
«extends»
«interface»
IrUserLocation
+locationReportReq()
+extendedLocationReportReq()
+triggeredLocationReportingStartReq()
+triggeredLocationReportingStop()
«extends»
«interface»
IrUserLocationExt
+locationReportCtxReq()
+extendedLocationReportCtxReq()
+triggeredLocationReportingStartCtxReq()
+geoLocationReportReq()
+periodicLocationReportingStartReq()
+periodicLocationReportingStop()

Interfaces that should be implemented by the plug-in are listed in the table below.

Table 13-1 Interfaces that shall be implemented by a plug-in for user location

Interface	Description
Resource	Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.
IrUserLocation	The user location plug-in interface provides functions for getting the position for a terminal, starting triggered and period location request.
IrUserLocationExt	The extended user location plug-in interface provides functions for getting the position for a terminal, starting triggered and period location request.

Interfaces that are implemented by the user location SC.

Table 13-2 Interfaces that are implemented by the user location SC, used by a user location plug-in.

Interface	Description
IrAppUserLocation	Call back interface for positioning results.
IrAppUserLocationExt	Extended call back interface for positioning results.

IrUserLocation

IrUserLocation inherits from the Resource interface and contains and adds an method for using ordering positioning requests.

Table 13-3 IrUserLocation

Method	Description
locationReportReq	Deprecated.
extendedLocationReportReq	Deprecated.
triggeredLocationReportingStart Req	Deprecated.
triggeredLocationReportingStop	Stops a previously started triggered location report request.

IrUserLocationExt

IrUserLocationExt is an extended version of the IrUserLocationExt interface and should be used instead of this.

Table 13-4 IrUserLocationExt

Method	Description
locationReportCtxReq	Request of a report on the location of one or several terminals
extendedLocationReportCtxReq	Request of a report on the location of one or several terminals. More information, such as altitude, can be provided in the response to the request that in the response to locationReportCtxReq.

Method	Description
triggeredLocationReportingStartCtxReq	Request for user location reports when the location of a terminal is changed, so it enters or exits a specific location. The reports are triggered by location change.
geoLocationReportReq	Request of report on the location for one or several terminals where the result is delivered as geographical data such as address, zip code and so on.
periodicLocationReportingStartReq	Request to start a periodic report on the location for one or several terminal. The desired interval between positioning reports is defined.
periodicLocationReportingStop	Request to stop a previously started periodic report.

IrAppUserLocation

IrAppUserLocation listens for events originating in the plug-in. There may be several listeners registered for each plug-in.

This interface inherits from the SCS interface. It makes it possible to narrow a SC object retrieved from the SC Manager to an IrAppUserLocation.

Table 13-5 IrAppUserLocation

Method	Description
locationReportRes	Delivers a report containing locations for one or several terminals to the Service Capability implementation.
locationReportErr	Informs the Service Capability implementation that a location report request has failed.
extendedLocationReportRes	Delivers a report containing extended location information about one or several users to the Service Capability implementation.
extendedLocationReportErr	Informs the Service Capability implementation that an extended location report request has failed.
triggeredLocationReport	Delivers a report containing triggered location information about one or several terminals to the Service Capability implementation.
triggeredLocationReportErr	Informs the Service Capability implementation that a triggered location report request has failed.

IrAppUserLocationExt

IrAppUserLocationExt listens for events originating in the plug-in. There may be several listeners registered for each plug-in. This interface extends the IrAppUserLocation interface and should be used instead of that.

This interface inherits from the SCS interface. It makes it possible to narrow a SC object retrieved from the SC Manager to an MessageListener.

Table 13-6 IrAppUserLocationExt

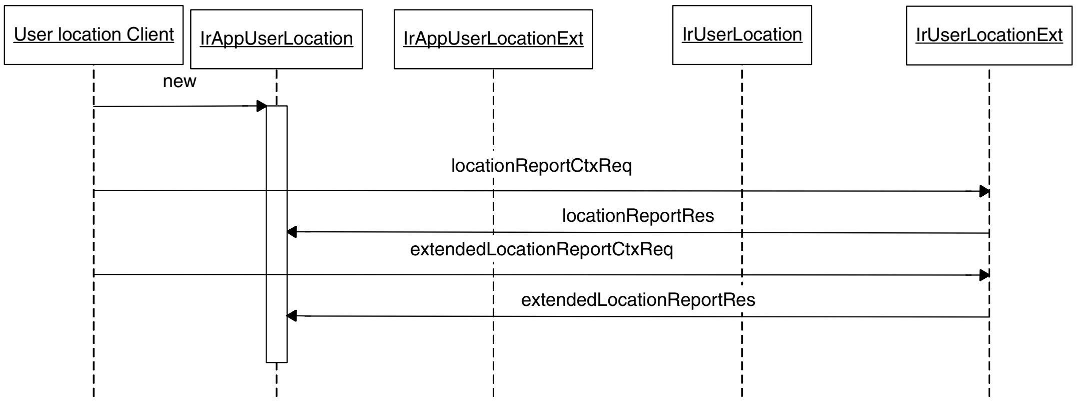
Method	Description
geoLocationReportRes	Delivers a report containing locations for one or several terminals.
triggeredGeoLocationReport	Delivers a report containing geographical information on the position for one or several terminals. The position data is geographical data such as address, zip code and so on.
triggeredGeoLocationReportErr	Informs the Service Capability implementation that a triggered geographical location report request has failed.
periodicLocationReport	Delivers a periodic report containing locations for one or several terminals.
periodicLocationReportErr	Informs the Service Capability implementation that a periodic location report request has failed.

Use cases for user location

Application-initiated user location

The following sequence diagram show a basic positioning request and an extended user location request between a User location client (for example ESPA user location SC) and a plug-in.

Figure 13-2 Application-initiated send message



Details about the sequence diagram:

- Prior to this sequence the user location client has received a plug-in. See "General usage (application-initiated events)" on page 2-15.
- The MessagingResourceClient application performs a locationReportCtxReq call to the implementation of the IrUserLocationExt interface. This interface is implemented in the plug-in. The call-back interface, IrAppUserLocation, is provided in call.
- The plug-in reports the location via locationReportRes. If the plug-in experienced problems getting the position, locationReportErr is used.
- The MessagingResourceClient application performs a extendedLocationReportCtxReq call to the implementation of the IrUserLocationExt interface. This interface is implemented in the plug-in. The call-back interface, IrAppUserLocation, is provided in call. The response to extendedLocationReportCtxReq provides can provide more detailed information such as altitude.
- The plug-in reports the location via extendedLocationReportRes. If the plug-in experienced problems getting the position, extendedLocationReportErr is used.

Network-triggered user location request

The current implementation of the ESPA User location SC, does not support network triggered events. A new user location implementation can however benefit from the network triggered parts of the user location plug-in interface.

The network triggered part of the user location uses the IrUserLocationExt interface, and starts a triggered location report using triggeredLocationReportingStartCtxReq. Supplied with this request is the call-back interface and the area of interest, together with information on wether the information shall be supplied when entering or leaving this specified area.

Note that the area can be either an area defined by longitude and latitudes or as an abstract geographical area, such as an a street or city.

The responses to the triggered user location request shall be sent to triggeredGeoLocationReport, defined in the IrAppUserLocationExt interface, or triggeredLocationReport, defined in the IrAppUserLocation interface.

CHAPTER 14

Policy rules and Policy Utilities

The following sections contain descriptions of Policy Rules and Policy Utilities:

- Mapping policy request data to variables in a Policy Rule
- Adding a rule to Policy Decision Point
- Defining a Policy Utility class
- Example Policy Utility

Mapping policy request data to variables in a Policy Rule

The policy request data is put in a PolicyRequest object which is sent to the Policy Service for evaluation. The data in the PolicyRequest object can be used from the rules evaluating the request.

The PolicyRequest object is created and sent to the rules engine in the Policy Enforcement Point (PEP).

For information on how a PEP is implemented is described in "PolicyManager" on page 3-7.

In the PEP, the PolicyRequest object, can be populated with the following standard data as described below:

- applicationID, the application ID of the requesting party.
- serviceProviderID, the Service Provider ID of the requesting party.
- nodeID, ignore this parameter.

- serviceName, the name of the software module from where the policy request originates. Used in the rules to check service contracts in the SLAs and for look-up of rules specific for the service.
- methodName, the name of the method which is evaluated.
- serviceGroup, must be “espa”.
- serviceCode, the service code provided by the application.
- requesterID, the service code provided by the application.

All of the above are Strings.

- transactionID, ignore this parameter.
- noOfActiveSessions, ignore this parameter.
- timeStamp, the time stamp when the request was fed to the Policy Engine. Use SLEE Time manager to get this timestamp.
- reqCounter, insert the number of target addresses in the request. If only one target address is used in the request set this value to 1. If using multiple target addresses in the request, use the number of target addresses.

All of these are Long values.

In addition to these standard values, it is possible to add AdditionalParameters, which consists of an array of AdditionalDataValue. This datatype provides a mechanism for the transferring other request data that the predefined to the Policy Decision Point. AdditionalDataValue consist of a name-value pair, where different types of values can be defined in the value part. The following datatypes can be defined in an AdditionalDataValue object.

- intValue(int val), for integer values.
- longValue(long val), for long values.
- stringValue(String val), for Strings.
- stringArrayValue(String[] val), for Arrays of String values.
- booleanValue(boolean val), for boolean values.
- shortValue(short val), for short values.
- charValue(char val), for char values.

- floatValue(float val), for float values.
- doubleValue(double val), for double values.
- intArrayValue(int[] val), for arrays of int values.

The name of the name-value pair is defined in the `dataName` member variable in the AdditionalData object.

Listing 14-1 Defining AdditionalData

```
AdditionalData adArray[] = new AdditionalData[1];

AdditionalDataValue targetAddressValue = new AdditionalDataValue();

AdditionalData adTargetAddressString = new AdditionalData();

targetAddressValue.stringValue(address);

adTargetAddressString.dataName = "targetAddress";

adTargetAddressString.dataValue = targetAddressValue;

adArray[0] = adTargetAddressString;

policyRequest.additionalParameters = adArray;
```

See Adding a rule to Policy Decision Point for information on how to use the PolicyRequest object in the rule.

Adding a rule to Policy Decision Point

The first thing to do when adding a rule to a Policy Decision Point is to define the name of the rule and to defined the priority of the rule. There are a set of pre-defined priority levels, which are mapped to a numerical value:

- minimum, where the value is $-1*10^9$
- low, where the value is $-1*10^6$
- high, where the value is $1*10^6$
- maximum, where the value is $1*10^9$

Rules with high priority are evaluated prior to rules of low priority.

Listing 14-2 Skeleton of a rule

```
rule DenySubscriberNotExists
{
priority = high;
  when
  {
   // fetch the policy request data and perform evaluations.
  }
then
  {
      // Take action on
  }
};
```

Getting data defined in the PolicyRequest

The PolicyRequest object that was sent to the rule engine can be retrieved in the rules.

The standard requests data is found in the rules via the same names as they are defined in the PolicyRequest object created in the PEP. Below is an example on how the rule assigns the PolicyRequest member variable serviceName to the rule variable sname via the PolicyRequest object. The rule object pr is assigned to the PolicyRequest object.

Listing 14-3 Policy Request data is fetched

```
?pr: event PolicyRequest(?sname: serviceName);
```

When the policy rule has evaluated the request and the decision is to deny the request, the rule's representation of the PolicyRequest object must be retracted. Retracting the PolicyRequest object aborts further rule enforcement.

Listing 14-4 Retract a request

```
retract (?pr);
```

The requests must also be retracted for allowed requests in the last rule of the execution flow. This could be achieved by adding a general finalizing allow rule that retracts the request. This rule should have priority minimum.

Listing 14-5 General finalizing allow rule that retracts a request

```
rule AllowServiceRequest
{
  priority = minimum;
  when
  {
       ?pr: event PolicyRequest();


  }
  then
  {
       retract (?pr);
       ?pr.allow();


  }
};
```

To fetch data defined as AdditionalValues, the data is fetched by name according to the example below. In the example the AdditionalValue named targetAdress is stored in the variable adddDataValue. The PolicyRequest object is pr.

Listing 14-6 Fetching AdditionalValue data

```
bind ?addDataValue = ?pr.getAdditionalDataStringValue("targetAddress");
```

Depending on the type, the data is fetched via different methods:

- getAdditionalDataIntValue(...), for int values.
- getAdditionalDataLongValue(...), for long values.
- getAdditionalDataStringValue(...), for String values.
- getAdditionalDataStringArrayValue(...), for arrays of String values.
- getAdditionalDataBooleanValue(...), for boolean values.
- getAdditionalDataShortValue(...), for short values.
- getAdditionalDataCharValue(...), for char values.
- getAdditionalDataFloatValue(...), for float values.
- getAdditionalDataDoubleValue(...), for double values.
- getAdditionalDataIntArrayValue(...), for arrays of int values.

If the datatype is unknown, it can be determined by invoking the discriminator method on the AdditionalDataValue object.

Listing 14-7 Determine the type of an AdditionalDatavalue

```
bind ?type = ?pr.getAdditionalData.dataValue.discriminator().value();
```

Where type is one of the following:

- AdditionalDataType._P_ADDITIONAL_INT
- AdditionalDataType._P_ADDITIONAL_LONG
- AdditionalDataType._P_ADDITIONAL_STRING
- AdditionalDataType._P_ADDITIONAL_STRING_ARRAY
- AdditionalDataType._P_ADDITIONAL_BOOLEAN
- AdditionalDataType._P_ADDITIONAL_SHORT
- AdditionalDataType._P_ADDITIONAL_CHAR
- AdditionalDataType._P_ADDITIONAL_FLOAT
- AdditionalDataType._P_ADDITIONAL_DOUBLE
- AdditionalDataType._P_ADDITIONAL_INT_ARRAY

Also see the JavaDoc for PolicyRequest.

Extending Service Level Agreements

Service Level Agreements (SLAs) are XML files that contains data which is enforced by Policy rules. The Service Level Agreements are created and loaded into the Policy Engine on these levels:

- Service provider
- Application
- Service provider traffic
- Total traffic

There are separate Policy rules that enforces these SLAs, one on service provider level and one on application level. The traffic SLAs are both enforced using one rule.

When extending Service Level Agreements, the following steps must be taken:

- The SLA schema file must be updated.
- The SLA schema file must be loaded into the Policy service.
- The rule files that operates on the data must be updated and loaded.

- The SLAs must be updated
- The updated SLAs must be loaded

Update SLA Schema

The SLA schema files are located in

`<installation directory>/bin/policy/sla_schema.`

There are three different SLA schema files:

- `app_sla_file.xsd`, which defines the SLA schema for the application level SLAs.
- `sp_sla_file.xsd`, which defines the SLA schema for the service provider level SLAs.
- `node_sla_file.xsd`, which defines the SLA schema for the service provider traffic and total traffic SLA.

Update the schema file with the new element. For example, if the service provider SLA schema needs a new element that defines a String value stated in the service provider SLAs as <additionalData>mydata</additionalData>, update the serviceContract in the schema file with the following element:

<xs:element name="additionalData" minOccurs="0" maxOccurs="1" type="xs:int"/>

Load new SLA schema into the Policy Service

Load the SLA schema into the Policy service using the Network Gatekeeper Management Tool using the following methods in the **Policy** service:

reloadApplicationXmlDriver, for reloading application level SLA schemas.

reloadServiceProviderXmlDriver, for reloading service provider level SLA schemas.

reloadNodeXmlDriver, for service provider traffic and total traffic SLA schemas.

Update and load rule files

In order to enforce the SLA data in, the rule files have to be updated with rules that enforce the new data in the SLA.

The data in the SLA is fetched from an object model the Policy engine creates from the data defined in the SLAs. The data in an SLA is fetched from the Policy Rules by name. For example, if a a tag in the SLA is <additionalData>, the data is fetched using the same name, as described below.

The rule gets the parameter aParam from the Policy request object, and puts in the local variable pr. The parameter aParam is compared with the data fetched from the SLA, and denies the request if the parameter given in the SLA is larger than the parameter provided in the Policy request.

Listing 14-8 Get SLA data and compare with a parameter in a Policy request

```
rule denyAParamValueNotAllowed
{
  priority = high;
  when
  {
       ?pr: event PolicyRequest(?serviceName: serviceName;
                                ?aParam: aParam);
       ?sc: ServiceContract(?scs: scs;
                            ?scs equals ?serviceName;
                            ?aParam > additionalData);
}
  then
  {
       retract (?pr);
       ?pr.deny("The parameter is not allowed!");
  }
};
```

The rules must be loaded into the Policy engine, this is performed using the Network Gatekeeper Management Tool using the following methods in the **Policy** Service:

loadApplicationRules, for loading application level rules.

loadServiceProviderRules, for loading service provider level rules.

loadNodeRules, for loading service provider traffic and total traffic rules.

Update SLAs

The SLAs needs to be updated with the tag and the data in the new tag. If extending the SLA with the tag defined in the schema file, update the service provider SLA with the tag and data:

<additionalData>mydata</additionalData>

Load new SLAs

The SLAs holding the new parameters must be loaded in to the Policy engine. See Network Gatekeeper User's guide for information on how to load a new SLAs.

Using a Policy Utility

A Policy Utility is a Java object that is used from a rule.

The rule language, Ilog IRL, makes it possible to perform basic evaluation and parameter substitution, but for more complex processing, call-outs to Java object might be necessary. Java objects can also be used for interaction with external systems such as databases, or prepaid systems.

Below is an example of how a Policy Utility object is called from a rule file.

Listing 14-9 Invoking a Policy Utility from a rule

```
if (MyPolicyUtility.getInstance().subscriberExists(?address,
                                                   ?sp,
                                                   ?app,
                                                   ?reqID,
                                                   ?sCode) == false) {
    retract (?pr);
    ?pr.deny("The Subscriber " + ?address + " is not in database.");
}
```

In the example above, the rule calls the method subscriberExists in the class MyPolicyUtility.

Defining a Policy Utility class

A Policy utility class has the following set of characteristics:

- It executes as a SLEE service
- It is a Singleton class

A Policy Utility class executes as a singleton class in order to be callable from the rule. The Policy Utility registers its classes in the Policy service, where they are instantiated and executed. As a consequence of the classloader hierarchy, there is a need of two consecutive restarts of the SLEE in which the Policy utility is installed in order to load the Policy Utility -one for the Policy utility to install itself in the SLEE and one to load the Policy Utility into the Policy Service.

When the Policy Utility is instantiated, it registers itself into Policy service. The constructor shall be private since it is a singleton class. A static public class is created to get the Policy Utility Class from the rule, and to instantiate the class if necessary.

Since the Policy Utility class will execute in the Policy service, it fetches the service context for the Policy via reflection.

Listing 14-10 Registering the Policy Utility class

```
Class internalPolicyContextClass =
Class.forName("com.incomit.policy.InternalPolicyContext");
java.lang.reflect.Method getServiceContextMethod =
internalPolicyContextClass.getMethod("getServiceContext", null);
java.lang.Object serviceContextResultObj =
getServiceContextMethod.invoke(null, null);
m_sc = (ServiceContext) serviceContextResultObj;
```

The Policy Utility will also use the same POA as the Policy service. The POA is only necessary when the Policy Utility creates new CORBA objects, for example listener objects when connecting to a plug-in.

Listing 14-11 Getting the POA in a Policy Utility Class

```
java.lang.reflect.Method getChildPOAMethod =

internalPolicyContextClass.getMethod("getChildPOA", null);

java.lang.Object poaResultObj = getChildPOAMethod.invoke(null, null);

m_poa = (org.omg.PortableServer.POA) poaResultObj;
```

The rule calls a method in the singleton class via a static method that checks if that the class is already instantiated, instantiates it if necessary, and returns the object. In the example in Listing 14-9 this method is named getInstance().

Example Policy Utility

In the example template modules in the Extension SDK, there is an example of a Policy Utility in the directory module_templates\policy_utility.

The Policy Utility is invoked from a rule and it checks if the address parameter is present is a subscriber database.

The subscriber database is interfaced using a Subscriber profile plug-in, and the Policy Utility uses the Plug-in Manager to get the plug-in.

CHAPTER 15

Using the Extension SDK templates

The Extension SDK for WebLogic Network Gatekeeper consists of template source code and build tools that assists the creation of WESPA modules, SESPA modules, ESPA service capabilities, and network plug-ins that execute in WebLogic Network Gatekeeper. It also contains examples of a Policy utility class that can be used from Policy rules.

The template source code illustrates how a request from an application is propagated through the internal layers of the Network Gatekeeper down to the plug-in. It also illustrates how a network initiated request is propagated from the plug-in, through the different layers and up to an application. The network initiated part of the templates also illustrates how the application registers to listen to network initiated events.

The following sections contain descriptions of the Extension SDK templates:

- Files for the SLEE service part of the network plug-in interfaces
- Introduction to the ESPA service capability
- Files for the ESPA service capability interfaces
- Files for the ESPA service capability implementation
- Files for the Policy utility
- Introduction to the SESPA module
- Files for the SESPA module interface
- Files for the SESPA module implementation
- Introduction to the WESPA module
- Files for the WESPA module interface
- Files for the WESPA module implementation
- Introduction to the test application
- Files for the test application
- Introduction to the network simulator
- Files for the network simulator application
- Preparing the development environment
- Using the templates from Eclipse

Prerequisites

Understanding of the network plug-in, ESPA service capability module, SESPA module, and WESPA module concepts as outlined in WebLogic Network Gatekeeper Product Description.

All template source code is written in Java and some modules use CORBA, so experience with Java and CORBA is essential.

The build environment is based upon the Ant build tool, so experience with Ant and Ant build files is necessary.

Orbacus 4.1.2 or 4.3 must be installed.

Java 2 SDK 1.4.2 must be installed.

It is necessary to have working knowledge of how to handle the WebLogic Network Gatekeeper.

When deploying software modules created with the Extension SDK, the software modules must be deployed to a WebLogic Network Gatekeeper 2.1 with the following patches installed:

- x_sespa_access.jar, patch version: R_WLNG_2_1_0_4
- slee.jar, patch version: R_WLNG_2_1_0_5

Installing the Extension SDK

Installation prerequisites

The following prerequisites must be fulfilled before starting to install an Extension SDK:

- Access to the product CDs or access to the Download Center.
- A password for extracting the installation file. The password was provided when you ordered Extension SDK for Network Gatekeeper
- The Java SDK for the platform must be downloaded and installed.
- Orbacus 4.1.2 or 4.3 must be installed.

Installation procedure

1. Download the Extension SDK software from the Download Center, or copy the file from the product CD, to the local file system.
2. The Extension SDK is installed by extracting the ZIP file `esdk21_wlng21.zip` to the file system. The file is extracted to the directory `bea\wlng21\esdk`. The directory structure described in "Directory structure for the templates" on page 15-15 is created.

Note: A password is required to extract this file. The password was provided when you ordered Extension SDK for Network Gatekeeper and can also be obtained from the BEA eLicense system.

3. Add the path to the Orbacus binary directory in the path environment variable. The Orbacus binaries are located in `<Orbacus installation path>\bin`.
4. Define the `ANT_HOME` environment variable to the Ant directory provided by the Extension SDK. The ant directory is located in `<install path to Extension SDK>\bea\wlng21\esdk\dev_tools\ant`

5. Add the path to the ant binary directory in the path environment variable. The Ant binaries are located in `<install path to Extension SDK>\bea\wlng21\esdk\dev_tools\ant\bin`.
6. Define the `JAVA_HOME` environment variable to the Java 1.4.2 directory.
7. Add the path to the Java binary directory in the path environment variable.
8. Define the properties described in "Preparing the development environment" on page 15-39.
9. If using Eclipse as a development environment, follow the instructions in "Using the templates from Eclipse" on page 15-43.

About WESPA, SESPA, ESPA service capability, and network plug-in software modules

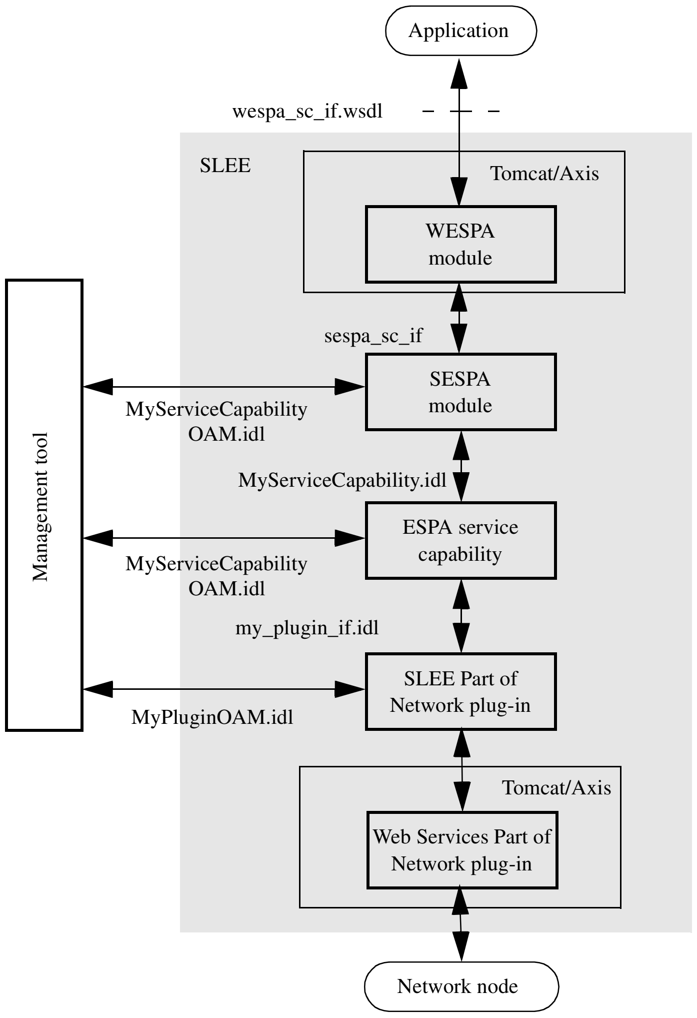
There are some alternatives when extending the WebLogic Network Gatekeeper:

1. Developing a new network plug-in to support a new network node. The corresponding ESPA service capability and above SW modules are already available in WebLogic Network Gatekeeper and providing service capability access through one or more existing network plug-ins.
2. Developing a whole traffic path including network plug-in, ESPA service capability, SESPA module, and WESPA module to provide a new service capability towards the applications.
3. Developing an application facing interfaces that uses an existing SESPA module.

An overview of the WebLogic Network Gatekeeper software architecture supporting the above extension alternatives is shown in Figure 15-1.

Figure 15-1 Software module and interface overview

.



WebLogic Network Gatekeeper is designed around a layered architecture model. This is reflected in the Extension SDK.

The southern most layer, the plug-in layer, contains logic for communication with underlying network systems. The interface between the ESPA Service Capability and the plug-in, the plug-in interface, is an asynchronous CORBA interface used for communication with the service capability layer, and is defined in my_plugin_if.idl. The south bound interface of a plug-in is undefined and depends on the underlying network system. Each plug-in specifies it's type (for example CALL_CONTROL) and it's supported address plans (for example E164). The type can be customized and could be any string. Based on these criteria, plug-ins are accessed from the service capability layer.

The middle most layer, the service capability layer, enforces traffic restrictions and policies. On the northbound side of the service capability layer is the CORBA based ESPA interface and the south bound is the plug-in interface. ESPA service capability implementations register themselves in the ESPA access framework from which they can be accessed, ensuring that they can only be accessed after client authenticity has been assured by ESPA access. ESPA service capabilities can for each service request get a a plug-in that the request shall be mapped onto based on plug-in type and address plan as described above.

The Extension SDK can be used to create web services based access interfaces. These interface modules are divided in two, one SESPA part which makes a stateless native Java representation of the ESPA CORBA interface, and one web services part (WESPA) executing in Tomcat/Axis.

The south bound interface of the SESPA layer is hence the ESPA interface. The northbound SESPA interface is defined using plain java classes. WESPA will access SESPA from the web services environment executing in Tomcat/Axis, therefore both SESPA and WESPA must be loaded in the same Java class loader. A utility in the SLEE is used for this purpose, the SLEE CommonLoader. The WESPA services' northbound interface is defined in plain java files, from which WSDL files are generated. A client can use the WSDL files to generate it's own stubs in the language it uses.

About the flow descriptions

The sections:

- "Traffic flow for application initiated requests" on page 15-7
- "Registration flow for network triggered requests" on page 15-9
- "Traffic flow for network triggered requests" on page 15-12

describe the flow through the template software modules provided in the `module_templates` directory in the Extension SDK. Use the decription together with the code provided in the templates.

Traffic flow for application initiated requests

WESPA service capability

For the application initiated requests, the template exposes two methods to an application in the WESPA service capability, myMethod(...) and myMethodWait(...). The method myMethod(...) returns the result asynchronously via a call to the Web Service method myMethodResult(...), or if an error has occurred via myMethodError(...). The corresponding calls are tied together via an assignment ID given by the application in myMethod(...) and provided as parameters in myMethodResult(...) and MyMethodError(...). The methods myMethodResult(...) and MyMethodError(...) are calling a Web Service that shall be implemented by the application.

The method myMethodWait(...) returns the result synchronously via the return value. The WESPA implementation calls the corresponding methods myMethod(...) and myMethodWait(...) exposed by the SESPA Service Capability module. These calls are pure Java calls.

The application must login to the WESPA Access module prior to calling myMethod(...) or myMethodWait(...).

See Network Gatekeeper Developer's Guide for Extended Web Services and Network Gatekeeper API Description for Extended Web Services for information on how to login to WESPA Access.

If the login was successful, a login ticket representing the session is returned. The login ticket must be provided by the application in every consecutive call to the WESPA module. The ticket is provided in the SOAP header and is retrieved by calling the method getCurrentSessionId() in com.bea.wespa.util.SOAPHeaderHandler. The session ID returned is provided as a parameter to the call to both myMethodWait(...) and myMethodWait(...) exposed by the SESPA service capability.

SESPA service capability module

For the application initiated requests, the template exposes two methods to the WESPA module in the SESPA service capability, myMethod(...) and myMethodWait(...). The method myMethod(...) returns the result asynchronously via a call to the WESPA method myMethodResult(...), or if an error has occurred via myMethodError(...). The corresponding calls are tied together via an assignment ID given by an application in myMethod(...) and provided as parameters in myMethodResult(...) and MyMethodError(...). The method myMethodWait(...) returns the result synchronously via the return value. The SESPA implementation calls the method myMethod(...) exposed by the ESPA Service Capability module.

For the method myMethodWait(...) exposed to the SESPA module, the asynchronous method myMethod(...) exposed by the ESPA module is called. The SESPA module waits for the ESPA module to call either the method myMethodResult(...) or myMethodError(...) on the SESPA modules listener interface until returning. The method could also return if the timer associated with the call to myMethodWait(...) expires. It is important to convert any, from an application point of view, synchronous calls to asynchronous as early as possible, in order to block as few threads as possible in the Network Gatekeeper. This means that these conversions should be done in the SESPA layer.

For each traffic method invocation, the SESPA service capability module retrieves the CORBA client session objects from the login ticket. First, the application session object is retrieved using the database helper utility defined in com.incomit.sespa.util.DbHelper. The object is provided in the constructor of MyServiceCapabilityImpl. By calling getApplicationSession(...) the application session object is returned. This object is provided as a parameter in when the traffic method request is propagated to the ESPA service capability module. The ESPA Manager object is also retrieved using the database helper utility. By calling getEspaManager(...) on the database helper utility the correct ESPA Manager object is returned. The Manager object is found based on the login ticket and the SLEE service name the ESPA Service capability module is registered under. The traffic method request is performed on the ESPA Manager object.

ESPA service capability module

For the application initiated requests, the template exposes one asynchronous method, myMethod(...), to the SESPA module. The result of the method myMethod(...) is reported asynchronously via a call to the SESPA callback method myMethodResult(...), or if an error has occurred via myMethodError(...).

The method myMethod(...) calls an internal method, myMethodInternal(...) which implements a Policy Enforcement Point. If the request is allowed, the class MyServiceCapabilityPluginTask is instantiated and provided in the request to scheduleResourceTask(...) on the plug-in manager. When the plug-in manager has found a suitable plug-in and allocated a task, the SLEE will call doTask(...) on MyServiceCapabilityPluginTask. A reference to the plug-in is provided as an argument to doTask(...). In the doTask(...) method, the ESPA service capability module call the method myMethodReq(...) exposed by the plug-in. The result of the call to the method myMethodReq(...) is reported asynchronously to the call back interface implementation MyPluginListener_impl via the methods myMethodRes(...) or myMethodErr(...).

For each traffic method invocation, the SESPA service capability module checks if the ESPA service capability module is suspended and if it is overloaded. If not, it checks if the application session is valid. This is done by calling checkSessionValidity(...) on the ESPA utility class com.incomit.espa.util.ApplicationSessionValidityHelper.session. The Application session object is provided as a parameter in each traffic method invocation.

Network protocol plug-in SLEE service part

For the application initiated requests, the template exposes one asynchronous method, myMethodReq(...), to the ESPA module. The method myMethodReq(...) schedules a new task using scheduleSLEETask(...) on the SLEE Task Manager. As an argument to scheduleSLEETask(...) is an instance of MyMethodReqTask which is the class that will perform the actual protocol-specific communication with the underlying network node. In MyMethodReqTask the method doTask(...) will be invoked when the SLEE Task manager has assigned a new task. In the implementation of doTask(...), the interface to the Web Services part of the plug-in is fetched from the SLEE Common Loader via a call to SleeCommonLoader.getInstance().getObject(OBJ_NETWORK_INTERFACE). The Web Services part of the plug-in is called via myNetworkMethod(...) and a call to the call-back is performed via a call to myMethodRes(...) on the listener object.

Network protocol plug-in Web Services part

In the Web services part of the network plug-in, the communication with the underlying network node takes place.

The SLEE services part of the network plug-in called the method myNetworkMethod(...) in the class NetworkInterfaceImpl. This method class the underlying node via a Web Services call and returns the result to the SLEE services part of the network plug-in.

Registration flow for network triggered requests

The network triggered part of the template is divided into two parts, a registration part where an application registers for network initiated events the traffic flow part where a request reaches the network plug-in and is propagated through the template traffic path up to an application. The registration part is initiated by the application and described below. The traffic path is described in “Traffic flow for network triggered requests” on page 15-12.

WESPA service capability

For the registration for network triggered notifications, the template exposes two methods to an application in the WESPA service capability, enableNetworkTriggeredEvents(...) and disableNetworkTriggeredEvents(...). The method enableNetworkTriggeredEvents(...) returns a ticket identifying the notification. The method disableNetworkTriggeredEvents(...) removes the registration for network initiated notifications. The WESPA implementation calls the corresponding methods enableNetworkTriggeredEvents(...) and disableNetworkTriggeredEvents(...) exposed by the SESPA Service Capability module. These calls are pure Java calls.

The application must login to the WESPA Access module prior to calling enableNetworkTriggeredEvents(...) or disableNetworkTriggeredEvents(...). See the Network Gatekeeper development documentation for information on how to login to WESPA Access. If the login was successful, a login ticket representing the session is returned. The login ticket must be provided by the application in every consecutive call to the WESPA module. The ticket is provided in the SOAP header and is retrieved by calling the method getCurrentSessionId() in com.bea.wespa.util.SOAPHeaderHandler. The session ID returned is provided as a parameter to the call to both enableNetworkTriggeredEvents(...) and disableNetworkTriggeredEvents(...) exposed by the SESPA service capability.

SESPA service capability module

For the registration part for network triggered notifications, the template exposes two methods to the WESPA module in the SESPA service capability, enableNetworkTriggeredEvents(...) and disableNetworkTriggeredEvents(...). When the method enableNetworkTriggeredEvents(...) is invoked, an object representing the notification, NotificationInfoId, is created and a check is performed that there are not other applications that have registered for notifications with the same criteria using getIdenticalNotifications(...) on the helper class MyServiceCapabilityPersistentStorage. If there are identical notification criteria, and the notification is created by the same application, the notification is fetched from the database by calling getNotification(...) on the helper class MyServiceCapabilityPersistentStorage. The listener object is created using the method internalAddNetworkListener(...) and the listener is stored in the database using storeNotification(...) on the helper class MyServiceCapabilityPersistentStorage. An event is broadcasted to other instances of the SESPA module via a call to generateEnableNotificationEvent(...) on the event helper class MyServiceCapabilityEventHelper. The method internalAddNetworkListener(...) creates a listener object from the class MyESPAServiceCapabilityNetworkListener_impl.

ESPA service capability module

When the ESPA Service capability module becomes activated, it registers the callback interface for the communication between the plug-in and the ESPA Service Capability module. The listener object is registered in the SC manger. The plug-in will query the SC manager for a suitable ESPA Service capability when a network triggered event occurs.

For the registration part for network triggered notifications, the template exposes two methods to the SESPA module in the ESPA service capability, addNetworkTriggeredEventListener(...) and removeNetworkTriggeredEventListener(...). When the method addNetworkTriggeredEventListener(...) is invoked a check that the application session is valid is performed. After this check, the method addNetworkTriggeredEventListenerInternal(...) is invoked. This method implements the Policy Evaluation Point and enables the notification via the NotificationHandler class. This class handles distribution of events between the different instances of the ESPA Service Capability module. The events are distributed when one of the following occurs:

- A new notification was added via the method addNetworkTriggeredEventListener(...).
- An existing notification was disabled via the method removeNetworkTriggeredEventListener(...).
- A callback interface was added to an already existing notification via the method addNetworkTriggeredEventListener(...).
- A callback interface was removed from an existing notification the method removeNetworkTriggeredEventListener(...).
- An object was declared non-functional by the SLEEZombieObjectSupervisor and was therefore removed from the listener.

NotificationHandler holds a hashtable, represented by the variable m_notifications, containing registered notifications and callback interfaces. The NotificationHandler class distributes the details about the notification, such as which application that has requested the notification and stores the information in the database using the NotificationHelper class and distributes the notification information to all other instances of the ESPA service capability module via the SLEE Event Handler. The event is created using generateEvent on the SLEE event channel.

The processEvent(...) method in the class NotificationHandler listens to the events and updates the instance of the ESPA Service Capability module according to the events.

Network protocol plug-in

The SLEE service part of the network protocol plug-in is not involved when registering for network triggered notifications. A listener, MyWPluginNetworkTriggeredEventListenerImpl, is instantiated and registered in the SLEE Common loader when the SLEE service is instantiated. The plug-in also registers in the Plug-in manager when it becomes activated.

The Web Services part of the plug-in registers the implementation of the network triggered interface NetworkInterfaceImpl and MyWPluginNetworkTriggeredEventResultListenerImpl in the SLEE Common Loader, when MyWPluginServlet.init(...) is called at startup of the servlet.

Traffic flow for network triggered requests

The network triggered part of the template is divided into two parts, a registration part where an application registers for network initiated events and the traffic flow part where a request reaches the network plug-in and is propagated through the template traffic path up to an application. The traffic flow is described below and the registration part is described in "Registration flow for network triggered requests" on page 15-9.

Web Services part of the network protocol plug-in

The network node performs the Web Services call myDeliverNetworkTriggeredEventMethod on the Web Service implemented in the Web Services part of the plug-in. The Web Service is bound to the class NetworkTriggeredEventListenerSoapBindingImpl.

The method myDeliverNetworkTriggeredEventMethod(...) is invoked on the class NetworkTriggeredEventListenerSoapBindingImpl. Via the method NetworkTriggeredEventListenerSoapBindingImpl.nameLookUp() the object in the SLEE service part of the plug-in is retrieved from the SLEE Common Loader via the name, OBJ_NETWORK_TRIGGERED_EVENT_LISTENER

The object in the SLEE Service part of the plug-in, implemented in MyWPluginNetworkTriggeredEventListenerImpl was registered in the SLEE Common loader when it was activated.

SLEE Service part of the network protocol plug-in

The Web Services part of the plug-in has performed a call to MyWPluginNetworkTriggeredEventListenerImpl.myDeliverNetworkTriggeredEventMethod(...
).

MyWPluginNetworkTriggeredEventListenerImpl, which implements the listener for network triggered events, was instantiated and registered in the SLEE Common Loader when the SLEE service part of the plug-in was activated, that is when doActivated() is invoked by the SLEE on the class MyPluginSleeService.

The class DeliverNetworkTriggeredEventTask is instantiated and provided as a scheduled task to the SLEE Task Manager via a call to scheduleSLEETask(...).

In DeliverNetworkTriggeredEventTask the method doTask(...) is invoked when the SLEE Task manager has assigned a new task.

In DeliverNetworkTriggeredEventTask.doTask() the class MyPluginNetworkTriggeredEventResultListenerImpl is instantiated. This class is used by the ESPA SC to send back responses to the plug-in describing the outcome of the request.

DeliverNetworkTriggeredEventTask.getListener().myDeliverNetworkTriggeredEventMethod(...) is invoked, where DeliverNetworkTriggeredEventTask.getListener() asks the SC manager for an object in the ESPA SC that listens for network initiated request and returns the listener. This object is instantiated from the class DeliverNetworkTriggeredEventTask in ESPA.

ESPA service capability module

The class MyPluginNetworkTriggeredEventListener_impl implements the listener object the SLEE service part of the plug-in calls when new network initiated events arrives. The class was instantiated and registered in the SC manager when it was activated.

The SLEE service part of the plug-in calls MyPluginNetworkTriggeredEventListener_impl.myDeliverNetworkTriggeredEventMethod(...) where the class DeliverNetworkTriggeredEventTask is instantiated and provided as a scheduled task to the SLEE Task Manager via a call to scheduleSLEETask(...).

In DeliverNetworkTriggeredEventTask the method doTask(...) is invoked when the SLEE Task manager has assigned a new task.

First, DeliverNetworkTriggeredEventTask.doTask() calls internalNotificationArrived(...) on the singleton class NotificationHandler.

The method NotificationHandler.internalNotificationArrived(...) checks if any application has registered for notifications for the events via the method NotificationHandler.findEnabledNotification(...). The method findEnabledNotification(...) returns a list of notifications enabled by the applications.

NotificationHandler.findEnabledNotification(...) checks in the Hashtable m_notifications for notifications registered by the applications. The method returns an object of type MyServiceCapabilityNotification which holds information on which application has registered for notifications and an array of IORs to the SESPA listeners that represents the listeners registered by the applications.

A Policy Evaluation Point is implemented where it is verified if the application is permitted to receive notifications. If the policy rules denies the request, a CDR is generated that the notifications could not be delivered because of a policy violation. If the policy rules permits the request, the method NotificationHandler.executeNotificationArrived(...) is invoked.

In NotificationHandler.executeNotificationArrived(...) the class MyServiceCapabilityNetworkTriggeredEventResultListenerImpl is instantiated. This object is used by the SESPA SC to report successful or failed deliveries of notifications to the ESPA SC. The method MyServiceCapabilityNotification.getCallback() is invoked. This method returns a SESPA object representing a callback that SESPA has registered for an application. MyServiceCapabilityNotification.getCallback() returns the IOR to the SESPA listeners that corresponds to a listener registered by an application.

If no such object is returned, MyServiceCapabilityNetworkTriggeredEventResultListenerImpl.myDeliverNetworkTriggered EventMethodError(...) is called.

If a callback object is found, the method MyServiceCapabilityNetworkTriggeredEventListener.myDeliverNetworkTriggeredEventMeth od(...) is called to the SESPA SC.

SESPA service capability module

The class MyESPAServiceCapabilityNetworkTriggeredEventListener_impl implements the listener object in the SESPA module. The class was instantiated and provided to ESPA when the application enabled the network triggered event listener.

ESPA calls myDeliverNetworkTriggeredEventMethod(...) on the object instantiated from the class MyESPAServiceCapabilityNetworkTriggeredEventListener_impl where the class DeliverNetworkTriggeredEventTask is instantiated and provided as a scheduled task to the SLEE Task Manager via a call to scheduleSLEETask(...).

In DeliverNetworkTriggeredEventTask the method doTask(...) is invoked when the SLEE Task manager has assigned a new task.

First, DeliverNetworkTriggeredEventTask.doTask() calls MyESPAServiceCapabilityNetworkTriggeredEventListener_impl.getNextCallback() which returns an object instantiated from the class MyServiceCapabilityCallbackInfo. The method gets the object from the AbstractList m_callbackList which holds a list of object instantiated from the class MyServiceCapabilityCallbackInfo. The class MyServiceCapabilityCallbackInfo holds information about the notification ticket and the endpoint of the Web Service implemented by the application that enabled network triggered notifications.

Then the object in the WESPA SC that performs the Web Services call to the application is fetched from the SLEE Common loader. The object returned is instantiated from the class MyServiceCapabilityNetworkTriggeredEventListener. The method myDeliverNetworkTriggeredEventMethod(...) is invoked on the class MyServiceCapabilityNetworkTriggeredEventListenerImpl in the WESPA SC.

After the calls has been returned from the WESPA SC, and the application, the method myDeliverNetworkTriggeredEventMethodResult(...) is called on the object provided as result listener. This object is instantiated from the class MyServiceCapabilityNetworkTriggeredEventResultListenerImpl. The method passes the result to the plug-in.

WESPA service capability module

The class MyServiceCapabilityNetworkTriggeredEventListenerImpl implements the method myDeliverNetworkTriggeredEventMethod(...). This method is called from the SESPA SC module.

myDeliverNetworkTriggeredEventMethod(...) calls getMyScNetworkListener(...) which returns an object instantiated from MyServiceCapabilityNetworkTriggeredEventListener.

getMyScNetworkListener uses the Axis generated class MyServiceCapabilityNetworkTriggeredEventListenerServiceLocator to create a proxy object for the application Web Service.

Finally the call to the Web Service implementation in the application is performed using a call to MyServiceCapabilityNetworkTriggeredEventListener.myDeliverNetworkTriggeredEventMeth od(...). When the application has returned the call, the execution proceeds in the ESPA SC module.

Directory structure for the templates

The following directory structure is used for the Extension SDK.

```
dev_tools\

doc\

eclipse\

idl\

        espa_if\
       plugin_if\
lib\
module_templates\
       build\
       client_impl\
       espa_sc_if\
       espa_sc_impl\
       network_if\
       network_simulator_impl\
       plugin_if\
       plugin_impl\
       policy\
              rules\
              sla\
       policy_util_impl\
       sespa_sc_if\
       sespa_sc_impl\
       wespa_sc_if\
       wespa_sc_impl\
       wplugin_if\
       wplugin_impl\
thirdparty\
wsdl\
```

The actual templates for the services are found under `module_templates`.

For each module there is an ant build file in `build.xml` directly under each module.

Source code is found under `src` and IDL interface files are found under `idl`. Any stubs generated from IDL files will be put under a directory `generated`, while compiled class files and packaged jar files will be stored under `lib`.

Below is a description of the individual directories

- `dev_tools`

  Contains the Ant build tool.

- `doc`

  Contains Java doc for all relevant interfaces.

- `eclipse`

  Contains files for setting up an Eclipse development environment for the templates.

- `idl\espa_if`

  Contains IDL-files for the interfaces to the standard ESPA service capability modules.

- `idl\plugin_if`

  Contains IDL-files for the interfaces to the standard plug-ins.

- `module_templates`

  Contains templates for interfaces and implementations of the extension modules. See description below for each sub-directory.

Directly under `module_templates\build` there are two files:

- `build.xml`

  This is a main build file for the entire Extension SDK. This build file prepares the environment by extracting necessary jar files under `lib`, and builds all targets under module_templates. Note that each module may be built by itself under `module_templates\<module_name>\build`, but this file must be built once initially to do the preparations.

- `build.properties`

  Contains configuration settings for Extension SDK.

- `module_templates\client_impl`

  Contains templates for the an application using the interfaces exposed by the Web Services interfaces.

- `module_templates\espa_sc_if`

  Contains templates for the ESPA service capability module interface.

- `module_templates\espa_sc_impl`

Contains templates for the ESPA service capability module implementation.

- `module_templates\network_if`

  Contains example WSDL interface descriptions for the communication between the Web Services part of the plug-in to an example network node (the example network simulator application).

- `module_templates\network_simulator_impl`

  Contains templates for a network simulator.

- `module_templates\plugin`

  Contains templates for the SLEE Service part of the plug-in implementation.

- `module_templates\plugin_if`

  Contains templates for the SLEE Service part of the plug-in interface.

- `module_templates\policy`

  Rules files and SLAs on service provider and application level.

- `module_templates\policy_util_impl`

  Contains templates for a Policy utility class.

- `module_templates\sespa_sc_if`

  Contains templates for the SESPA module interface.

- `module_templates\sespa_sc_impl`

  Contains templates for the SESPA module implementation.

- `module_templates\wespa_sc_if`

  Contains templates for the WESPA module interface.

- `module_templates\wespa_sc_impl`

  Contains templates for the WESPA module interface.

- `module_templates\wespa_sc_impl`

  Contains templates for the WESPA module implementation.

- `module_templates\wplugin_if`

  Contains templates for the interface part of the plug-in that executes in Tomcat (Web Services part).

- `module_templates\wplugin_impl`

  Contains templates for the implementation part of the plug-in that executes in Tomcat (Web Services part).

- `lib`

  Jar files necessary for compiling the templates.

- `thirdparty`

  Necessary third party libraries and binaries.

- `wsdl`

  Interface definition file for the Access Web Service interface.

Introduction to the network plug-in

A network plug-in collaborates with a set of actors:

- SLEE
- ESPA service capability
- Plug-in manager
- SC manager
- Underlying network, or other, node.

The plug-in is built-up by a set of classes and IDL-files.

The SLEE Service part of the plug-in uses a set of utility classes that implements the Basic SLEE services interfaces ServiceDeployable and ServiceAccessible, and extends these classes as described in "Help classes for network plug-ins" on page 2-8.

Files for the SLEE service part of the network plug-in interfaces

my_plugin_if.idl

Location: `plugin_if\idl`

This template file defines the interfaces between the network plug-in and the ESPA service capability module.

The module definitions in this file are used by both the ESPA service capability module implementation and the plug-in implementation. The template holds these interfaces:

`MyPluginListener`, `MyPlugin`, and `MyPluginNetworkListener`

`MyPlugin` is used by the ESPA service capability module to invoke methods on the plug-in. The method definition `myMethodReq` serves as a template for an asynchronous method request. It uses the parameter `assignmentID` to keep track of corresponding requests and responses. The parameter `MyPluginListener` is the callback interface as defined in the interface `MyPluginListener`.

`MyPluginListener` is used by the plug-in to invoke callback methods implemented by the ESPA service capability module. It holds two template methods:

`myMethodRes` and `myMethodErr`.

`myMethodRes` is invoked upon successful completion of a method invocation and `myMethodErr` is invoked upon a failed completion of a method invocation.

It is the responsibility of the plug-in to invoke these methods, while it is the responsibility of the ESPA service capability module to implement them.

`MyPluginNetworkListener` is used by the plug-in to invoke methods implemented by the ESPA service capability module when network triggered events arrives to the plug-in. It holds the method:

`myDeliverNetworkTriggeredEventMethod` which is invoked when the plug-in receives a network initiated event. It is the responsibility of the plug-in to invoke this method, while it is the responsibility of the ESPA service capability module to implement it.

Files for the SLEE service network plug-in implementation

MyPluginOAM.idl

Location: `plugin_impl\idl`

The purpose of this file is to define the interface between the plug-in and the OAM functionality of the SLEE. Methods defined in this interface will be accessible via the Network Gatekeeper OAM interface and the Management Tool.

The module definition in this file reflects the module definition in the plug-in interface.

MyPluginContext.java

Location: `plugin_impl\src\com\acompany\plugin\myplugintype`

Implements the class MyPluginContext. The purpose of this class is to define the service context for the plug-in and to hold data that is global for the plug-in. This class extends the wrapper class PluginContext. The following static variables are defined:

- `PLUGIN_TYPE` defines the plug-in type that the plug-in specifies when registering in the plug-in manager. Service capability implementations will use this type identifier when retrieving plug-ins for handling service requests. The type must match one of the allowed types in the plug-in manager service. Custom types are allowed, but must be registered in the plug-in manager.
- `ADDRESS_PLANS` is an array holding information on the address plans supported by the plug-in. Example:

  ```
  new TrAddressPlan[] {TrAddressPlan.R_ADDRESS_PLAN_IP}
  ```

Possible values are defined in the JavaDoc for the plug-in interfaces.

The package name reflects the module definition in the plug-in interface.

MyPluginSLEEService.java

Location: `plugin_impl\src\com\acompany\plugin\myplugintype`

Implements the class MyPluginSleeService. The purpose of this class is to implement the state control of the plug-in as a SLEE service and to instantiate and provide a reference to the class that implements the traffic interface, the class MyPlugin_impl.

When the service gets activated and started, when the doActivated() and doStarted() methods are invoked, it creates the object that listens for network initiated traffic from the class

MyPluginNetworkTriggeredInterfaceImpl, and it registers it in the SLEE Common Loader so it will be accessible by the Web Services part of the plug-in. See "Plug-ins that executes as a SLEE service and a web application" on page 7-1 for more information about interaction between the web application part of a plug-in and the SLEE service part of a plug-in.

MyPlugin_impl.java

Location: `plugin_impl\src\com\acompany\plugin\myplugintype`

Implements the class MyPlugin_impl. The purpose of this class is to implement the traffic interface of the plug-in as a SLEE service and to implement the plug-in interface used by the ESPA service capability. The template implements the plug-in interface, used by the ESPA service capability module. The actual logic is performed in a separate class, `MyMethodReqTask`. This task class is scheduled as a SLEE task. The SLEE will put it in it's transaction queue and assign a thread to it and let it execute in due time.

```
public void myMethodReq(....) {

...

   MyMethodReqTask task = new MyMethodReqTask(Context.getServiceContext(),
listener, address, data, assignmentID);


Context.getServiceContext().getSLEEContext().getTaskManager().scheduleSLEE
Task(task);

...

}
```

The class MyMethodReqTask performs the actual processing. See MyMethodReqTask.java.

Since the implementation of the interface used by the ESPA service capability module resides in this class, the class must extend the generated POA implementation of the IDL interface:

```
public class MyPlugin_impl extends MyPluginPOA
```

The POA name MyPluginPOA reflects the name of the plug-in interface. In this case `MyPlugin`. See the description of `my_plugin_if.idl`.

The methods defined in the plug-in interface are implemented here. The method `myMethodReq` serves as a template for all methods to be used in the plug-in interface. All methods defined in the plug-in interface must be implemented here. Also the corresponding callback interface must be used. For example, the method `myMethodReq` and the callback interface `myPluginListener` is defined in the plug-in interface, and `myMethodReq` method signature is:

```
myMethodReq(.,.., int assignmentID, MyPluginListener, listener)
```

The `assignmentID` is used to keep corresponding requests and responses together.

MyMethodReqTask.java

Location: `plugin_impl\src\com\acompany\plugin\myplugintype`

Implements the class MyMethodReqTask. This class performs the operations towards the Web Service part of the plug-in. It is implemented in a separate class in order to use the SLEE task processing functions for asynchronous operations. It fetches the object that defines the interface to the Web Services part of the plug-in from the SLEE Common Loader and calls the method `myNetworkMethod` implemented in the Web Services part of the plug-in.

The package name reflects the module definition in the plug-in interface.

DeliverNetworkTriggeredEventTask.java

Location: `plugin_impl\src\com\acompany\plugin\myplugintype`

Implements the class DeliverNetworkTriggeredEventTask. The purpose of this class is to deliver an network triggered event to the ESPA SC via the SC manager.

MyPluginNetworkTriggeredEventResultListenerImpl.java

Location: `plugin_impl\src\com\acompany\plugin\myplugintype`

Implements the class MyPluginNetworkTriggeredEventResultListenerImpl. The purpose of this class is to receive the result from a previously delivered network triggered event from the ESPA SC. The results are processed in the methods myDeliverNetworkTriggeredEventMethodResult(...) and myDeliverNetworkTriggeredEventMethodError(...). Both these methods schedules SLEE Tasks that performs the actual processing, in the classes MyDeliverNetworkTriggeredEventMethodResultTask and MyDeliverNetworkTriggeredEventMethodErrorTask.

MyWPluginNetworkTriggeredEventListenerImpl.java

Location: `plugin_impl\src\com\acompany\plugin\myplugintype`

Implements the class MyWPluginNetworkTriggeredEventListenerImpl. The purpose of this class is to receive notifications on network triggered events from the Web Services part of the plug-in and to schedule a SLEE Task on the class DeliverNetworkTriggeredEventTask when a notification arrives.

MyPluginOAM_impl.java

Location: `plugin_impl\src\com\acompany\plugin\myplugintype`

Implements the class MyPluginOAM_impl. The purpose of this class is to implement the OAM methods that are offered towards the plug-in.

The package name reflects the module definition in the plug-in interface.

Files for the Web Services network plug-in implementation

MyWPluginNetworkTriggeredEventResultListenerImpl.java

Location: `wplugin_impl\src\com\acompany\wplugin\myplugintype`

The purpose of this file is to propagate the result a previously delivered network initiated requests to the network node.

MyWPluginServlet.java

Location: `wplugin_impl\src\com\acompany\wplugin\myplugintype`

The purpose of this file is to add the object implementing the interface the SLEE service part of the plug-in uses to propagate the request to the Web Services part of the plug-in. The object is added to the SLEE Common loader in the `init(...)` method of the servlet.

NetworkInterfaceImpl.java

Location: `wplugin_impl\src\com\acompany\wplugin\myplugintype`

The purpose of this file is implement the interface to the network node.

NetworkTriggeredEventListenerSoapBindingImpl.java

Location: `wplugin_impl\src\com\acompany\wplugin\myplugintype`

This class implements the Web Service called by an underlying network node when network initiated requests arrives. It implements the Web Service method `myDeliverNetworkTriggeredEventMethod(...)` and calls the corresponding method implemented in the SLEE Service part of the plug-in.

Introduction to the ESPA service capability

An ESPA service capability module collaborates with the following set of actors:

- SLEE
- Application
- SESPA module
- Network plug-in

- Policy service

The ESPA service capability module is built-up by a set of classes and IDL-files.

Files for the ESPA service capability interfaces

MyServiceCapability_if.idl

Location: `espa_sc_module_if/idl`

This template file defines the interfaces between the ESPA service capability module and a SESPA module.

The module definitions in this file are used by both the ESPA service capability module implementation and the SESPA module.

In contrary to the plug-in interface, which is asynchronous only, the ESPA interface offers both synchronous and asynchronous request methods to it's clients (SESPA in this case). A client that uses the synchronous request method (see below) does not have to implement or use the callback interface.

The template holds two interfaces for southbound requests:

- `MyServiceCapabilityManager`
- `MyServiceCapabilityListener`

The `MyServiceCapabilityManager` interface is used by a SESPA module to invoke methods on the ESPA service capability module. It contains the methods:

- `myMethod`, a template for an asynchronous method request. It returns an assignmentID to keep track of corresponding requests and responses.
- `addNetworkTriggeredEventListener`, for enabling listeners for network triggered events.
- `removeNetworkTriggeredEventListener,` for removing listeners for network triggered events.

The `MyServiceCapabilityListener` interface is used by the ESPA service capability module to provide the result of an asynchronous method invocation made by a SESPA module. It contains two methods:

`myMethodResult` and `myMethodError`.

`myMethodResult` is invoked upon successful completion of an asynchronous method invocation and `myMethodError` is invoked upon a failed completion of an asynchronous method invocation.

For network initiated traffic three are two interfaces:

- `MyServiceCapabilityNetworkTriggeredEventListener`, for forwarding network initiated requests to the SESPA SC.
- `MyServiceCapabilityNetworkTriggeredEventResultListener`, for delivering the result of previously delivered network triggered notifications.

Files for the ESPA service capability implementation

MyServiceCapabilityOAM.idl

Location: `espa_sc_impl\idl`

The purpose of this file is to define the interface between the ESPA service capability module and the OAM functionality of the SLEE.

ChargingHelper.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

Performs charging, that is generate Charging Data Records (CDRs).

DeliverNetworkTriggeredEventTask.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

Delivers notifications on network triggered events to the SESPA SC.

MyPluginNetworkTriggeredEventListener_impl.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

Listens for network triggered notifications from the plug-in and orders to the SLEE Task Manager to execute doTask in DeliverNetworkTriggeredEventTask.

MyServiceCapabilityNetworkTriggeredEventResultListenerImp l.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

Listens for results of previously delivered network triggered notifications.

MyPluginListener_impl.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

This file holds a class that implements the plug-in callback interface (`MyPluginListener_impl`). It also handles a supervision timer on request invocations towards the plug-in to avoid session hangs if the plug-in fails to deliver a result or error callback.

It delivers the result to the ESPA listener in SESPA. For result deliverance to the ESPA clients, it uses an the inner helper class `MyMethodResultTask` and `MyMethodErrorTask`.

MyServiceCapabilityOAM_impl.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

This class implements the OAM interface of the ESPA service capability. It utilizes the `MyServiceCapabilityPersistentStorage` class to store data of persistent nature in the database.

MyServiceCapabilityPersistentStorage.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

This file holds a helper class used for storing and retrieving configuration data in the database.

MyServiceCapabilityService_impl.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

This file holds the class serving as the service deployable interface. It handles notifications about SLEE service state change and also implements the ESPA Service interface defined in ESPA Access. This interface must be supported by any service that should be accessible via the ESPA Access framework.

NotificationHandler.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

This file contains a class with methods for distributing events to other ESPA SCs. It broadcasts and listens to events originating from an application adding or removing listeners for network triggered events. It uses incoming network triggered notifications from the network plug-in and connects them with listeners that has been registered by the SESPA SC.

MyServiceCapabilityManager_impl.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

This file implements the server part of the `MyServiceCapabilityManager` interface. It implements the Policy Enforcement Point (PEP) and utilizes an inner helper class `MyServiceCapabilityPluginTask` for performing the actual invocations towards the plug-in.

The following line schedules the helper class as a SLEE resource task, which causes the SLEE to obtain a plug-in according to the type defined in `PLUGIN_TYPE` and executes the task from a separate thread:

```
getResourceManager().scheduleResourceTask(new TrProperty[0],
                                           MyServiceCapabilityContext.PLUGIN_TYPE,
                                           TrAddress,
                                           resourceContext,
                                           0, // prio
                          MyServiceCapabilityContext.POLICY_SERVICE_GROUP,
                                           1, // sendlist size
                                           task,
                  MyServiceCapabilityContext.getServiceContext());
```

MyServiceCapabilityContext.java

Location: `espa_sc_impl\src\com\acompany\espa\mysctype`

The purpose of this file is to define the service context for the ESPA service capability module and to hold global data for the ESPA service capability. The following static variables are defined:

- `POLICY_SERVICE_NAME` defines the service name, visible in the Policy service Rule files for this service are loaded with this name.
- `POLICY_SERVICE_GROUP` defines the Policy service group of this service.

- PLUGIN_TYPE defines the plug-in type string that the service uses when obtaining plug-in references from the plug-in manager. This must be the same name as the plug-in uses when it registers itself in the plug-in manager.
- SERVICE_TYPE to define the ESPA service capability type the ESPA service capability module belongs to. Possible values are defined in the JavaDoc for the ESPA service capability and plug-in interfaces.

Other service types than specified in the JavaDoc are also allowed. These are registered through the Management tool at installation.

Policy implementation concept

A Policy Enforcement Point (PEP) forwards data about a certain method invocation to a Policy Decision Point (PDP), where the data is evaluated, and possible manipulated, by a policy rule. The data, which may be modified, is returned to the PEP and the execution continues if the request was allowed according to the PDP. If the request was not allowed, an exception is thrown.

The implementation of a PEP resides in the method definition of the method that will implement a PEP. In the templates, the file `MyServiceCapabilityManager_impl.java` in the directory `espa_sc_impl\src\com\acompany\espa\mysctype` holds an example on how a PEP may be implemented.

Below is a description of how to implement a PEP, based on the example in the Extension SDK.

The example PEP resides in the method `myMethodInternal(...)`. The basic implementation steps of building a PEP is creating an instance of the class `PolicyRequest_impl` and populating it with service context specific data, and then passing it to the policy service for evaluation. Policy evaluation may result in approval or denial of the service request. Denial is communicated to the PEP implementor through exceptions. If a PEP is accepted, the `PolicyRequest` is returned by the policy service, containing the same data as the original `PolicyRequest` except for that some data may have been modified by the policy rule.

The template example code in `myMethodInternal()` shows how the `PolicyRequest` is instantiated and populated with mandatory parameters. Any properties specific to the particular context where the PEP is enforced may be passed to policy rules as optional additional parameters in the `PolicyRequest`. In the example, the data and address parameters are passed as optional.

Files for the Policy utility

MyPolicyUtility.java

Location: `policy_util_impl\src\com\acompany\policy\util`

The purpose of this class is to expose a public method, `subscriberExists`, that checks in a database if a user exists. The class is a singleton class in order to be called from a Policy Rule. It uses the Subscriber profile plug-in interface to fetch subscriber data. The plug-in is found in `lib\b_db_sp_resource.jar`. See "Creating an example Policy Utility" on page 18-1 for information on how to deploy the plug-in and to provision users in the database.

The Policy Utility called from the example rule `DenySubscriberNotExists,` found in `ESPA_myservicecapability.ilr` in `policy\rules\sp\.` The rule is triggered from PEP in the template ESPA Service Capability, see "Policy implementation concept" on page 15-30.

MyPolicyUtilityOAM_impl.java

Location: `policy_util_impl\src\com\acompany\policy\util`

Implements the OAM interface for the Policy Utility. In the template, there are no OAM methods.

MyPolicyUtilityException.java

Location: `policy_util_impl\src\com\acompany\policy\util`

Defines an exception for the Policy Utility class.

Introduction to the SESPA module

A SESPA module collaborates with the following set of actors:

- SLEE
- WESPA module
- ESPA service capability

The SESPA module is built-up by a set of classes and IDL-files.

Files for the SESPA module interface

MyServiceCapability.java and MyESPAServiceCapabilityListener.java

Location: `sespa_sc_if\src\com\acompany\sespa\mysctype`

These template files define the interfaces between the SESPA module and a WESPA module.

The name of these files reflect the name of the SESPA module.

The templates define two interfaces:

- `MyServiceCapability`
- `MyServiceCapabilityListener`

The `MySESPAServiceCapability` interface is used by a WESPA module to invoke methods on the SESPA module. It contains these methods:

- `myMethod`, a template for an asynchronous method request. It returns an assignmentID to keep track of corresponding requests and responses.
- `myMethodWait`, a template for a synchronous method request. It returns the result of the request.
- `enableNetworkTriggeredEvents`, a template for enabling listeners for network triggered requests.
- `disableNetworkTriggeredEvents`, a template for disabling of previously registered listeners for network triggered requests.

The `MyServiceCapabilityListener` interface is a callback interface used by the SESPA module to provide the result of an asynchronous method invocation made by a WESPA module. It contains two methods, `myMethodResult` and `myMethodError`.

`myMethodResult` is invoked upon successful completion of an asynchronous method invocation and `myMethodError` is invoked upon a failed completion of an asynchronous method invocation.

MyServiceCapabilityNetworkTriggeredEventListener.java

Location: `sespa_sc_if\src\com\acompany\sespa\mysctype`

This template file define the interface between the SESPA module and a WESPA module for network triggered notifications.

The `MyServiceCapabilityNetworkTriggeredEventListener` interface is used by a SESPA module to invoke methods on the WESPA module. It contains the method:

- `myDeliverNetworkTriggeredEventMethod`, for delivering network initiated requests to the WESPA SC.

Files for the SESPA module implementation

MyServiceCapabilityOAM.idl

Location: `sespa_sc_impl\idl`

The purpose of this file is to define the interface between the SESPA service capability module and the OAM functionality of the SLEE.

MyServiceCapabilityOAM.idl

Location: `sespa_sc_impl\idl`

The purpose of this file is to deliver notifications on network triggered events from the SESPA SC module to the WESPA SC module.

The purpose of this file is to define the interface between the SESPA service capability module and the OAM functionality of the SLEE.

MyESPAServiceCapabilityListener_impl.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

This is the asynchronous ESPA callback interface implementation. It forwards results received from ESPA to the WESPA module.

MyESPAServiceCapabilityNetworkTriggeredEventListener_impl.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

The purpose of this class is to implement the listener the ESPA SC module calls when the ESPA SC receives notifications on network triggered notifications. It also maps ESPA login information and notification IDs to login tickets and notification IDs used in the WESPA SC.

MyServiceCapabilityEventHelper.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

The purpose of this class is to keep the different instances of the SESPA SC in sync. It generates events when an application has registered for notifications on network triggered events and it also handles reception of these events. It is also responsible for acting on events related to the login ticket expirations and refresh.

MyServiceCapabilityService.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

The purpose of this class is to implement the service deployable interface. It instantiates help classes when the SESPA module's state is changed from `installed` to `started` and performs class loader registration at activation. At deactivation the classes are unregistered from the class loader. Clean up procedures are performed both when the module is deactivated and stopped.

The reason for the class loader registrations is that the interface between SESPA and WESPA is a normal Java interface. Class definitions and objects must therefore be loaded into a class loader that is common for both SESPA and WESPA. The interface class files, `MySESPAServiceCapability.class` and `MySESPAServiceCapabilityListener.class`, must only be included in the SESPA jar file and not in the WESPA war file since the services must share a common definition. The `SLEECommonLoader` class loader is used by both services for registering and retrieving classes and objects. This is a parent class loader of both services, the WESPA service executes in Tomcat, and Tomcat has SLEECommonLoader as parent class loader.

MyServiceCapabilityOAM_impl.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

The purpose of this class is to implement the service manageable object and the OAM methods defined in the `MyServiceCapabilityOAM.idl` file. In addition, it sets the service context and creates a child POA for the ESPA service capability listeners.

MyServiceCapabilityPersistentStorage.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

The purpose of this class is to handle operations towards the database.

MyServiceCapabilityContext.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

This file is a helper class that keeps track of the ESPA service capability listeners and other service context related data.

MyServiceCapabilityImpl.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

This is the SESPA interface implementation. It performs the mapping between the SESPA and ESPA interfaces. In addition, it validates the login ticket received from WESPA.

NotificationInfo.java and NotificationInfoId.java

Location: `sespa_sc_impl\src\com\acompany\sespa\mysctype`

Holds data about notifications for network triggered events.

Introduction to the WESPA module

An WESPA module collaborates with the following set of actors:

- client application
- SESPA module

The WESPA module is built-up by a set of classes.

Files for the WESPA module interface

MyServiceCapability.java and MyServiceCapabilityListener.java

Location: `wespa_sc_if\src\com\acompany\wespa\mysctype`

These template files define the interfaces between the WESPA module and an application.

The name of these files reflect the name of the WESPA module.

The templates hold these interfaces for application initiated requests:

- `MyServiceCapability`

- `MyServiceCapabilityListener`

The `MyServiceCapability` interface is used by a client to invoke methods on the SESPA module. It contains these methods:

- `myMethod`, a template for an asynchronous method request. It returns an assignmentID to keep track of corresponding requests and responses.
- `myMethodWait`, a template for a synchronous method request. It returns the result of the request.
- `enableNetworkTriggeredEvents`, for enabling listeners for network triggered events.
- `disableNetworkTriggeredEvents`, for disabling previously registered listeners for network triggered events.

The `MyServiceCapabilityListener` interface is a callback interface used by the WESPA module to provide the result of an asynchronous method invocation made by a client. It contains two methods, `myMethodResult` and `myMethodError`.

`myMethodResult` is invoked upon successful completion of an asynchronous method invocation and `myMethodError` is invoked upon a failed completion of an asynchronous method invocation.

MyServiceCapabilityException.java

Location: `wespa_sc_if\src\com\acompany\wespa\mysctype`

The file is a class that implements exception handling.

MyServiceCapabilityNetworkTriggeredEventListener.java

Location: `wespa_sc_if\src\com\acompany\wespa\mysctype`

This file defines the interface for delivering notifications on network triggered events.

Files for the WESPA module implementation

MyServiceCapabilityListenerImpl.java

Location: `wespa_sc_impl\src\com\acompany\wespa\mysctype`

This is the asynchronous SESPA callback interface implementation. It forwards results received from SESPA to the application.

MyServiceCapabilityNetworkTriggeredEventListenerImpl.java

Location: `wespa_sc_impl\src\com\acompany\wespa\mysctype`

This is the implementation of the interface used for delivering network triggered notifications to an application.

MyServiceCapabilitySoapBindingImpl.java

Location: `wespa_sc_impl\src\com\acompany\wespa\mysctype`

This class implements the server part of the WESPA `MyServiceCapability` interface. It is deployed in the Axis web services engine.

Introduction to the test application

The test application uses the Web Service implemented by the WESPA Service capability module and implements the Web Service called by the WESPA Service Capability module.

Files for the test application

LoginHelper.java and LoginInfo.java

Location: `client_impl\src\com\acompany\test`

Handles login towards WESPA Access and holds the login information.

MyScHelper.java

Location: `client_impl\src\com\acompany\test`

Performs the calls towards the Web Service exposed by the WESPA Service Capability.

MyServiceCapabilityListenerImpl.java

Location: `client_impl\src\com\acompany\test`

Implements the Web Service the WESPA Service Capability module uses to reports results of asynchronous application triggered requests.

TestClient.java

Location: `client_impl\src\com\acompany\test`

Main program for the test client that implements a menu and call the different parts of the test client according to the menu choices and information given.

TraceLogService_impl.java and TraceLogService.java

Location: `client_impl\src\com\acompany\test`

Helper class and interface to perform trace.

WespaHelper.java

Location: `client_impl\src\com\acompany\test`

Helper class that creates Web Services calls and creates security headers.

Introduction to the network simulator

The network simulator application uses a Web Service implemented by the Web Services part of the network plug-in and implements the Web Service called by the Web Services part of the network plug-in. For application initiated requests it receives the request and prints the request data to System.out. For network initiated requests it performs a Web Services call to the Web Services part of the network plug-in.

Files for the network simulator application

NetworkTriggeredInterfaceHelper.java

Location: `network_simulator_impl\src\com\acompany\test`

Performs network initiated requests to the Web Services part of the network plug-in. Uses a Web Service.

NetworkInterfaceImpl.java

Location: `network_simulator_impl\src\com\acompany\test`

Receives application triggered requests from the plug-in. Implements a Web Service.

TestClient.java

Location: `network_simulator_impl\src\com\acompany\test`

Main program for the test client that implements a menu and call the different parts of the test client according to the menu choices and information given.

TraceLogService_impl.java and TraceLogService.java

Location: `network_simulator_impl\src\com\acompany\test`

Helper class and interface to perform trace.

Preparing the development environment

Copy templates

In order to create a new set of modules, copy the relevant modules from `bea\wlng21\esdk\module_templates` to a new directory where the development will take place. This directory will be referred to as `exampleproj`.

Preparing the build environment

In the file `exampleproj\build.properties`, change the properties described below.

Note: All slashes that separates directories, should be regular slashes, not back slashes (\) on Windows systems also.

Define the property:

- wlng.root.dir to the directory where the test environments WebLogic Network Gatekeeper is installed, for example `C:/wlng21`
- wlng.sleeManager.username to the username for the user that manages the Network Gatekeeper, by default this is `install`.
- wlng.sleeManager.password to the password for the user that manages the Network Gatekeeper, by default this is `install`.
- esdk.root.dir to the directory where the Extension SDK is installed, for example `C:/bea/wlng21/esdk`
- thirdparty.orbacus.dir to the directory where Orbacus is installed.

- pipe.root.dir to the directory where the new modules will be developed. In the example this directory is `exampleproj`.

Adapting the build files for the modules

Depending on which modules to be developed, the build files for these module needs to be adapted to reflect the new modules. All module-specific build files are named build.xml, and are located directly under the directory for the module.

For a completely new traffic path, there is a main build file, `exampleproj\build\build.xml`, that calls a a set of build files specific for the individual modules. If the new project does not include all modules this build file should be adapted to include only the relevant modules.

Properties common for all modules

The following properties can be changed in the module-specific build files to reflect any changes in interface names, module name etc:

- local.majorVersion, main version information for the module.
- local.minorVersion, sub-version of the module.

The version information will be added to the deployment descriptor for the module.

- local.prodName, the name of the module, used in JavaDoc generation.
- local.jarName, the name of the generated jar file.
- local.deployName, the name of module as seen in the Management tool. This name will be added to the deployment descriptor.
- local.deployMaxAlarms, the maximum number of the severe alarms the software module is permitted to emit before it is put into state deactivated.
- local.deploySvcDeployable, the class that implements the Service Deployable interface. Full package name must be given.
- local.deploySvcAccessible, the class that implements the Service Accessible interface. Full package name must be given.
- local.deploySvcManageable, the class that implements the Service Manageable interface. Full package name must be given.
- local.deploySvcManageableIdl, the name of the IDL file describing the OAM interface for the module.

- local.javadocFilename, the file name of the zipped JavaDoc.
- local.javadocTitle, the title of the JavaDoc.
- local.javadocIncludedPaths, the paths to which JavaDoc to be built.
- local.javadocIncludedPackages, the packages to include in the JavaDoc.
- local.javadocExcludedPackages, the packages to exclude from the JavaDoc.

Deployment descriptor

All software modules executing as SLEE services are packed in a jar file. This is automatically performed by the build files.

In the root of the jar file there is a deployment descriptor. This is an XML file describing how the software module is deployed. The name of the XML file is always srv_depl.xml, and the build files automatically creates this file in the ant target "jar". Below is a description of the tags in the XML file.

`<SLEE_SERVICE>` Defines the service deployment descriptor. It has the following attributes:

- `name`: the name of the SLEE Service, fetched from the property local.deployName.
- `version`: the version of the service, fetched from the property local.fullVersion.
- `max_alarms`:-maximum number of CRITICAL alarms the service is allowed to generate before the service is set is state ERROR by the SLEE.
- `company`: unused.
- `trace`: ON or OFF. Defines if trace shall be activated for the service when it is started for the first time.
- `db_share`: unused.
- parent_class_loader_service: Which service to be used as parent class loader. If the service shall interact with a servlet or a Web Service executing in Embedded Tomcat, this attribute shall be set to `"Slee_common_loader"`. If it is a SESPA service, set this attribute to `"SESPA_access"`. SESPA_Access has the `Slee_common_loader` as a parent class loader, so the SESPA services have access to the classes in the SLEE Common loader. See "Plug-ins that executes as a SLEE service and a web application" on page 7-1 and "Stateless adapter framework" on page 5-4 for more information on SLEE Common Loader.

`<SERVICE_DEPLOYABLE>` Defines the class that implements the ServiceDeployable interface.

`<SERVICE_ACCESSIBLE>` Defines the class that implements the ServiceAccessible interface. Contains the following tag:

- `<NAMESERVICE_VISIBILITY>` ON or OFF. Defines if the CORBA objects that implements the ServiceAccessible interface shall be registered in the CORBA nameservice, and be reachable from outside the SLEE.

`<SERVICE_MANAGEABLE>` Defines the class that implements the ServiceManageable interface. Contains the following tag:

- `<SERVICE_MANAGEABLE_IDL>` Defines the IDL file that describes the OAM interface.

Properties for the plug-in implementation

- local.pluginIfIdl.dir, the path to the plug-in interface IDL definition.
- local.pluginIfLib.dir, the path to the plug-in interface class files.
- local.pluginIfLib.packages, the package name of the plug-in interfaces.
- local.pluginIf.jar, the name of the jar-file for the plug-in interfaces.

Properties for the ESPA implementation

- local.espaIfLib.dir, the path to the ESPA interface class files.
- local.espaIf.jar, name of the jar-file for the ESPA interface class files.
- local.pluginIfLib.dir, the path to the plug-in interface class files.
- local.pluginIf.jar, the name of the jar-file for the plug-in interfaces.
- oamFile, the name of the IDL file defining the OAM interface.

Properties for the SESPA implementation

- local.sespaIfLib.dir, the path to the SESPA interface class files.
- local.sespaIf.jar, name of the jar-file for the SESPA interface class files.
- local.espaIfLib.dir, the path to the ESPA interface class files.
- local.espaIf.jar, name of the jar-file for the ESPA interface class files.
- local.classForRmi, name of the class implementing the RMI interface exposed by SESPA.

- oamFile, the name of the IDL file defining the OAM interface.

Properties for the WESPA implementation

- local.sespaIfLib.dir, the path to the SESPA interface class files.
- local.sespaIf.jar, name of the jar-file for the SESPA interface class files.
- local.wespaIfLib.dir, the path to the WESPA interface class files.
- local.wespaIf.jar, name of the jar-file for the WESPA interface class files.

Properties for the WESPA client

- myScWsdl, the path to the WSDL file describing the WESPA Web Service.
- myScListenerWsdl, the path to the WSDL file describing the WESPA listener.
- wespaAccessWsdl, the path to the WSDL file describing the WESPA Access Web Service.

Using the templates from Eclipse

The templates and the build environment supports Eclipse 3.1.1.

Below is a suggestion on how to setup an Eclipse workspace for the templates.

1. Extract the file `bea\wlng21\esdk\eclipse\lib_eclipse.zip` to `bea\wlng21\esdk`.
2. Extract the file `bea\wlng21\esdk\eclipse\thirdparty_eclipse.zip` to `bea\wlng21\esdk`.
3. Extract the file `bea\wlng21\esdk\eclipse\module_templates_eclipse.zip` to `bea\wlng21\esdk\module_templates`.
4. Copy the directory `bea\wlng21\esdk\module_templates` to the working directory for your project. This directory will be referred to as `exampleproj`.
5. Start Eclipse.
6. Select the workspace to be `bea\wlng21\esdk\example_proj`

   The following projects should be defined:

   – build

   – client_impl

- espa_sc_if
- espa_sc_impl
- lib
- network_if
- network_simulator_impl
- plugin_if
- plugin_impl
- policy_util_impl
- sespa_sc_if
- sespa_sc_impl
- thirdparty
- wespa_sc_if
- wespa_sc_impl
- wplugin_if
- wplugin_impl

7. Delete the project lib. Do not delete the contents.
8. Create a new Java Project referring to `bea\wlng21\esdk\lib` by:
   a. Selecting File, New Project...
   b. Select Java Project and click Next.
   c. Name the project lib, and choose Create a project from existing source and choose the directory `bea\wlng21\esdk\lib`.
   d. Click Finish.

   Now a project named lib is created.
9. Delete the project thirdparty. Do not delete the contents.
10. Create a new Java Project referring to `bea\wlng21\esdk\thirdparty` by:
    a. Selecting File, New Project...
    b. Select Java Project and click Next.

   c. Name the project thirdparty, and choose Create a project from existing source and choose the directory `bea\wlng21\esdk\thirdparty`.

   d. Click Finish.

   Now a project named thirdparty is created.

11. Define the Ant Home property by Choosing Window, Preferences... and then select Ant Runtime. Click the Ant Home... Button and select the directory `bea\wlng21\esdk\dev_tools\ant`.

12. Make sure that the additional Ant targets are known in the workspace.

   a. Select Preferences... in the Windows menu item.

   b. Under Ant, Runtime click on Add External Jars... and select the files `ant-contrib.jar` and `idldepend-0-8-1.jar` in the directory `bea\wlng21\esdk\dev_tools\ant\lib`.

13. Edit build.properties in the build project to reflect your installation. See "Preparing the development environment" on page 15-39 for information on which properties to edit.

14. Run the ant task named dist on the file build.xml in the build project. Right-click on the file and select Run As... and select Ant Build...

   Now the project starts to build.

15. For each project, select the project and do a refresh by Choosing File, Refresh.

   The indicators on the project symbol, white crosses on a red background, should disappear when refreshing the project.

16. The local build files, located directly under the sub directory for each module needs to be adapted to the desired structure, see "Adapting the build files for the modules" on page 15-40.

Creating an example network plug-in

The following section provides a description on how to create an example network protocol plug-in:

In this section, the example network plug-in is described.

The plug-in consists of two parts:

- a SLEE service part that executes as a SLEE service
- A Web Services part that executes as a Web Service

The SLEE service part of the example plug-in has the following properties:

- Package: `com.acompany.plugin.myplugintype`
- Interface used by ESPA service capability: `MyPlugin`
- Interface used by the network plug-in acting on the ESPA service capability: `MyPluginListener`
- Method implemented by the plug-in, defined in the plug-in interface:
  - `myMethodReq`
  - `myDeliverNetworkTriggeredEventMethodResult`
  - `myDeliverNetworkTriggeredEventMethodError`
- Methods invoked from the plug-in:
  - `MyMethodRes`
  - `MyMethodErr`
  - `myDeliverNetworkTriggeredEventMethod`
- Type for the plug-in: `MYPLUGIN_TYPE`
- Address format supported: E164
- Project names: `plugin_if` and `plugin_impl`
- Product names: `My_Plugin_IF` and `My_Plugin_Impl`
- Jar file names: `plugin_myplugin_if.jar` and `b_plugin_myplugin.jar`

The Web Services part of the example plug-in has the following properties:

- Package: com.acompany.wplugin.myplugintype
- Interface used by the SLEE Service part of the plug-in service capability: `MyPluginNetworkInterface`
- Interface used by the Web Service network plug-in acting on the SLEE part of the plug-in: MyWPluginNetworkTriggeredEventResultListener
- Method implemented by the Web Services network plug-in, defined in the plug-in interface:
  - my`MethodReq`

  - `myDeliverNetworkTriggeredEventMethodResult`
  - `myDeliverNetworkTriggeredEventMethodError`
- Methods used by the plug-in, invoked from the plug-in:
  - `MyMethodRes`
  - `MyMethodErr`
  - `myDeliverNetworkTriggeredEventMethod`
- Project names: `wplugin_if` and `wplugin_impl`
- Product names: `wplugin_if` and `wplugin_impl`
- Jar file names: `wplugin_if.jar` and `wplugin.war`

General preparations for the SLEE part of the plug-in

1. Make sure the files for the SLEE plug-in and SLEE plug-in interfaces are copied to the directory `exampleproj`. That is, all files and directories in `module_templates\plugin_impl and module_templates\plugin_if`.
2. Make sure the files for the Web Services part of the plug-in interfaces are copied to the directory `exampleproj`. That is, all files and directories in `module_templates\wplugin_if` and `module_templates\wplugin_impl`
3. Change directory to `bea\wlng21\esdk\exampleproj\build` and issue the command `ant`

Preparing the SLEE plug-in interface

Set up the build environment

4. Edit the file `exampleproj\plugin_if\build.xml`.
5. Edit the properties described in "Adapting the build files for the modules" on page 15-40 to reflect the desired names.

Define the plug-in interface structure

6. In the file `exampleproj\plugin_if\idl\my_plugin_if.idl`, change the plug-in interface structure to reflect the desired package structure. For example, the structure:

```
module com {
```

```
            module acompany{
                    module myplugintype{
```

Will create the plug-in structure `com.acompany.myplugintype`

Interfaces in the plug-in

These interfaces are defined:

- `MyPluginListener`, which will be implemented by the ESPA service capability implementation.
- `MyPlugin`, which will be implemented by the plug-in implementation.
- `MyPluginNetworkTriggeredEventListener`, which will be implemented by the ESPA service capability implementation.
- `MyPluginNetworkTriggeredEventResultListener`, which will be implemented by the plug-in implementation.

In the example two methods related to application-initiated request will be called from by the plug-in to the ESPA service capability, `myMethodRes` and `myMethodErr`. Both methods are responses to the operation `exampleMethodReq`, invoked by the ESPA service capability and implemented by the plug-in.

- `myMethodRes` will be invoked if the operation `myMethodReq` was successful
- `myMethodErr` will be invoked if the operation `myMethodReq` failed.

`assignmentID` is the ID that connects an invocation to `myMethodReq` and the corresponding invocation of `myMethodRes` or `myMethodErr`.

In the example one method related to network initiated request will be called from by the plug-in to the ESPA service capability, `myDeliverNetworkTriggeredEventMethod`. This method is used to deliver network triggered events to the ESPA service capability. As a response to a network triggered event, the following methods will be invoked from the ESPA service capability:

- `myDeliverNetworkTriggeredEventMethodResult` will be invoked if the operation `myDeliverNetworkTriggeredEventMethod` was successful
- `myDeliverNetworkTriggeredEventMethodError` will be invoked if the operation `myDeliverNetworkTriggeredEventMethod` failed.

networkTransactionId is the ID that connects an invocation to `myDeliverNetworkTriggeredEventMethod` and the corresponding invocation of `myDeliverNetworkTriggeredEventMethodResult` or `myDeliverNetworkTriggeredEventMethodError`.

7. Add or change additional methods to be used in the interface between the ESPA service capability and the plug-in. Use the definitions in `exampleproj\plugin_if\idl\my_plugin_if.idl` as templates.

Compilation of the SLEE plug-in interface

8. Compile the Web Services plug-in interface by changing directory to `exampleproj\wplugin_if` and execute the command `ant`

   This compiles the interface between the SLEE plug-in and the Web services part of the plug-in to be used by the SLEE plug-in implementation.

9. Compile the SLEE plug-in interface by changing directory to `exampleproj\plugin_if` and execute the command `ant`

   This compiles the SLEE plug-in interface, and generates Java and CORBA files to be used by the plug-in implementation.

Preparing the SLEE plug-in implementation

Set up the build environment

10. Edit the file `exampleproj\plugin_impl\build.xml`.

11. Edit the properties described in "Adapting the build files for the modules" on page 15-40 to reflect the desired names.

    Note: Always use a prefix in the jar name. The autostarted SLEE services are started in alphabetic order based on the jar name. All network plug-ins should have a prefix so they are started prior to the ESPA service capability modules.

Defining the plug-in OAM methods

12. Edit the files `exampleproj\plugin_impl\idl\MyPluginOAM.idl`. Define any additional OAM methods.

Adapting the plug-in interface implementation

13. Adapt the plug-in interface so it implements the plug-in interface as defined in Interfaces in the plug-in.
14. Edit the file `MyPlugin_impl.java` in `exampleproj\plugin_impl\src\com\acompany\plugin\myplugintype.`
15. If the name of the interface in the plug-in has changed, change the name of the POA the class extends to reflect the new name.
16. If the methods in the plug-in interface has changed, also change the names of the implementing methods in this class.
17. Edit the file `MyMethodReqTask.java` in `exampleproj\plugin_impl\src\com\acompany\plugin\myplugintype`

    Adapt the methods so they use call-back methods defined in the plug-in interface as defined in Interfaces in the plug-in.

Plug-in type definition

Each plug-in has a type definition and information on supported address plans.

18. Edit the file `MyPluginContext.java in exampleproj\plugin_impl\src\com\acompany\plugin\` and change the `PLUGIN_TYPE` definition to the desired type.

    Define which address plan(s) that is (are) supported in the `TrAddressPlan[]` structure.

    As default, the address plan is `TrAddressPlan.R_ADDRESS_PLAN_E164`, see the JavaDoc for the ESPA service capability and plug-in interfaces for a list of address plans.

Compilation of the SLEE plug-in implementation

Compile the plug-in implementation by changing directory to `exampleproj\plugin_impl` and execute the command `ant`

Installing the SLEE plug-in

This section describes how to install and deploy the plug-in that was created using the instructions in this section. For instructions on how to use the Management Tool and how to register a plug-in in more detail, see WebLogic Network Gatekeeper User's Guide.

1. Make sure you have access (by ftp directly to the file system) to the SLEE the plug-in shall be installed in.
2. Open the Management Tool and select the SLEE where to install the plug-in.
3. In the management tool, select the **Plugin_Manager** service, and invoke the method `addType`. Use the type defined for the plug-in.

   In the example, the plug-in type is `MYPLUGIN_TYPE,` as defined in step 18. in "Plug-in type definition".
4. In the **SLEE_deployment** service, select **install**. Enter the URL to the jar-file in the field **ServiceJarURL** and click Invoke.

   In the example, the path is:

   `file:///<drive>/exampleproj/plugin_impl/lib` and the name was defined in the build file property local.jarName.

   Note: Use three (3) slashes before the drive name on Windows systems.
5. In the **SLEE_deployment** service, select **start**. Enter the SLEE name for the plug-in. Click **Invoke**.

   The name was defined in the build file property local.deployName.
6. In the **SLEE_deployment** service, select **activate**. Enter the SLEE name for the plug-in. Click **Invoke**.

Setting up a plug-in route

It is necessary to define a route, so the plug-in can be selected by the plug-in manager.

7. In the management tool, select the **`Plugin_Manager`** service, and invoke the method **`getIdList`**. Do not define any parameters.

   An ID for the plug-in is returned.
8. In the management tool, choose the **Plugin_Manager** service, and invoke the method **addRoute**. In the **Id** field, enter the ID returned from the previous operation.

   In the **addressExpression** field, enter `^.*` in order make the plug-in accept all destination address numbers, or other routing criteria that is desired.

General preparations for the Web Services part of the plug-in

1. Make sure the files for the Web Service plug-in and plug-in interfaces are copied to the directory `exampleproj`. That is, all files and directories in `module_templates\wplugin_impl and module_templates\wplugin_if`.
2. Change directory to `bea\wlng21\esdk\build` and issue the command `ant`

Preparing the Web Services plug-in interface

Set up the build environment

3. Edit the file `exampleproj\wplugin_if\build.xml`.
4. Edit the properties described in "Adapting the build files for the modules" on page 15-40 to reflect the desired names.

Interfaces in the plug-in

There are three interfaces defined:

- `MyWPluginNetworkInterface`, which will be implemented by the Web Services part of the plug-in and the request is passed on to the network.
- `MyWPluginNetworkTriggeredEventListener`, which will be implemented by the SLEE Service part of the plug-in implementation.
- `MyWPluginNetworkTriggeredEventResultListener`, which is implemented Web Services part of the plug-in.

In the example these methods will be called from the SLEE service part of the plug-in to Web Services part of the plug-in.

- `myNetworkMethod` will be invoked as a result of an application triggered request.
- `myDeliverNetworkTriggeredEventMethod` will be invoked on the SLEE part of the network plug-in as a result of a network triggered event.
- `myDeliverNetworkTriggeredEventMethodResult` or `myDeliverNetworkTriggeredEventMethodError` will be invoked on the Web Services part of the plug-in by the SLEE part of the network plug-in as a result of a network triggered event.

5. Add or change additional methods to be used in the interface between the SLEE service part of the network plug-in and the Web Services part of the plug-in. Use the definitions in `exampleproj\wplugin\myplugintype` as templates.

Compilation of the Web Services part of the plug-in interface

6. Compile the plug-in interface by changing directory to `exampleproj\wplugin_if` and execute the command `ant`

   This compiles the plug-in interface.

Preparing the Web Services part of the plug-in implementation

Set up the build environment

7. Edit the file `exampleproj\plugin_impl\build.xml`.

8. Edit the properties described in "Adapting the build files for the modules" on page 15-40 to reflect the desired names.

   Note: Always use a prefix in the jar name. The autostarted SLEE services are started in alphabetic order based on the jar name. All network plug-ins should have a prefix so they are started prior to the ESPA service capability modules.

Adapting the Web Services plug-in interface implementation

9. Adapt the plug-in interface so it implements the plug-in interface as defined in "Interfaces in the plug-in" on page 16-8.

10. Edit the file `NetworkInterfaceImpl.java`, `MyWPluginNetworkTriggeredEventResultListenerImpl.java` and `MyWPluginServlet.java` in `exampleproj\wplugin_impl\src\com\acompany\wplugin\myplugintype`.

11. If the names of the interfaces has changed, change the name of the names the classes implements to reflect the new name.

12. If the methods in the interfaces has changed, also change the names of the implementing methods in this class.

Compilation of the Web Services plug-in implementation

Generate Java stubs from the WSDL file that describes the interface between the Web Services part of the plug-in and the network node, by changing directory to `exampleproj\network_if` and execute the command `ant`

Compile the Web Services plug-in implementation by changing directory to `exampleproj\wplugin_impl` and execute the command `ant`

Installing the Web Service plug-in

This section describes how to install and deploy the Web Services plug-in that was created using the instructions in this section. For instructions on how to use the Management Tool, see WebLogic Network Gatekeeper User's Guide.

1. Make sure you have access (by ftp directly to the file system) to the SLEE the plug-in shall be installed in.

1. Copy the generated war file to the `/<install dir>/slee/bin/autowar` directory in the Network Gatekeeper.

2. Open the Management Tool and select the SLEE running on the server the file was copied to.

3. In the **Embedded_tomcat** service, choose **addContext**. Enter the following parameter data:

Parameter	**Description**
contextPath	The context path to be used. For example: `/exampleContext` The Web Service will be reached in the following URL: http://<IP-address>:<port>/<contextPath>/services/<method>

docBase	Document root. Can be a war-file or a directory. Can be specified with an absolute or a relative path name or an URL. In the example the files are stored in:
	`/usr/local/slee/bin/autowar/`
	`<name of war file>.war`

useCookies	Enable cookies (TRUE/FALSE) for session handling. Use FALSE.
autostart	Start this context during next service restart (TRUE/FALSE). Use TRUE.

4. Click **Invoke**.

   The Web Service part of the plug-in is started.

Creating an example ESPA Service Capability module

The following section provides a description on how to create an example ESPA service capability module:

- General preparations
- Preparing the ESPA service capability interface
- Compilation of the ESPA service capability module interface
- Preparing the ESPA service capability module implementation
- Implement the Policy Enforcement Point
- Compilation of the ESPA service capability module implementation
- Adapting the policy rules
- Installing the ESPA service capability module
- Update Service Level Agreements (SLAs)
- Install policy rules

The example ESPA service capability will use the interfaces defined for the example network plug-in prepared in "Creating an example network plug-in" on page 16-1. The ESPA service capability has the following properties:

- Package: `com.acompany.espa.mysctype`

- Interface used by the SESPA module, acting on the ESPA service capability for application-initiated requests:
  `MyServiceCapabilityManager`
- Interface used by the ESPA service capability for application-initiated requests, acting on the SESPA module:
  `MyServiceCapabilityListener`
- Methods implemented by the ESPA service capability for application-initiated requests, defined in the `MyServiceCapabilityManager` interface:
  - `myMethod`
  - `addNetworkTriggeredEventListener`
  - `removeNetworkTriggeredEventListener`
- Methods invoked by the ESPA service capability, implemented the SESPA module:
  - `myMethodRes`
  - `myMethodErr`
- Interface used by the SESPA module, acting on the ESPA service capability for network-triggered requests:
  `MyServiceCapabilityNetworkTriggeredEventListener`
- Interface used by the ESPA service capability for application-triggered requests, acting on the SESPA module:
  `MyServiceCapabilityNetworkTriggeredEventResultListener`
- Methods implemented by the SESPA service capability for network-triggered requests, defined in the `MyServiceCapabilityNetworkTriggeredEventListener` interface:
  - `myDeliverNetworkTriggeredEventMethod`
- Methods invoke by the SESPA service capability, implemented by the ESPA module:
  - `myDeliverNetworkTriggeredEventMethodResult`
  - `myDeliverNetworkTriggeredEventMethodError`
- ESPA service capability type: MY_SCS_TYPE

General preparations

This is general preparations.

1. Make sure the files for the service capability and the service capability interfaces are copied to the directory `exampleproj`. That is, all files and directories in `module_templates\espa_sc_impl and module_templates\`espa_sc_if.

2. Change directory to `bea\wlng21\esdk\build` and issue the command `ant`

Preparing the ESPA service capability interface

Set up the build environment

3. Edit the file `exampleproj\espa_sc_if\build.xml`.

Edit the properties described in "Adapting the build files for the modules" on page 15-40 to reflect the desired names.

Define the ESPA service capability module interface structure

4. In the file `exampleproj\espa_sc_if\idl\MyServiceCapability_if.idl`, change the interface structure to reflect the desired package structure. For example, the structure:

```
module com {

   module acompany {

      module espa {

         module mysctype {
```

This will create the ESPA service capability interface structure `com.acompany.espa.mysctype`

Interfaces to the ESPA service capability module

5. Add or change additional methods to be used in the interface between the SESPA service capability and the ESPA Service Capability. Use the definitions in `exampleproj\espa_sc_if\idl\MyServiceCapability_if.idl` as templates.

In the template file, the method `myMethod` is an example of how to define an asynchronous method invoked by an application on the ESPA service capability.

6. Edit the service name (used as ESPA service capability type) to the desired type, change the definition of `SERVICE_NAME`

   The service name is used as ESPA service capability type when registering the ESPA service capability in the SC manager. It is also used as in the SLAs in the **<scs>** tag.

Compilation of the ESPA service capability module interface

7. Compile the ESPA service capability interface by changing directory to `exampleproj\espa_sc_module_if\build\` and execute the command `ant`

   This compiles the ESPA service capability interface and generates Java stubs from the IDL interface.

Preparing the ESPA service capability module implementation

Set up the build environment

8. Edit the file `exampleproj\espa_sc_impl\build.xml`.

   Edit the properties described in "Adapting the build files for the modules" on page 15-40 reflect the desired names.

   Note: Always use a prefix in the jar name. The autostarted SLEE services are started in alphabetic order based on the jar name. All Service Capabilities should have a prefix so they are started after the network protocol plug-ins but before the SESPA modules.

Defining the OAM methods

9. Edit the files `exampleproj\espa_sc_impl\MyServiceCapabilityOAM.idl`. Define any additional OAM methods.

ESPA service capability module plug-in listener interface implementation

The plug-in interface implementation has to be updated with any changes to the plug-in interface defined in "Interfaces in the plug-in" on page 16-4.

10. Open the file implementing the call-back interface from the plug-in, `MyPluginListener_impl.java` in `exampleproj\spa_sc_impl\src\com\acompany\espa\mysctype\`

11. Adapt the import statement for the plug-in interfaces if there has been any changes.

12. Adapt the name of the class that the extends the listener, the name of the class is the same as as the classname for the callback interface, with the addition POA, for example:

```
public class MyPluginListener_impl extends
com.my_company.my_plugin.MyPluginListenerPOA
```

13. Adapt the method names if there have been any changes or additions.

ESPA service capability module service manager implementation

The interface has to be updated so it implements the `MyServiceCapabilityManager` interface as defined in "Interfaces to the ESPA service capability module" on page 17-3.

14. Open the file `MyServiceCapabilityManager_impl.java` in `exampleproj\espa_sc_impl\src\com\acompany\espa\mysctype`
15. Adapt the import statement for the plug-in interfaces if there have been any changes to them.
16. In the `doTask` method, change the line:

    ```
    MyPlugin myPlugin = MyPluginHelper.narrow(resource)
    ```

    to reflect any changes in names.
17. Adapt the method names to reflect the names in the interface definitions.

ESPA service capability module persistent storage

18. Open the file `MyServiceCapabilityPersistentStorage.java` in `exampleproj\espa_sc_impl\src\com\acompany\espa\mysctype.`
19. Edit the database table names, by changing GLOBAL_CONFIG_TABLE, LOCAL_CONFIG_TABLE`, and` ESPA_MYSERVICECAPABILITY_REQUEST_TIMEOUT_EVENT to a name for the tables holding persistent data.

ESPA service capability module context

20. Open the file `MyServiceCapabilityContext.java` in `exampleproj\espa_sc_impl\src\com\acompany\espa\mysctype.`
21. Change `POLICY_SERVICE_NAME` to a value that will be used to identify which Service Capability the Policy request originates from.

22. Change `PLUGIN_TYPE` to a value that will be used to identify which plug-in the Service Capability will connect to. This is the same type as the plug-in registered itself as, see "Plug-in type definition" on page 16-6.

Implement the Policy Enforcement Point

The Policy Enforcement Point (PEP) in the example ESPA module resides in `MyServiceCapabilityManager_impl.java`, in the method `myMethodInternal()`.

The service name is fetched from `MyServiceCapabilityContext.POLICY_SERVICE_NAME`.

The data that is to be accessible from rules executing at this PEP must be passed in the policy request. These reside in the `additionalParameters` of the `PolicyRequest` class.

Each data parameter is contained in an `AdditionalData` class, containing one `AdditionalDataValue` class and a data name which is an arbitrary string. The example code in the template shows how strings are used as additional data, but the `AdditionalData` class also supports the following data types:

- `intValue(int)`
- `longValue(long)`
- `stringValue(String)`
- `stringArrayValue(String[])`
- `booleanValue(boolean)`
- `shortValue(short)`
- `charValue(char)`
- `floatValue(float)`
- `doubleValue(double)`
- `intArrayValue(int[])`

It is up to the rule implementation to extract additional values according to their defined data types. The PEP implementation must also extract any modified data returned in the modified policy request (`modifiedRequest` in the example code).

Compilation of the ESPA service capability module implementation

Prior to this, the plug-in interface and the ESPA service capability module interface must have been built.

23. Compile the ESPA service capability module implementation by changing directory to `exampleproj\espa_sc_module_impl` and execute the command `ant`

Adapting the policy rules

24. Open the files:

    `exampleproj\policy\rules\app\ESPA_myservicecapability.ilr`

    `exampleproj\policy\rules\sp\ESPA_myservicecapability.ilr`

25. Depending on the policy implementation (the PEP) the two rules listed under the heading `START_APP_My_Service_Capability_Specific_Rules` and `START_SP_My_Service_Capability_Specific_Rules` might have to be changed.

    If the data format has been changed, the parameter

    `?dataString: getAdditionalDataStringValue` has to be updated accordingly.

    If the method name has been changed in the policy implementation, see "Implement the Policy Enforcement Point" on page 17-6, the parameter `?methodName.equals` has to be updated.

Installing the ESPA service capability module

This section describes how to install and deploy the ESPA service capability module that was created using the instructions in this section. For instructions on how to use the Management Tool and how to register an ESPA service capability module in more detail, see WebLogic Network Gatekeeper User's Guide.

1. Make sure you have access (by ftp directly to the file system) to the SLEE in that the ESPA service capability module shall be installed in.

2. Open the Management Tool and select the SLEE in which to install the ESPA service capability module.

3. In the Management tool, select the **SCS_Manager** service, and invoke the method **`addType`**. Use the type defined for the ESPA service capability.

   In the example, the SCS type is `MY_SCS_TYPE`. This is defined in the plug-in interface definition file, my_plug_in_if.idl in the constant `MY_SCS_TYPE`.

4. In the **SLEE_deployment** service, select **install**. Enter the URL to the jar-file in the field **ServiceJarURL** and click **Invoke**.

In the example, the path and file name is `file:///<drive>/exampleproj/espa_sc_module_impl/lib` and the name was defined in the build file property local.jarName.

Note: On Windows systems, use three (3) slashes prior to the disk name.

5. In the **SLEE_deployment** service, select **start**. Enter the SLEE name for the ESPA service capability module. Click **Invoke**.

   The name was defined in the build file property local.deployName.

6. In the **SLEE_deployment** service, select **activate**. Enter the SLEE name for the ESPA service capability module. Click **Invoke**.

Update Service Level Agreements (SLAs)

Any application and service provider using the newly created ESPA service capability need to have valid SLAs for the new ESPA service capability. The SLA-files need to be updated with service contracts for the ESPA service capability. For information on how to write the SLA-files, see WebLogic Network Gatekeeper User's Guide.

The SC name (entered in the `<scs></scs>` tag) is the name defined as service name in Step 6. on page 3.

Install policy rules

In order to enforce the SLA, load the policy rules defined in "Adapting the policy rules" on page 17-7 into Network Gatekeeper.

1. In the **`Policy`** service, select **loadApplicationRules**. Enter the URL to the application rule irl-file in the field **`irlUrl`** and the service name in the **serviceName** field. Click **Invoke**.

   In the example, the path and file name is
   ```
   file:///<drive>/exampleproj/policy/rules/app/ESPA_myservicecapability
   .ilr
   ```

   The service name is the name defined in Step 6. on page 3.

   Note: On Windows systems, use three (3) slashes prior to the disk name.

2. In the **Policy** service, select **loadServiceProviderRules**. Enter the URL to the service provider rule irl-file in the field **irlUrl** and the service name in the **serviceName** field. Click **Invoke**.

In the example, the path and file name is
`file:///<drive>/exampleproj/policy/rules/sp/ESPA_myservicecapability.ilr`

Creating an example Policy Utility

The following section provides a description on how to create an example PolicyUtility:

- General preparations
- Preparing the Policy utility
- Installing the Policy Utility
- Install policy rules
- Install Subscriber profile plug-in
- Provision data to the database

The example uses the data provided by the PEP, a rule will extract the relevant data and call the Policy utility. The Policy utility uses a Subscriber Profile plug-in to do a look-up in the database if a subscriber with a certain address is registered in the database. If defined, the Policy utility will allow the requests, otherwise deny it. The Policy utility has the following properties:

- Package: com.acompany.policy.util
- Rule used to call the Policy Utility class: `DenySubscriberNotExists`
- Methodname defined in the PEP: `myMethod`
- Rule example in: `bea\wlng21\esdk\policy\rules\sp\ESPA_myservicecapability.ilr`

General preparations

This is general preparations. Do not perform these two steps if they already have been performed when creating the network plug-in as described in "Creating an example network plug-in" on page 16-1.

1. Make sure the files for the Policy utility are copied to the directory `exampleproj`. That is, all files and directories in `\module_templates\policy_util_impl`
2. Change directory to `bea\wlng21\esdk\build` and issue the command `ant`

Preparing the Policy utility

Set up the build environment

3. Edit the file `exampleproj\policy_util_impl\build.xml`.

Edit the properties described in "Adapting the build files for the modules" on page 15-40 to reflect the desired names.

Defining the OAM methods

4. Edit the files `exampleproj\policy_util_impl\idl\MyPolicyUtilOAM.idl`. Define any additional OAM methods.

Policy utility implementation and rule files

The Policy utility implementation is a singleton class, that provides the public methodsubscriberExists. This method is called from the rule.

5. Adapt the template Policy utility singleton class MyPolicyUtility.java in `exampleproj\policy_util_impl\src\com\acompany\policy\util` to suit the specific needs. The template implementation uses a subscriber profile plug-in t check if a subscription exists, thus giving an example on how to interact with a subscriber profile plug-in and the plug-in manager.
6. Adapt the import statement for Policy Utility class in the rule file if there have been any changes.
7. Adapt the name of the class that is called in the rule file if there has been any changes.
8. Adapt the method names if there have been any changes or additions.

Installing the Policy Utility

This section describes how to install and deploy the Policy Utility that was created using the instructions in this section. For instructions on how to use the Management Tool in more detail, see WebLogic Network Gatekeeper User's Guide.

9. Make sure you have access (by ftp directly to the file system) to the SLEE in that the Policy utility shall be installed in.
10. Open the Management Tool and select the SLEE in which to install the Policy Utility.
11. In the **SLEE_deployment** service, select **install**. Enter the URL to the jar-file in the field **ServiceJarURL** and click **Invoke**.

    In the example, the path and file name is `file:///<drive>/exampleproj/policy_util_impl/lib` and the name was defined in the build file property local.jarName.

    Note: On Windows systems, use three (3) slashes prior to the disk name.
12. In the **SLEE_deployment** service, select **start**. Enter the SLEE name for the Policy Utility. Click **Invoke**.

    The name was defined in the build file property local.deployName.
13. In the **SLEE_deployment** service, select **activate**. Enter the SLEE name for the Policy Utility. Click **Invoke**.

Note: If a Policy utility class has been changed and needs to be reinstalled, uninstall the Policy Utility using the methods **deactivate**, **stop**, and **uninstall** in the **SLEE_deployment** service via the Management Tool. Then, install the new Policy Utility using the procedure described above and restart the SLEE.

Install policy rules

In order to load the rules that calls the Policy utility, follow the instructions in "Install policy rules" on page 17-8.

Install Subscriber profile plug-in

The rule uses the subscriber profile plug-in provided in `bea\wlng21\esdk\lib\b_db_sp_resource.jar`.

Install this plug-in according to the schema described in "Installing the Policy Utility" on page 18-3. The Plug-in has the SLEE name **`Plugin_subscriber_profile_DB`**.

Provision data to the database

Via the Management Tool, use the OAM methods **createSubscriber** and **`deleteSubscriber`** in the SLEE service **`Plugin_subscriber_profile_DB`** to add and delete users from the database. The Policy utility checks if the user exists in this database.

Creating an example SESPA module

The following section provides a description on how to create an example SESPA module:

- General preparations
- Preparing the SESPA service capability interface
- SESPA service capability module interface compilation
- Preparing the SESPA service capability module implementation
- Compilation of the SESPA service capability module implementation
- Installing the SESPA service capability module

The example SESPA module will use the interfaces defined for the example ESPA service capability module prepared in "Creating an example ESPA Service Capability module" on page 17-1. The SESPA module will have the following properties:

- package: `com.acompany.espa.mysctype`
- Interface used by the WESPA module for application-initiated requests, acting on the SESPA module: `MyServiceCapability`
- Interface used by the SESPA module for application-initiated requests, acting on the WESPA module: `MyServiceCapabilityListener`
- Methods implemented by the SESPA module, defined in the `MyServiceCapability` interface:
  - `myMethodWait`

  - `myMethod`
  - `enableNetworkTriggeredEvents`
  - `disableNetworkTriggeredEvents`

- Methods invoke by the WESPA service capability for application-initiated requests, defined in the `MyServiceCapabilityListener` interface, and implemented by the WESPA module:
  - `myMethodResult`
  - `myMethodError`
- Interface used by the WESPA module for application-initiated requests, acting on the SESPA module: `MyServiceCapability`
- Interface used by the SESPA module for application-initiated requests, acting on the WESPA module: `MyServiceCapabilityListener`
- Methods implemented by the SESPA module, defined in the `MyServiceCapability` interface:
  - `myMethodWait`
  - `myMethod`
  - `enableNetworkTriggeredEvents`
  - `disableNetworkTriggeredEvents`
- Methods invoke by the WESPA service capability for application-initiated requests, defined in the `MyServiceCapabilityListener` interface, and implemented by the WESPA module:
  - `myMethodResult`
  - `myMethodError`
- Interface used by the SESPA module for network-triggered requests, acting on the WESPA module: `MyServiceCapabilityNetworkTriggeredEventListener`
- Methods used by the SESPA module, defined in the `MyServiceCapabilityNetworkTriggeredEventListener` interface:
  - `myDeliverNetworkTriggeredEventMethod`
  - `deactivate`

General preparations

This is general preparations. Do not perform these two steps if they already have been performed when creating the Service Capability as described in "Creating an example ESPA Service Capability module" on page 17-1.

1. Make sure the files for the SESPA Module and the SESPA interfaces are copied to the directory `exampleproj`. That is, all files and directories in `module_templates\sespa_sc_impl and module_templates\s`espa_sc_if.

2. Change directory to `bea\wlng21\esdk\build` and issue the command `ant`

Preparing the SESPA service capability interface

Build environment

3. Edit the file `exampleproj\sespa_sc_if\build.xml`.

   Edit the properties described in "Adapting the build files for the modules" on page 15-40 to reflect the desired names.

SESPA service capability module interface structure

4. Rename the directories to reflect the desired package structure. Also change the package definitions accordingly.

SESPA service capability module interfaces

5. Add any additional methods to be used in the interface between the SESPA service capability and the WESPA module. Use the method definitions in `MyServiceCapability.java`, `MyServiceCapabilityListener.java` and `MyServiceCapabilityNetworkTriggeredEventListener.java` as templates.

   In the template file, the method `myMethod` is an example of how to define an asynchronous method invoked by the WESPA module on the SESPA module and `myMethodWait` is an example of a synchronous method.

   An `assignmentID` is given when invoking the method `myMethod`. The ID connects an invocation to `myMethod` and the corresponding invocation of `myMethodResult` or `myMethodError`.

SESPA service capability module interface compilation

6. Compile the SESPA service capability interface by changing directory to `exampleproj\sespa_sc_if\` and execute the command `ant`

Preparing the SESPA service capability module implementation

Set up the build environment

7. Edit the file `exampleproj\sespa_sc_impl\build.xml`.

   Edit the properties described in "Adapting the build files for the modules" on page 15-40 reflect the desired names.

Defining the OAM methods

8. Edit the files `exampleproj\espa_sc_impl\MyServiceCapabilityOAM.idl`. Define any additional OAM methods and adapt the structure to the structure defined in "SESPA service capability module interface structure" on page 19-3

SESPA service capability module structure

9. Rename the directories to reflect the desired package structure. Also change the package definitions accordingly.

SESPA service capability module listener implementation

The listener interface implementation has to be updated so it implements the ESPA service capability module listener interface as defined in "Interfaces to the ESPA service capability module" on page 17-3.

10. Open the file `MyESPAServiceCapabilityListener_impl.java` in `exampleproj\sespa_sc_impl\src\com\acompany\sespa\mysctype`.

11. Adapt the import statement for the ESPA Service Capability interface if there has been any changes.

12. Adapt the name of the class that the extends the listener, the name of the class is the same as as the classname for the callback interface, with the addition POA, for example:

```
public class MyESPAServiceCapabilityListener_impl extends
MyServiceCapabilityListenerPOA
```

SESPA service capability module implementation

The interface has to be updated so it implements the `MyServiceCapability` interface as defined in "SESPA service capability module interfaces" on page 19-3.

13. Open the file `MyServiceCapabilityImpl.java` in `exampleproj\sespa_sc_impl\src\com\acompany\sespa\mysctype.`

    Adapt the import statement for the ESPA Service Capability module manager if there has been any changes to them.

14. IAdapt the method names to reflect the names in the interface definitions.

SESPA service capability module service

15. Open the file `MyServiceCapabilityService.java` in `exampleproj\sespa_sc_impl\src\com\acompany\sespa\mysctype.`
16. Define the name that the WESPA modules will use to fetch the SESPA Service Capability implementation. The name is defined when the SESPA Service Capability registers itself in SLEE Common Loader, SleeCommonLoader.getInstance().addObject(...).

Compilation of the SESPA service capability module implementation

Prior to this, the plug-in interface, ESPA service capability module interface, and the SESPA service capability interface must have been built.

17. Compile the SESPA service capability module implementation by changing directory to `exampleproj\sespa_sc_impl` and execute the command `ant`

Installing the SESPA service capability module

This section describes how to install and deploy the SESPA service capability module that was created using the instructions in this section. For instructions on how to use Management tool and how to register an SESPA service capability module in more detail, see WebLogic Network gatekeeper User's Guide.

1. Make sure you have access (by ftp directly to the file system) to the SLEE the SESPA service capability module shall be installed in.

2. Open the Management Tool and select the SLEE in which to install the SESPA service capability module.
3. In the **SLEE_deployment** service, choose **install**. Enter the URL to the jar-file in the field **`ServiceJarURL`** and click **Invoke**.

   In the example, the path and file name is `file:///<drive>/exampleproj/sespa_sc_impl/lib` and the name was defined in the build file property local.jarName.

Note: On Windows systems, use three (3) slashes prior to the drive name.

4. In the **SLEE_deployment** service, choose **start**. Enter the SLEE name for the SESPA service capability module. Click **Invoke**.

   The name was defined in the build file property local.deployName.
5. In the **`SLEE_deployment`** service, choose **activate**. Enter the SLEE name for the SESPA service capability module. Click **Invoke**.

   The name was defined in the build file property local.deployName.

Creating an example WESPA module

The following section provides a description on how to create an example WESPA module:

- General preparations
- Preparing the WESPA service capability interface
- WESPA service capability module interface compilation
- Preparing the WESPA service capability module implementation
- Compilation of the WESPA service capability module implementation
- Installing the WESPA service capability module

The example WESPA module will use the interfaces defined for the example SESPA service capability module prepared in "Creating an example SESPA module" on page 19-1. The WESPA module will have the following properties:

- package: `com.acompany.wespa.mysctype`
- Interface used by the client, acting on the WESPA module: `MyServiceCapability`
- Interface used by the WESPA module, acting on the client: `MyServiceCapabilityListener`
- Method implemented by the WESPA module for application-initiated requests, defined in the `MyServiceCapability` interface:
  - `myMethod`

  – `myMethoddWait`
  - `enableNetworkTriggeredEvents`
  - `disableNetworkTriggeredEvents`

- Methods invoked by the WESPA service capability for application-initiated requests, defined in the `MyServiceCapabilityListener` interface, and implemented by the client:
  – `myMethodResult`
  – `myMethodError`
  - `deactivate`
- Interface used by the WESPA module, acting on the client: `MyServiceCapabilityNetworkTriggeredEventListener`
- Methods invoked by the WESPA service capability for network-triggered requests, defined in the `MyServiceCapabilityNetworkTriggeredEventListener` interface, and implemented by the client:
  – `myDeliverNetworkTriggeredEventMethod`
  – `deactivate`

General preparations

This is general preparations. Do not perform these two steps if they already have been performed when creating the plug-in, ESPA service capability, or SESPA module in the previous chapters.

1. Make sure the files for the WESPA Module and the WESPA interfaces are copied to the directory `exampleproj`. That is, all files and directories in `module_templates\wespa_sc_impl and module_templates\w`espa_sc_if.
2. Change directory to `bea\wlng21\esdk\build` and issue the command `ant`

Preparing the WESPA service capability interface

Build environment

3. Edit the file `exampleproj\wespa_sc_if\build.xml`.

   Edit the properties described in “Adapting the build files for the modules” on page 15-40 to reflect the desired names.

4. Edit the Java to WSDL mapping. In all `<axis-java2wsdl>` tags, adapt the names, and locations of the WSDL files to be generated, package names of the implementing classes, and the namespace to use.

WESPA service capability module interface structure

5. Rename the directories to reflect the desired package structure. Also change the package definitions accordingly.

WESPA service capability module interfaces

6. Add any additional methods to be used in the interface between an application and the WESPA module. Use the method definitions in `MyServiceCapability.java`, `MyServiceCapabilityListener.java` and `MyServiceCapabilityNetworkTriggeredEventListener.java` as templates.

   In the template file, the method `myMethod` is an example of how to define an asynchronous method invoked by the WESPA module on the SESPA module and `myMethodWait` is an example of a synchronous method.

   There are dummy implementations of the interfaces, `MyServiceCapability_dummy.java` and `MyServiceCapabilityListener_dummy.java`. Adapt the dummy implementations to the interface definitions.

   An `assignmentID` is given when invoking the method `myMethod`. The ID connects an invocation to `myMethod` and the corresponding invocation of `myMethodResult` or `myMethodError`.

WESPA service capability module interface compilation

7. Compile the WESPA module interface by changing directory to `exampleproj\wespa_sc_if` and execute the command `ant`

Preparing the WESPA service capability module implementation

Set up the build environment

8. Edit the file `exampleproj\wespa_sc_impl\build.xml`.

9. Edit the properties described in "Adapting the build files for the modules" on page 15-40 reflect the desired names.

10. Edit the war-file name. Change the property local.warName to the name of the war file to be deployed in Tomcat.

WESPA service capability module structure

11. Rename the directories to reflect the desired package structure. Also change the package definitions accordingly.

SESPA service capability listener implementation

The listener interface implementation has to be updated so it implements the SESPA service capability module listener interface as defined in "SESPA service capability module interfaces" on page 19-3.

12. Open the file `MyServiceCapabilityListenerImpl.java` in `exampleproj\wespa_sc_impl\src\com\acompany\wespa\mysctype`.

13. Open the file `MyServiceCapabilityListenerImpl.java` in `exampleproj\wespa_sc_impl\src\com\acompany\wespa\mysctype`.

14. Adapt the import statement for the SESPA Service Capability interface if there has been any changes.

15. Adapt the name of the class that the implements the listener, if necessary.

WESPA service capability SOAP binding implementation

The interface has to be updated so it implements the `MyServiceCapability` interface as defined in "WESPA service capability module interfaces" on page 20-3.

16. Open the file `MyServiceCapabilitySoapBindingImpl.java` in `exampleproj\wespa_sc_impl\src\com\acompany\wespa\mysctype`.

17. Adapt the import statement for the SESPA Service Capability interface if there has been any changes.

18. The nameLookUp() method must be adapted to use fetch the SESPA implementation from SLEE Common loader. The lookup i performed by name. The name was given in "SESPA service capability module service" on page 19-5. The cast from Object obj must also be adapted to any changes in package structures and class names.

19. Open the file `web.xml in wespa_sc_impl\deploy\server`
20. Edit the reference to the servlet implementation. Adapt the classnames and package definitions in the tag <servlet> if necessary.

Compilation of the WESPA service capability module implementation

Prior to this, the plug-in interface, SESPA and WESPA module interfaces must have been built.

21. Compile the WESPA module implementation by changing directory to `exampleproj\wespa_sc_impl` and perform the command `ant`

Installing the WESPA service capability module

This section describes how to install and deploy the WESPA module that was created using the instructions in this section. For detailed instructions on how to use the Management Tool, see WebLogic Network Gatekeeper User's Guide.

1. Copy the generated war file to the `/<install dir>/slee/bin/autowar` directory in the Network Gatekeeper.
2. Open the Management Tool and select the SLEE running on the server the file was copied to.

3. In the **Embedded_tomcat** service, choose **addContext**. Enter the following parameter data:

Parameter	Description
contextPath	The context path to be used. For example: `/exampleContext` The Web Service will be reached in the following URL: http://<IP-address>:<port>/<contextPath>/services/<method>

docBase	Document root. Can be a war-file or a directory. Can be specified with an absolute or a relative path name or an URL. In the example the files are stored in:
	`/usr/local/slee/bin/autowar/`
	`<name of war file>.war`

useCookies	Enable cookies (TRUE/FALSE) for session handling. Use FALSE.
autostart	Start this context during next service restart (TRUE/FALSE). Use TRUE.

4. Click **Invoke**.

   The WESPA module is started.

Creating an example application

The following section provides a description on how to create an example application that uses the interfaces defined for the example WESPA module prepared in "Creating an example WESPA module" on page 20-1:

- General preparations
- Preparing the application
- Compilation of the test application
- Running the application

The application will have the following properties:

- package `com.acompany.test`
- Interface used by the test application client, acting on the WESPA module: `MyServiceCapability`
- Interface used by the WESPA module, acting on the test application: `MyServiceCapabilityListener`
- Interface used by the WESPA module to deliver network triggered notifications, acting on the test application: `MyServiceCapabilityNetworkTriggeredEventListener`
- Method implemented by the WESPA module, defined in the `MyServiceCapability` interface:
  - `myMethod`

  - `myMethodWait`
  - `enableNetworkTriggeredEvents`
  - `disableNetworkTriggeredEvents`
- Methods invoke by the WESPA service capability, defined in the `MyServiceCapabilityListener` interface, and implemented by the test application:
  - `myMethodResult`
  - `myMethodError`
- Methods invoke by the WESPA service capability, defined in the `MyServiceCapabilityNetworkTriggeredEventListener` interface, and implemented by the test application:
  - `myDeliverNetworkTriggeredEventMethod`

General preparations

This is general preparations. Do not perform these two steps if they already have been performed when creating the SESPA Service Capability as described in "Creating an example SESPA module" on page 19-1.

1. Make sure the files for the application are copied to the directory `exampleproj`. That is, all files and directories in `module_templates\client_impl`.
2. Change directory to `bea\wlng21\esdk\build` and issue the command `ant`

Preparing the application

3. Open the file `exampleproj\client_impl\build.xml`.
4. Edit the location of the `MyServiceCapability.wsdl` and `MyServiceCapabilityListener.wsdl` files. Change the properties `myScWsdl and myScListenerWsdl`.
5. Edit the location of the `MyServiceCapabilityNetworkTriggeredEventListener.wsdl` file. Change the property `myScNetworkTriggeredEventListenerWsdl`.
6. Edit mapping namespaces for the WSDL generation. Change the namespace mapping properties if there have been any changes to classnames or package structure.
7. Edit the location of the deployment descriptor, in the replaceregexp tag, for the application's web service if there have been any changes to classnames or package structure.

Adapt the application to use the example WESPA module interfaces

8. Rename the directories to reflect the desired package structure. Also change the package definitions accordingly.

Compilation of the test application

Prior to this, the WESPA module must have been built.

9. Compile the application by changing directory to `exampleproj\client_impl` and execute the command `ant`

Running the application

This section describes how to prepare the start script, register the application in Network Gatekeeper and how to set up the Service Level Agreement.

Prepare start script

10. Open the file `exampleproj\client_impl\runMe.bat` and edit the `HOST` variable to fit the port that Network Gatekeeper listens to.

11. Edit the URL for the Web Service for the new service capability. Change:

    ```
    http://%HOST%/wespa_myservicecapability/services/MyServiceCapability
    ```

    To reflect any changes done in the deployment descriptor for the WESPA module.

    The URL must be according to where the WESPA module was deployed, see "Installing the WESPA service capability module" on page 20-5.

Register application data

The application has a text based interface through which the login is performed. The following login data has to be registered in the Network Gatekeeper before the application can log in:

- Service provider ID - for example: `test_sp`
- Application ID - for example: `test_appl`
- Application instance group ID - for example: `test_group`

- Application instance group password - for example: `test_group_pwd`

For information on how to perform the actual application registration, see WebLogic Network Gatekeeper User's Guide.

Both the service provider and application level SLAs must allow the application to use the **ESPA_example_sc** created in "Creating an example ESPA Service Capability module" on page 17-1.

Run the application

12. Execute the start script `exampleproj\client_impl\runMe.bat`
13. In the application, make sure that the application is configured to use the Access Web Service and the Web Service exposed by the WESPA SC module.

Make sure that the network simulator application is started and configured as described in "Creating an example network simulator" on page 22-1.

Print the configuration to make sure the server and login information is set up correctly, otherwise configure login information and server details.

Perform a login.

Use menu option `5. Start MySc test, synchronous mode` to send an application initiated synchronous request to the WESPA SC.

Use menu option `6. Start MySc test, asynchronous mode` to send an application initiated asynchronous request to the WESPA SC.

Provide an assignment ID that will be provided to the WESPA SC. The destination address and the payload data are automatically provided by the test application.

In both the synchronous and the asynchronous call the test application will output the outcome of the request.

The payload data, together with the destination address and the transaction ID is propagated through the modules and delivered to the network simulator which returns a reply. If there is an error the application will printout an error message. A network transaction ID provided by the network simulator is returned.

When an application performs an application triggered request, data about the request is printed in the console of the network simulator.

In order for the application to receive network triggered notifications, it must enable notifications for a specific address.

Use menu option `7. Enable network triggered events` to enable the notifications.

An listener ID is returned. Use this ID when disabling network triggered events.

Use menu option `8. Disable network triggered events` to enable the notifications.

See "Creating an example network simulator" on page 22-1 for information on how to build and run the network simulator application.

Creating an example network simulator

The following section provides a description on how to create an example network simulator that uses the interfaces defined for the network node that communicates with the Web Services part of the plug-in, see "Creating an example network plug-in" on page 16-1:

- General preparations
- Preparing the network simulator application
- Compilation of the network simulator application
- Running the network simulator application

The network simulator application will have the following properties:

- package `com.acompany.test`
- Interface used by the Web Services plug-in module, acting on the network simulator application:
  `NetworkInterface.wsdl`
- Interface used by the network simulator application to deliver network triggered notifications, acting on the Web Services part of the network plug-in:
  `NetworkTriggeredEventListener.wsdl`
- Interface used by the Web Services plug-in module, in response to an network triggered event: `NetworkTriggeredEventResultListener.wsdl`
- Method used by the Web Services plug-in module, defined in `NetworkInterface.wsdl`, implemented by the network simulator application

  - `myNetworkMethod`

- Method used by the Web Services plug-in module, defined in the `NetworkTriggeredEventResultListener.wsdl`, and implemented by the test application:
  - `myDeliverNetworkTriggeredEventMethodResult`
  - `myDeliverNetworkTriggeredEventMethodError`
- Methods invoked by the network simulator application, defined in `NetworkTriggeredEventListener.wsdl`, and implemented by the Web Services part of the plug-in:
  - `myDeliverNetworkTriggeredEventMethod`

General preparations

This is general preparations.

1. Make sure the files for the network simulator application are copied to the directory `exampleproj`. That is, all files and directories in `module_templates\client_impl`.
2. Change directory to `bea\wlng21\esdk\build` and issue the command `ant`

Preparing the network simulator application

3. Open the file `exampleproj\client_impl\build.xml`.
4. Edit the location of the `NetworkInterface.wsdl`, `NetworkTriggeredEventListener.wsdl`, and `NetworkTriggeredEventResultListener.wsdl` files. Change the properties `networkTriggeredEventListenerWsdl`, `networkTriggeredEventResultListenerWsdl` and `networkInterfaceWsdl`.
5. Edit mapping namespaces for the WSDL generation. Change the namespace mapping properties if there have been any changes to classnames or package structure.
6. Edit the location of the deployment descriptor, in the replaceregexp tag, for the application's web service if there have been ably changes to classnames or package structure.

Compilation of the network simulator application

7. Compile the network simulator application by changing directory to `network_simulator_impl` and execute the command `ant`

Running the network simulator application

This section describes how to prepare the start script for the network simulator application.

Prepare start script

8. Open the file `exampleproj\network_simulator_impl\runMe.bat` and edit the `HOST` variable to fit the port that Network Gatekeeper listens to.

9. Edit the namesspace for the Web Service that implements the Web Service used by the network simulator application and implemented in the Web Services part of the plug-in. Adapt:

   `http://%HOST%/wplugin/services/NetworkTriggeredEventListener`

   To reflect any changes done when deploying the Web Services part of the plug-in.

   The namespace must be according to where the Web Services part of the plug-in was deployed, see "Installing the Web Service plug-in" on page 16-10.

Run the network simulator application

10. Execute the start script `exampleproj\network_simulator_impl\runMe.bat`

11. In the network simulator application, make sure that the network simulator is configured to use the Web Service that the Web Services part of the plug-in implements for listening to network initiated events. By default, the Web service is deployed in `http://<Tomcat URL>:8080/wplugin/services/NetworkTriggeredEventListener`

Use menu option `5 -Deliver network triggered event` to send a network triggered event to the Web Services part of the plug-in. Provide the payload data, originating address, destination address, and a network transaction ID. Make sure that the test application has enabled network triggered events for the destination address.

The data provided, together with an originating address, a destination address and a network transaction ID is propagated through the modules and delivered to the application. If there is an error the network simulator application will printout an error message.

When an application performs an application triggered request, data about the request is printed in the console of the network simulator.

See “Creating an example application” on page 21-1 for information on how to build and use the example application.

Release notes

What is the Extension SDK for Network Gatekeeper 2.1

The Extension SDK for Network Gatekeeper is a new product that allows developers to create extensions to the WebLogic Network Gatekeeper.

Extensions can be made as completely new traffic flows from northbound interfaces to network protocol plug-ins. Extensions can also be made to the northbound interfaces, while reusing the existing Service capabilities and creating new northbound interfaces that uses the existing interfaces provided by the SESPA part of the Service Capabilities. Extension can also be created on the network protocol layer, that is creating new plug-ins that implement the existing plug-in interfaces.

A set of code templates and rule file templates, together with a build environment for the extensions are included in the Extension SDK.

Notes on installation

In order to deploy extension that are created using the Extension SDK in a WebLogic Network Gatekeeper, the following patches are necessary to Network Gatekeeper 2.1:

- x_sespa_access.jar, patch version: R_WLNG_2_1_0_4
- slee.jar, patch version: R_WLNG_2_1_0_5

Interface changes

The Extension SDK, and software modules developed using the Extension SDK use interfaces internal in the Network Gatekeeper as well as utility classes specific for the Extension SDK.

The code template structure for the ESPA and SESPA modules may change in the next release of the Extension SDK. The current structure will however continue to be supported.

Documentation

- The documentation for Extension SDK for WebLogic Network Gatekeeper 2.1, is available on e-docs. The URL is password protected, the login details are provided when ordering the software.

Operating system and third party software versions

Operating system

The Extension SDK has been tested using the following configurations.

- HP-UX 11.23
- SunOS 5.9 (Solaris 9)
- Red Hat Enterprise Linux AS release 3
- Windows XP

Java

For HP-UX: sdk14_14207_1122, version 1.4.2.07. JavaTM for HP-UX 11i Out-of-Box tool: 2.0.3

For Linux: jrockit-j2sdk1.4.2_05-linux-ia32

For Sun Solaris: j2sdk-1_4_2_04-solaris-sparc

For Windows: j2sdk-1_4_2_04-windows-i586-p

ORB

Orbacus 4.1.2 or 4.3.

Orbacus is not provided with the Extension SDK. A developer needs to acquire this separately. Visit http://www.iona.com for more information.

Ant

Ant 1.6.5. Ant is provided with the Extension SDK.

Known Issues

The following section describes known issues, and notes about these issues, in Extension SDK for Network Gatekeeper 2.1.

Change Request Number	Description
CR256059	Login session validity period checks must be disabled in the Network Gatekeeper in order to support network triggered events. When having login session validity period checks enabled in the Network Gatekeeper, the network triggered part of the example provided in the module templates will not work. See Network Gatekeeper User's guide for information on how to disable checking of validity periods for the login tickets.