Functional Specification for Container Integration

Author(s): Peter.Danielsson@ericsson.com, Eltjo.Boersma@ericsson.com
Version: prel A4

1 Introduction

<List proposed feature(s). Introduce the basic vocabulary. Why is this interesting? List capabilities that may be normally expected, but are not being supported. Are there any limitations and caveats that need to be disclosed?>

1.1 Structure

This document is part of a the overall function specification OFS:

1. Functional Specification for SailFin Administration
   Author(s): yamini@sun.com
   Contributors: Irfan A, Vijay G
   Version:0.5

1.2 Features

<List all requirements and features you are implementing. List those which may be normally expected to be implemented but are not.>

1.2.1 Internationalized Logging

All logging in the SEVERE and WARNING level should be Internationalized. The actual log-messages should be stored in a properties-file.

1.2.2 Call Flow and SIP Message Inspection

Call Flow can be seen as detailed logging of all the transactions going in and out of the container. This includes the SIP Messages, that has to be parsed and the important information in all messages logged as well.

1.2.3 Monitoring

Statistics collected in the Saifin code has to be reported using the same or similar mechanisms as for the rest of Glassfish.

1.2.4 Configuration

Configuration should be dynamic and allow (most) changes without having to restart the server.

1.2.5 Application Verifier

Support for SIP applications (.sar-files) has to be added to the application verifier

2 Design Overview

2.1 General design concerns

Container integration is to integrate the sip container with (mostly cross cutting functionality) Glassfish frameworks such as Config and Monitoring. The intention is to do so without introducing dependencies between Glassfish and Sailfin. However also the reverse is not wanted, that is no direct dependencies. Therefore the integration module will deal with all direct dependencies with Glassfish as depicted in the diagram below (Figure 1). (Note that the admin module is extension code to glassfish to introduce Sailfin to Glassfish and obviously also depends on Glassfish.)

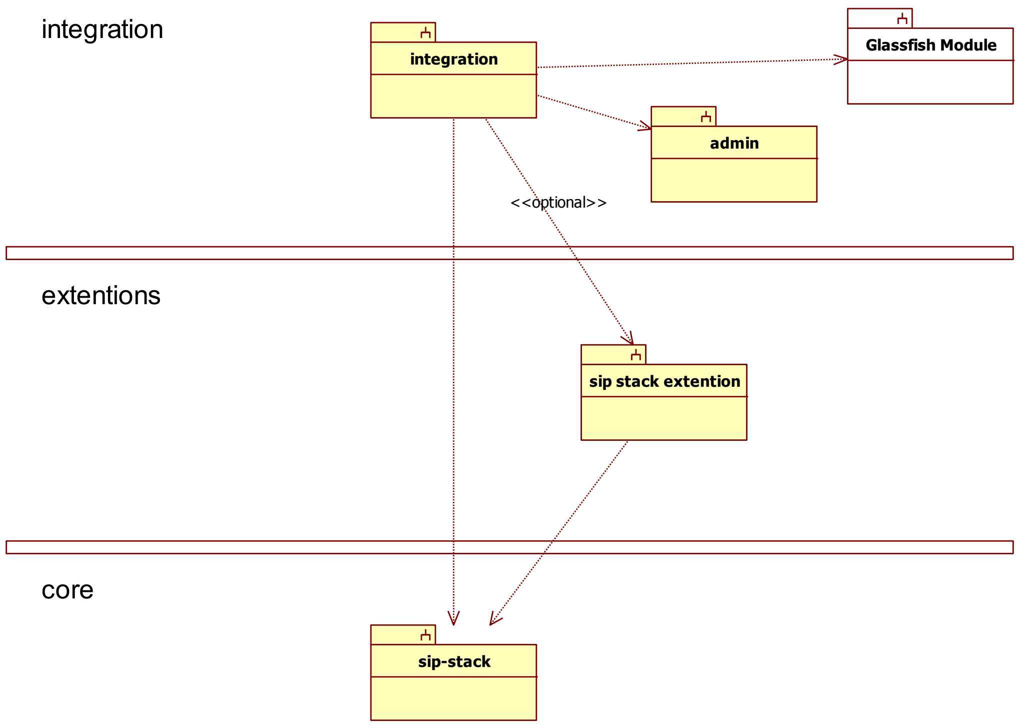

Figure 1: Dependencies between the modules, integration is to resolve dependency with Glassfish.

The integration module will strive to us the same pattern to resolve this dependency issue. The pattern is to use dependency injection to infuse by integration implemented realizations of interfaces that are specified by the sip-stack module. The dependency injection is applied to factories that are used within the sip-stack to obtain an instance implementing a certain interface. Although the infused realization of the interface depends on Glassfish the user of the interface does not. See the diagram below for the pattern (Figure 2).

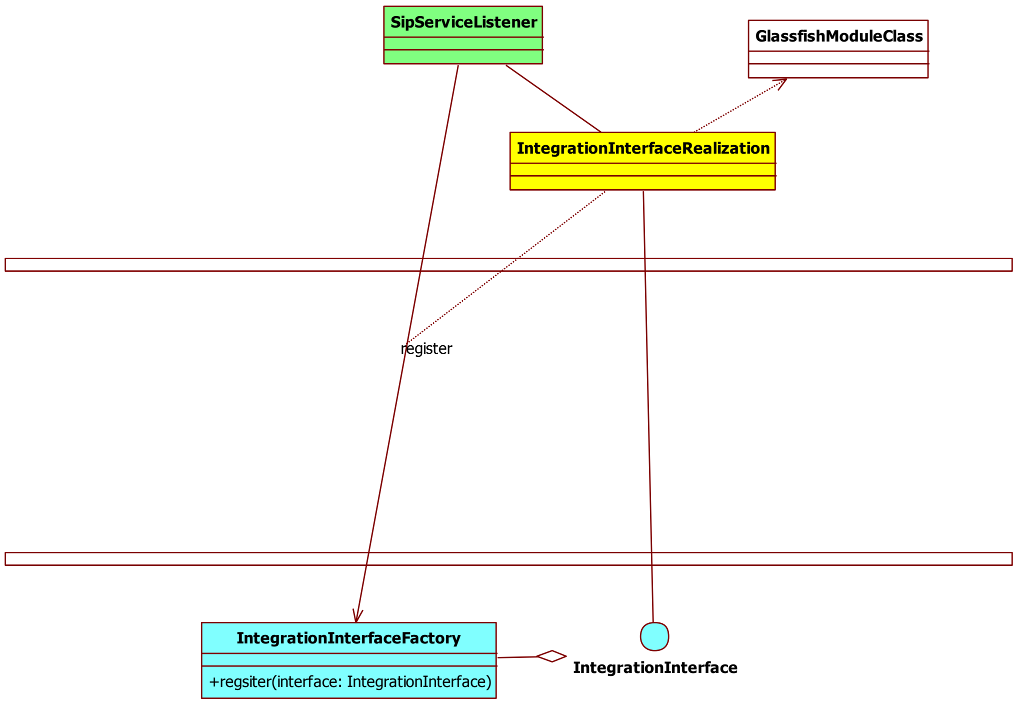
SipServiceListener
GlassfishModuleClass
IntegrationInterfaceRealization
register
IntegrationInterfaceFactory
+regsiter(interface: IntegrationInterface)
IntegrationInterface

Figure 2: Integration pattern, realizations are injected into factories to resolve dependency issues.

2.2 Internationalized logging

This part does not really require any new design. The design already available in Glassfish is sufficient. It just has to be applied to the Sailfin code.

The issues needed to be documented are what log messages should be used and what keys are used in the code.

We need to follow the logging guidelines of GF: GlassFish Log guidelines
This explains:

- General logging rules to follow.
- When to apply localization (for what log levels).
- Log resource bundle generation tool for exception logging.

Unclear is the usage of the message-catalog, seems that it is not used yet or is out dated.

2.2.1 Message id convention

The sip-container in total should get a message id prefix:

- `sip`

Within this scope we can define a own naming hierarchy. Suggestion us to have the following:

- `sip.<module><short message description>`
- `sip.stack.<layer><short message description>`

This may be a bid too fine grained. Other domains use `<prefix>.<short message description>`.

2.2.2 Logger name convention

Logger names should be sensible.

If we follow other GF containers we get the following:

- `javax.enterprise.system.container.sip`

See for this the following file `com.sun.logging.LogDomain` (the base logger with in GF is somehow `javax`). Currently `SipContainer` is used in the sip container.

Decided on logger name: `javax.enterprise.system.container.sip`.

2.3 Call Flow and SIP Message Inspection

2.3.1 Reporters

Call Flow reporting and SIP Message Inspection is handled by classes implementing the `Reporter` interface. Currently the `reporter`-classes are being called with the instance of the current `Layer` in the SIP stack and the current SIP request or SIP response. The reporters can be attached to any layer in the SIP stack and will be called before control is transferred to that layer.

2.3.2 Extensions to the Layers

To handle the reporters the `Layer` interface has been modified to include the registering of the reporter and to access the registered reporters.

2.3.3 Registering the reporters

Reporters are initialized at the same time as the SIP stack. An extra argument (`reporters`) to the layer in the `dispatcher.xml` tells what reporters will be called for that layer. It is possible to add new reporters if they are put in the package `org.jvnet.glassfish.comms.admin.callflow.reporter` and implements the `Reporter` interface.

2.3.4 Call Flow Reporting

Currently call flow is only reported at the incoming calls. This is handled by the `CallFlowReporter` and to enable reporting the CallFlowAgent must be enabled.

2.3.5 SIP Message Inspection

A simple inspection is being done by the reporter `SipMessageReporter`. This reporter simply logs the requests and responses passing through a layer to the log file using `LogLevel.FINE.`

2.4 Monitoring

2.4.1 Monitoring Managers

Monitoring of the SIP stack is handled by separate monitoring managers that corresponds to different layers in the stack. The monitoring managers collects statistics from the layers and presents them to the Glassfish monitoring framework. The available monitors are *network, manager, session manager, transaction manager* and *overload protection manager*.

2.4.2 Registering the Monitoring Managers

When the monitoring framework in Glassfish is initialized the monitoring managers for the SIP stack is being registered as well. This is handled by a call to `SipMonitoringManagerImpl.registerAllStats().`Due to the startup order the SIP stack is not yet initialized when the Monitoring Managers are registered so after initialization the layers in the SIP stack is being inspected and connected to corresponding monitoring managers.

2.4.3 New MonitoringObjectTypes

To register the different Monitoring Managers in the Glassfish monitoring framework the class `MonitoriedObjectType` has been extended to allow more types. This is utilized when registering a monitoring Manager by generating a new type based on the name of the manager.

2.4.4 Collected statistics in the Sip Container

2.4.4.1 NetworkManager

- Invalid Sip Messages
- Received Sip Requests
- Received Sip Responses
- Sent Sip Requests
- Sent Sip Responses

2.4.4.2 OverloadProtectionManager

- Overload Rejected Sip Responses
- Overload Rejected Http Responses

2.4.4.3 SessionManager:

- Failed Sip Dialogs
- Expires Sip Dialogs
- Successful Sip Dialogs
- Total Sip Dialog Count
- Total Sip Dialog Life Time
- Concurrent Sip Dialogs

2.4.4.4 TransactionManager:

- Sip Client Counters
- Sip Server Transactions
- Total Sip Transaction Time
- Total Sip Transaction Count

2.5 Configuration

2.5.1 Deployables

2.5.1.1 Application Router

See deployment module function specification.

2.5.1.2 Sip stack extensions

The sip stack extension plugins jars go on the classpath.

Sip stack layer configuration will be refactored to allow an overriding mechanism on how the sip-stack layers are configured. In the current implementation the LayerHandler class has several responsibilities among which; maintaining the Layer model, parsing the layer configuration (dispatcher.xml).

The picture below shows how the sip stack layer configuration will be refactored:

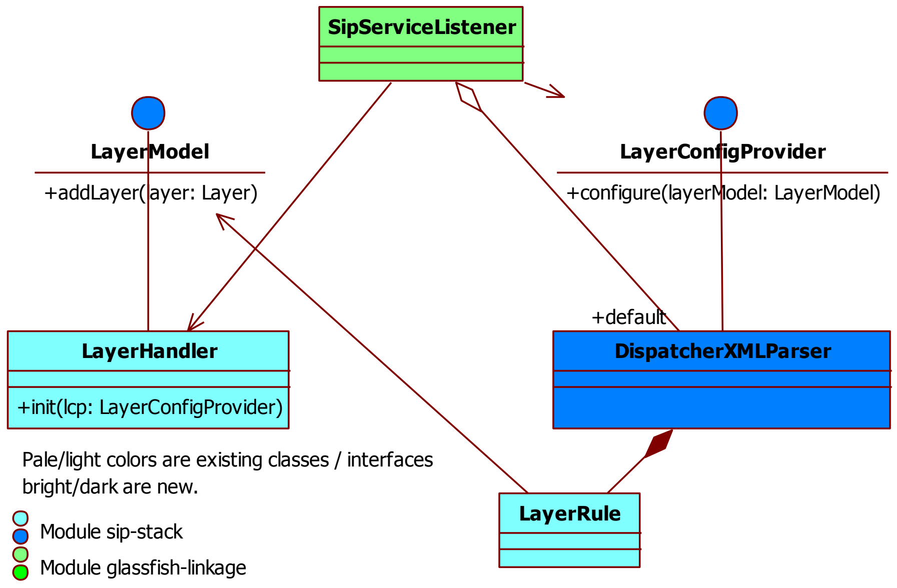

Figure 3: Class diagram depicting the refactoring of the LayerHandler to provide means to override the layer configuration (dispatcher.xml) that is necessary to allow extensibility of the sip-stack.

The LayerHandler still exist but its parser responsibility will be factored out as the default implementation of a LayerConfigProvider. To allow proper dependency between the LayerConfigProvider and the LayerHandler it implements a LayerModel interface.

The default LayerConfigProvider can be overridden by setting a property on the sip-container:

- `sip.extention.layer.config.provider`

The default LayerConfigProvider 'DispatcherXMLParser' can be redirected to read a non pre-packaged dispatcher.xml by setting the following property on the sip-container:

- `sip.extention.default.layer.config.provider.dispatcher.xml.location`

2.5.2 Configurables

2.5.2.1 Domain xml elements

The new sip related elements of the domain.xml as described in the domain.dtd. Will result among others into new ConfigBeans, ElementChangeEvents and ElementChangeEventListeners as a result of an generation process using the domain.dtd as input. The sip containers' current way of configuring has to be adjusted to couple it to the Glassfish Config model. See Figure 4.

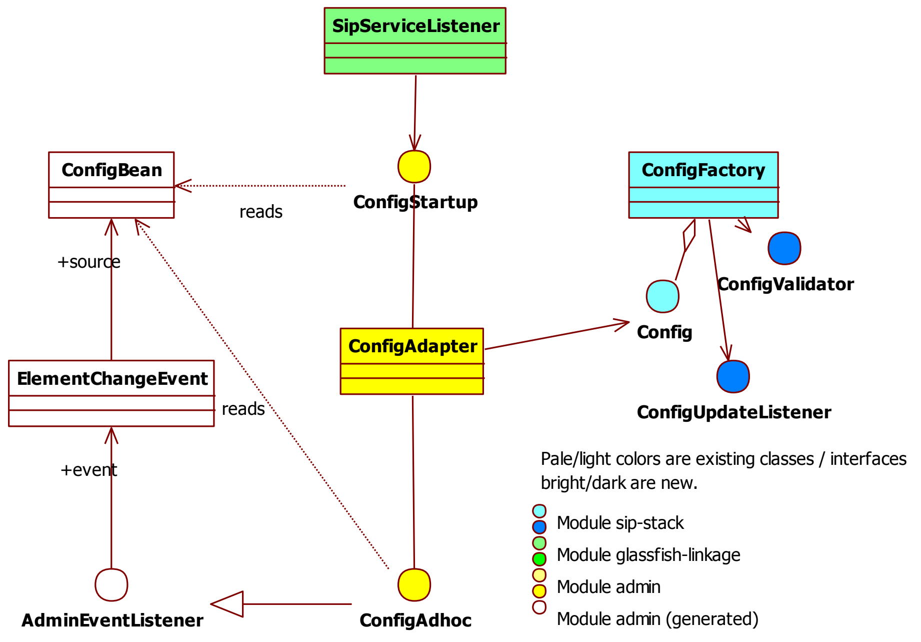

Figure 4: Class diagram depicting how the Glassfish Config Model is coupled to sip container configuration and the changes to the sip container configuration (not completely depicted).

Therefore the current sip-container configuration has to be updated to allow this to work. The refactoring mainly strips the old JMX based administering support and the runtime polling mechanism that accompanies it. See Figure 5 for more details on the refactoring. After refactoring remains an internal interface is used by the Sailfin admin module to realize the coupling. This also makes sure that the proper dependencies are honored (the admin depends on the sip-stack not the other way around.)

Two phases can be distinguished. There is a startup phase; where the ConfigBeans have to read and the sip-stack (container in general) has to be configured /initialized accordingly. The realization of the ConfigStartup takes care of this for this phase.

After the startup phase the sip-container is fully configured and initialized and enters the active state. Admin users may change the Configuration at the Console or issue CLI command that update the ConfigBean. AdminEventListerners will be triggered on each server instance. The ConfigAdhoc realization also implements the AdminEventListener and listens for ElementChangeEvents specific for the elements that concern the sip-container. This ConfigAdhoc realization will interact with the ConfigFactory or the Config to make sure that the changes are taken into effect.
The ConfigAdapter realizes the ConfigStartup and ConfigAdhoc tasks.

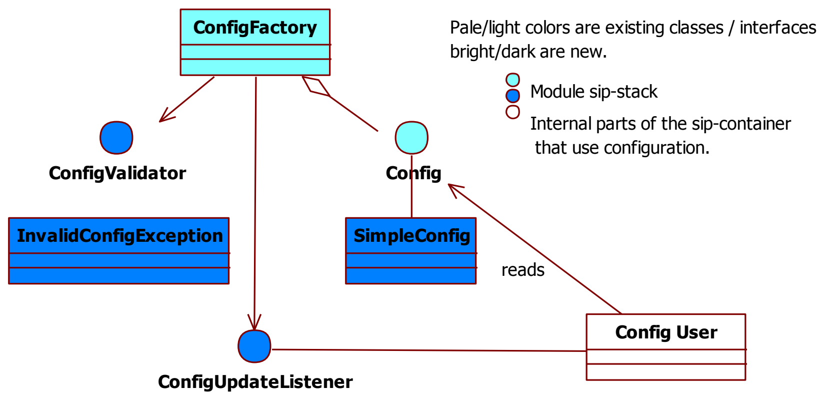

Figure 5: Class diagram depicting the details refactoring details of the configuration mechanism in the sip-container.

Figure 5 shows the participants that take part in such an interaction. The ConfigFactory supplies the Config to Config Users. To allow Config updates in a controlled manner. Each request will be screened by a ConfigurationValidator implementation. In the case that the configuration item to update proofs to be invalid an InvalidConfigException is thrown. Minimally a SEVERE message is logged, but the intention is to allow other means of reporting to be realized as well, such as raising an Alarm by interaction with a FaultManager. Note that this Configuration Validation is to be considered in a latter stage. The initial implementation will not validate yet.

The ConfigFactory keeps a Config which is the embodiment of a Configuration store. Currently this Config is implemented using the java Preferences. A new ‘simple’ implementation will be provided that just keeps track of the configuration items in memory. The eventual realization may very well consider interacting with the ConfigBeans as the ‘storage’, but not necessarily.

In plain cases of configuration usage by a Config User the Config is read on a need basis and the latest update of the ConfigBeans is used by the Config User. This works well for the cases where a config setting is used i.e. when a new session is created. However there are cases when external interactions can’t trigger reading latest updates. For these cases the Config User implements an ConfigChangeListener and registers it at the ConfigFactory / Config releazation. The ConfigFactory / Config will make sure that the Config User is notified of the changes so it can take the necessary measures to dynamically reconfigure it self.

TODO More on what the refactoring of the sip-stack module will imply.

Refactory of the sip-stack module considering Config will imply that the RuntimeConfig will be remove and most of the mbean usage aswell. All direct usages of Preferences will be removed and System properties won't be allowed any more except for temporary optional settings, in frequent used configuration setting (such as the 503disabled) are to be realized using sip-container.properties.

2.5.2.2 Sip stack extensions

General
Using sip-container.properties
Following the following naming convention:

- `sip.extention.<plugin_name>.<plugin_parameter>`

load balancer
Suggested plugin name for the load balancer is `load-balancer` (alternative could be `user-centric`).

- `sip.extention.load-balancer.external.port (example)`

Note that the default load balancer most likely will come with an own domain xml element and will not make use of sip-container.properties.

overload protection
Suggested plugin name is `overload-protection`. Here is the list of all properties applicable for the overload-protection plugin.

- `sip.extention.overload-protection.HttpThreshold`
- `sip.extention.overload-protection.ItThreshold`
- `sip.extention.overload-protection.MmThreshold`
- `sip.extention.overload-protection.NumberOfSamples`
- `sip.extention.overload-protection.OverLoadRegulation`
- `sip.extention.overload-protection.SampleRate`
- `sip.extention.overload-protection.SrThreshold`

2.5.2.3 dns-agent

dns-java specified in OFS.

2.6 Application Verifier

Editor note: planning steps.
-structure of .sar-files
-parsing sip.xml
-glassfish\avk\src\tools\com\sun\enterprise\tools\verifier
-depend on jsr289
-clone tests for web-applications, remove and modify

3 Performance

<How do you want performance team to measure this sub-system? Any micro benchmarks necessary?Any goals? Anticipated scalability limits or goals?>

4 Management

4.1 Interfaces

4.1.1 Exported Interfaces

Interface	Proposed Stability Classification	Specified in	Comments
Sip-container extension	EVOLVING	This document	Extension mechanism for the Sip-container. Allows layers to be added to the sip stack, by configuring plugin jars.

4.1.2 Deployables

The *get/set* commands of *asadmin* can be used to manipulate the following SIP parameters to facilitate the deployment of the listed deployables.

Sip stack extensions
A sip stack extension can be deployed using the get/set commands for the element attributes indicated.

Attribute	Value	Definition	Default
server.java-config.classpath-prefix or server.java-config.classpath-suffix	`Path to jar of sip-container extension`	The sip container can be extended by adding the extension classes to the classpath. Note that the extension needs to be declared as part of the sip-stack as defined by the LayerConfigProvider. (using the default LayerConfigProvider one needs to override the default dispatcher.xml file with one that contains the extension. See sip-container extension configuration properties.)	`(optional)`

4.1.3 Configurables

The *get/set* commands of *asadmin* can be used to manipulate the following SIP parameters.

Sip stack extensions
Sip stack extension configuration properties can be set using the get/set commands for the element indicated.

Element	Property	Definition	Default
server.sip-container.properties	`sip.extention.layer.config.provider`	Property indicating a class to override the default LayerConfigProvider.	`com.ericsson.ssa.config.DispatcherXMLParser`
	`sip.extention.default.layer.config.provider.dispatcher.xml.location`	Property indicating a resource location to override the default dispatcher.xml file. Only applicable with the default LayerConfigProvider.	`dispatcher.xml`

	`sip.extention.<plugin_name>.<plugin_parameter>`	Free purpose properties. Plugin defines these. Refer to plugin specification for specific property definitions.	`(Depends on plugin)`

5 Packaging, Files, and Location

<Does this feature add new jar files or extend existing ones? Where are they located?>

6 Quality

For the overall Quality approach see OFS.
Positive test:
Run FT test case with changed but correct configuration.

Negative test:
Run FT test case with changed but incorrect configuration.

7 Documentation Requirements

See OFS.

8 Open Issues