Functional Specification for Deployment

Author(s): prasad.subramanian@sun.com

Version	*Comments*	*Date*
0.5	Initial Draft	07/21/2007
0.6	Feedback from Sreeram.duvur@sun.com	08/06/2007
0.7	Added specification for EAR with embedded .sar deployment	08/19/2007

References

Extension Deployment SPI	Proposal by hong.zhang@sun.com: http://jse.east/~hzhang/deployment/CodeReview/extension_module/Proposal_for_plugging_in_extension_module_to_glassfish.html
JSR 289	http://www.jcp.org/en/jsr/detail?id=289

1 Introduction

Project Sailfin introduces the need to deploy standalone SIP Applications and SIP Applications embbeded in an Enterprise Application. In addition JSR289 introduces annotations specfic to the SIP Servlet API. The deployment and annotation processing module would need to support deployment of the new “types” of modules and processing of the annotations introduced by JSR289.

1.1 Features

1.1.1 Deployment

Deployment of Standalone SIP Applications , Converged HTTP and SIP Applications and Enterprise Applications containing a standalone SIP Application or Converged HTTP and SIP application or both. A mechanism is provided to plug in the code for the processing the SIP Application module (either standalone or embedded in an ear) to the existing deployment code, and hereby avoiding incisive code changes. This mechanism would also re-use code for processing the Web Components in a converged HTTP and SIP Application.

1.1.2 Support for Runtime Deployment Descriptors

This specification defines the Runtime Deployment Descriptor for SIP Applications , which a deployer could use to configure an application.

1.1.3 Application Upgrade

This specification would also specify the deployment changes required to support hot application upgrades on a per instance basis.

1.1.4 Annotation Processing

Provide a mechanism to process annotations specified by JSR 289 specification. This mechanism is also pluggable , and custom annotation handlers and annotation processors would be plugged in the existing annotation processing framework.

The existing mechanism of processing Java EE annotations is used to process annotations within SIP Servlets being deployed.

1.1.5 Verification

Support for Application verification for Java EE modules is provided. Support for verification of SIP Applications , based on JSR289 would also be provided and is defined in another specification.

1.2 Features not covered

1.3 Terminology

This section introduces some concepts and terms used in the specification.

SIP Application
A module equivalent of a Web Application , except that it has only SIP Servlets. The deployment descriptor is sip.xml . The archive layout is the same as that of the Web Archive (WAR). The archive can have extension of either .war or .sar

Converged SIP and HTTP Application
A combined Web and SIP Application, it has both SIP Servlets and all allowed components of a Web Application. The deployment descriptors in this case are sip.xml and web.xml. The archive layout is the same as that of a Web Archive. The archive can have extension of either .war or .sar
Converged Java EE and SIP Application
An Enterprise Application with one or more Java EE components and a SIP Application. The SIP Application is represented as war in the application.xml. The archive has an extension of .ear .

2 Design Overview

2.1 Basic Design Principle

2.1.1 Binary Overlay

A major consideration for design is to enable binary overlay of the SIP Application deployment code on GlassFish. What this translates to is that all the code that supports deployment of SIP Applications would be plugged into the GlassFish deployment backend, without changing any of the GlassFish code and hence GlassFish code would have no dependency on the SIP Application deployment code.

2.1.2 Deployment Artifacts

The following set of classes constitute the main artifacts that play a role in the deployment of any module
Archivist : This class represents the archive , and provide ability to read the deployment descriptors.
Deployer : The class that handles the exploding of the archive, reading the deployment descriptors, processing of annotations, creation of an object that represents the module/application descriptor.

ModulesManager : This class is responsible for managing the lifecycle of a module for loading and unloading
Loader : This class is responsible for loading or unloading the module to the target container.
ModuleManager : This class represents the module being deployed and holds meta data for the module being deployed / undeployed.

2.1.3 Extension Modules

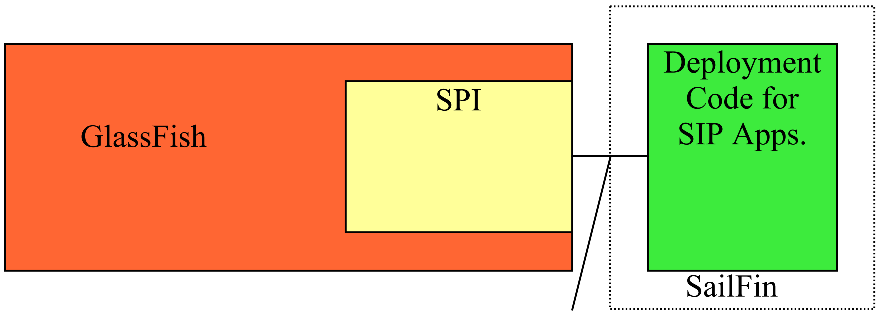
The concept of Extension Module has been introduced in GlassFish V2. All modules that do not fit under a standard Java EE type or as a LifeCycle or Mbeans module are termed as extension modules. Each of the extension modules have their own “type”. The term extension module is an unbrella term to cover all such modules and unlike a Web Module does not indicate the type of the module. All extension modules are represented by an extension-module element in the domain.xml

2.1.4 Extension Deployment SPI

An SPI has been proposed in GlassFish , to provide a mechanism to write pluggable, custom deployment code , to deploy/undeploy extension modules. This SPI abstracts the basic artifacts that are needed to explode an archive, process the deployment descriptors, create a descriptor object and load to the target container.
The SPI consists of two parts
a. interfaces for custom deployer /archivist, bundle descriptor and loader to be implemented for processing the custom module (in this case SIP Applications)
b. Additional classes in the GlassFish deployment backend that would handle the extension modules. These classes are the equivalent of a Archivist, Deployer, Manager and Loader for extension modules as specfied in Section 2.1.3

Implementation of the SPI
The deployment functionality for SIP Application, Converged SIP Applications is provided by the implementation of the interfaces provided in the SPI. The implementation encapsulate all the deployment processing needed for processing SIP Application and delegate to the processing of the Web Components in these applications to the Web Module deployment code.

The figure below illustrates a schematic of the SPI in GlassFish and how it used in SailFin.



Implementation of SPI classes, plugged in

2.2 Design of the SPI

The SPI has been proposed by Hong Zhang (hong.zhang@sun.com) .As described earlier the SPI has two parts , interfaces and deployment classes to handle extension module.

2.2.1 Interfaces

The SPI exposes three main interfaces

com.sun.enterprise.deployment.interfaces.pluggable.ArchiveDeployer
This provides an interface that combines both the Archivist and Deployer classes in the GlassFish. The implementation of this interface is responsible for
a. expanding the archive if needed
b. reading the deployment descriptors
c. cleaning up the deployment descriptors after unload
d. reading deployment descriptors during the AS startup

com.sun.enterprise.deployment.interfaces.pluggable.ArchiveLoader
This interfaces provides a mechanism for the user to load/unload the module to the target container. The implementor of this interface is expected to know about the interfaces that the target container exposed for loading/unloading. The implemetor may choose not to use this interfaces to implement the class that loads/unloads the module and may go with a custom loader specific to the target container.

com.sun.enterprise.deployment.interfaces.pluggable.ArchiveDescriptor
This interface represents the object that would hold the meta data from the Deployment Descriptor of the module being deployed.

2.2.2 Deployment Classes for extension module

In addition to these interfaces there are new classes introduced in GlassFish deployment code which would enable plugging in of the custom deployment classes during runtime and invoking them when a module classified as an extensions module is deployed.

com.sun.enterprise.deployment.pluggable.PluggableDeploymentInfo
This class is the registry for all module types that can be deployed and their corresponding Deployer classes.
It associates an implementation of the *ArchiveDeployer* interface with the module type that can be deployed. It also creates an instance of an ExtensionModuleArchivist and ExtensionModuleDeployer for this module type.
In addition it would also optionally register a custom loader which would be an implementation of the *ArchiveLoader*, for loading and unloading of modules. In case the developer wishes to use a loading mechanism specific to the target container, then the *ArchiveLoader* need not be registered and the *ExtensionModuleDeployer* provides a method to register a custom loader (which need not implement the *ArchiveLoader*) specific to the container.

This class also creates a custom *AnnotationProcessor* based on the moduleType, and registers custom AnnotationHandler classes for processing annotations specific to the extension module type.

The registration happens when the target container starts up. The target container is responsible for registering the customer Deployer/Loader/Descriptor classes for any extension modules that it accepts.

com.sun.enterprise.deployment.archivist.ExtensionModuleArchivist
This class is like a universal Archivist class for all extension modules being deployed. Based on the module type of the module being deployed , it delegates the function of its methods to the implementation of the *ArchiveDeployer*. For example the if the module being deployed is of type FOO then the *ExtensionModuleArchivst* would delegate the operation of its main methods the implementation of the *ArchiveDeployer* for FOO (e.g. FOOArchiveDeployer). The *ExtensionModuleArchivist* delgates to the *expand()* and *handles()* method of the *ArchiveDeployer*

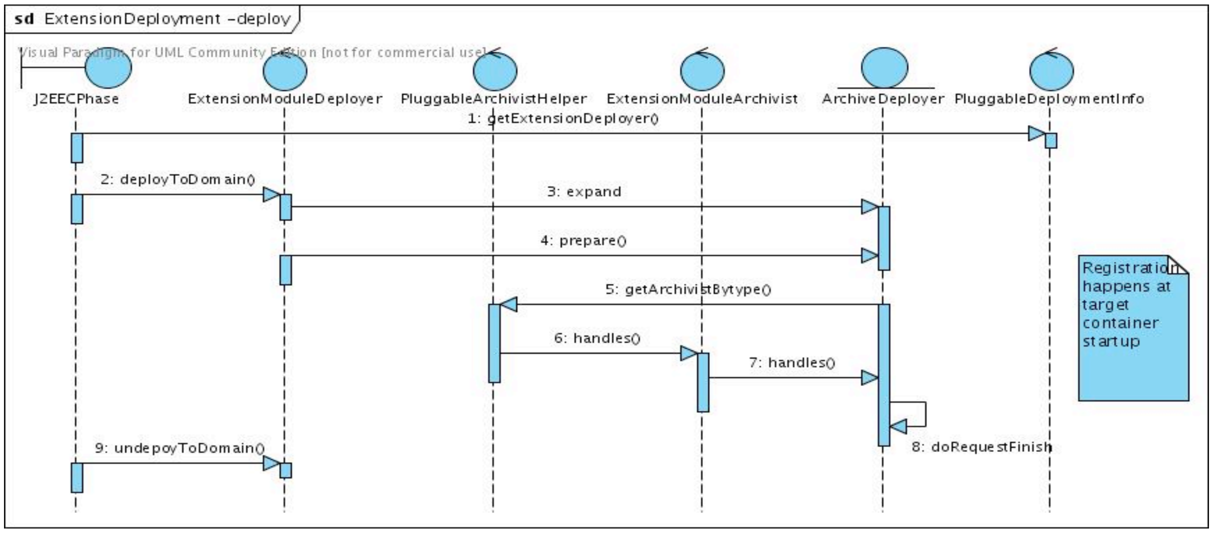
com.sun.enterprise.deployment.backend.ExtensionModuleDeployer
This class is like a universal *Deployer* class for all extension modules being deployed. Based on the module type of the module being deployed , it delegates the function of its methods to the implementation of the *ArchiveDeployer*. For example the if the module being deployed is of type FOO then the *ExtensionModuleDeployer* would delegate the operation of its main methods the implementation of the ArchiveDeployer for FOO (e.g. FOOArchiveDeployer). The *ExtensionModuleDeployer* delgates to the *prepare()* and *cleanup()* method of the *ArchiveDeployer*.
The *ExtensionModuleDeployer* also registers a custom ModulesManager by delegating to the *ArchiveDeployer*. If implemented it registers a custom manager that is specific to the target container.

com.sun.enterprise.instance.ExtensionModuleConfigManager
This class is a universal ModulesManager class for all extension modules being deployed. The *ExtensionModulesConfigManager* also delegates to the implementation *ArchiveDeployer* where needed .

com.sun.enterprise.server.ExtensionModuleManager
This class is the EventListener for Deploy Events. For each extension module type if there is a custom loader (implementing *ArchiveLoader*) defined (in *ExtensionModuleDeployer*) , the *ExtensionModuleManager* processes that event and delegates to the *ExtensionModuleLoader* for loading or unloading operations. Note that this class has no role to play if there is no loader (implementing *ArchiveLoader*) registered with the *ExtensionModuleDeployer*.

com.sun.enterprise.server.ExtensionModuleLoader
This is the wrapper Loader class for Extension Modules, which delegates the loading operations to the *ArchiveLoader* implementation. Similar to the *ExtensionModulesManager* this class has no role to play if there is no custom loader (implementing *ArchiveLoader*) defined for a module type.

The following sequence diagram explains how the implementation of the ArchiveDeployer plugs into the GlassFish runtime.

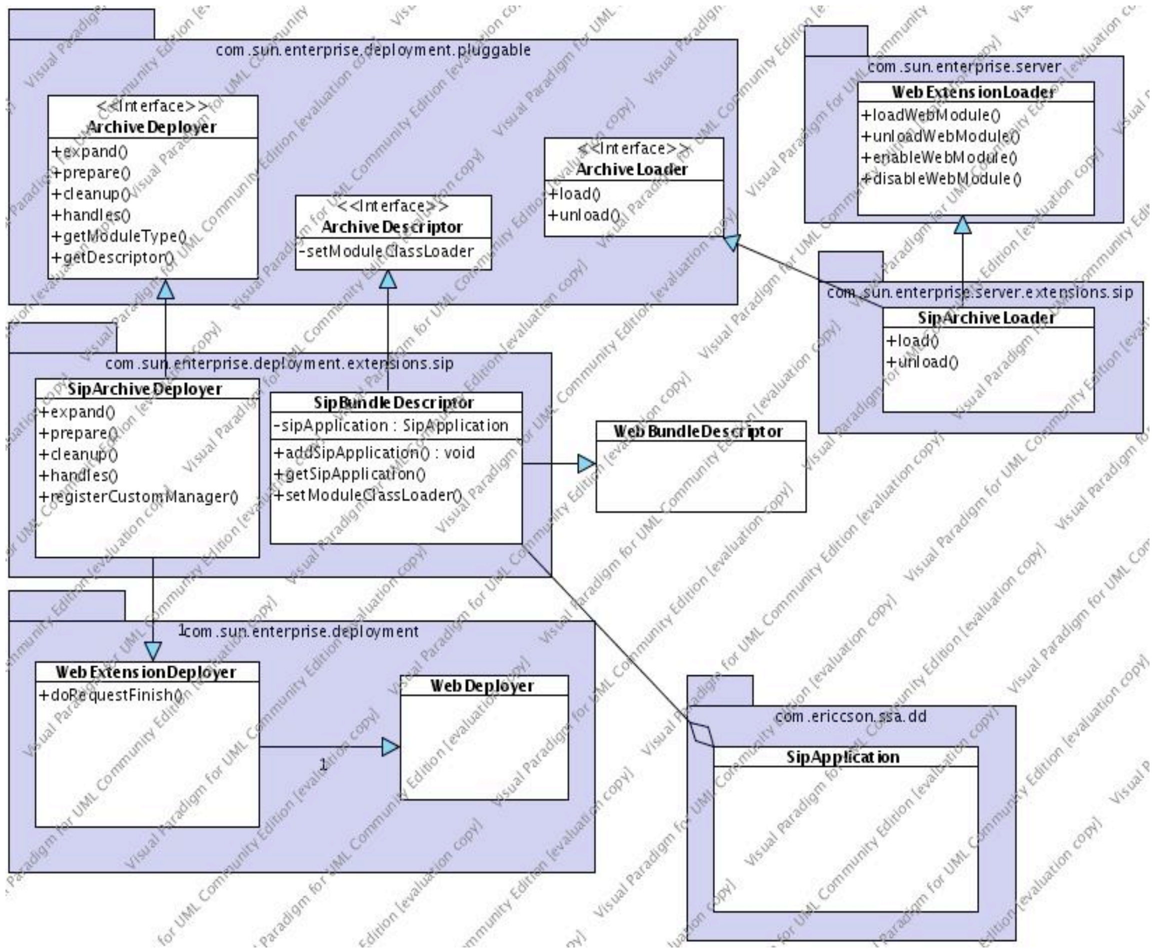


2.3 Implementation of the SPI

2.3.1 Deployment Backend Changes

An implementation of the *ArchiveDeployer* and *ArchiveDescriptor* is provided to handle the deployment of SIP Applications/Converged SIP Applications. The processing of the the HTTP components of a converged SIP Applications is delegated to the Web Deployment component. The implementation of SPI spans deployment and Web Container

The main artifacts of this implementation are in this class diagram.



Module Type
As required by the SPI the module type for the SIP and Converged HTTP+SIP Applications is the FQ class name of the deployer *com.sun.enterprise.deployment.backend.extensions.sip.SipArchiveDeployer*

Bundle Descriptor
com.sun.enterprise.deployment.backend.extensions.sip.SipBundleDescriptor extends *WebBundleDescriptor* and implements *ArchiveDeployer*. The sip.xml meta data is represented by the *com.ericsson.ssa.dd.SipApplication* class which is a instance variable in *SipBundleDescriptor*. In case there is no web.xml in an SIP Application, then the *SipBundleDescriptor* would have the *SipApplication* object and the contents of the default-web.xml.

Deployer
com.sun.enterprise.deployment.backend.extensions.sip.SipArchiveDeployer which implements *ArchiveDeployer.*
This class acts as both an Archivist and Deployer for SIP and Converged HTTP+SIP Applications.

The *handles()* method determines if the archive being deployed can be handled by this class.

The *expand()* method explodes the archive.

The *prepare()* method parses the sip.xml and creates a *SipApplication* object. To parse the descriptors (web+ persistence) , this class delegates to a wrapper class called the *WebExtensionDeployer*. This delegate will also process standard Java EE annotations , The *prepare()* method takes in a argument to determine if this a deployment phase or a application server startup phase.

The *prepare()* method would also read the deployment descriptors form the generated/xml folders during application server startup.
Processing of custom annotations would be done in the prepare() method during the deployment phase.

com.sun.enterprise.deployment.WebExtensionDeployer
This is an extension of the *WebDeployer* class , and it exposes wrapper methods that call only those methods of the *WebDeployer* that would be needed to process the Web components of an extension module being deployed.

DeployEventListener (ModulesManager)
com.sun.enterprise.server.ExtensionModuleManager
The ExtensionModuleManager is used as the modules manager for SIP Applications. This class creates a ExtensionModuleLoader instance and delegates the loading / unloading of the Application to this class. The ExtensionModuleLoader looks up the ExtensionModuleDeployer and gets the ArchiveLoader implementation for the SIP Application. The ArchiveLoader is described in the next section.

ArchiveLoader
com.sun.enterprise.server.extensions.sip.SipArchiveLoader
This class implements the *com.sun.enterprise.deployment.pluggable.interfaces.ArchiveLoader* interface. Given that the target container for SIP Applications is the Web Container, a new class that encapsulates the loading/unloading functionality of the WebContainer is introduced.

This class *com.sun.enterprise.server.WebExtensionLoader* exposes methods that create the necessary artifacts (using GlassFish implementation classes) for loading and unloading. The *SipArchiveLoader* extends this class and delegates creation of artifacts for loading and unloading and the loading and unloading operation itself to this class.

2.3.2 Web Container Changes

The target container for SIP Applications both converged and standalone , is the Web Container. The SIP Servlet Container is bootstrapped by the Web Container during startup and can be seen as an extension of the WebContainer. The application loading mechanism in the SIP Servlet Container is similar to the Web Container.

Loading an Application in the Web Container during deployment

The following section describes how a Web Application is loaded in the WebContainer.
1. A Web Application is represented by the class *com.sun.enterprise.web.WebModule* .WebModule extends *org.apache.catalina.core.StandardContext.*

The meta data from the deployment descriptors is represented by *com.sun.enterprise.deployment.WebBundleDescriptor.*
The meta data from the WebBundleDescriptor is used to configure the WebModule , by *com.sun.enterprise.web.WebModuleContextConfig*, which implements org.apache.catalina.startup.Lifecycle interface .

2. The loadWebModule(...) method in com.sun.enterprise.web.WebContainer is invoked by com.sun.enterprise.server.WebModuleDeployEventListener.

3. A new instance of WebModule is created , and an instance of the WebModuleContextConfig is also created. The WebModuleContextConfig is populated with the configuration meta data for the Application. This WebModuleContextConfig is then registered as a Lifecycle listener for the WebModule instance.

4. The WebModule is added to the parent , which is a Virtual Server object as a child.

5. If the VirtualServer is running then the WebModule is started. The start event is captured by WebModuleContextConfig and the WebModule is configured.

Loading an Application in the Web Container during server startup

The sequence of loading a Web Application during server startup is different from that of deployment.
1. The *WebModule* beans for all Web Applications deployed to each of the Virtual Servers is read.
2. For each Web Module deployed on the Virtual Server the *WebBundleDescriptor* object is created by reading the deployment descriptors from the "*${cm.sun.aas.instanceRoot}/generated/xml*" folder.
3. Using the *WebBundleDescriptor* obtained, the same sequence as mentioned in the deployment phase is followed.

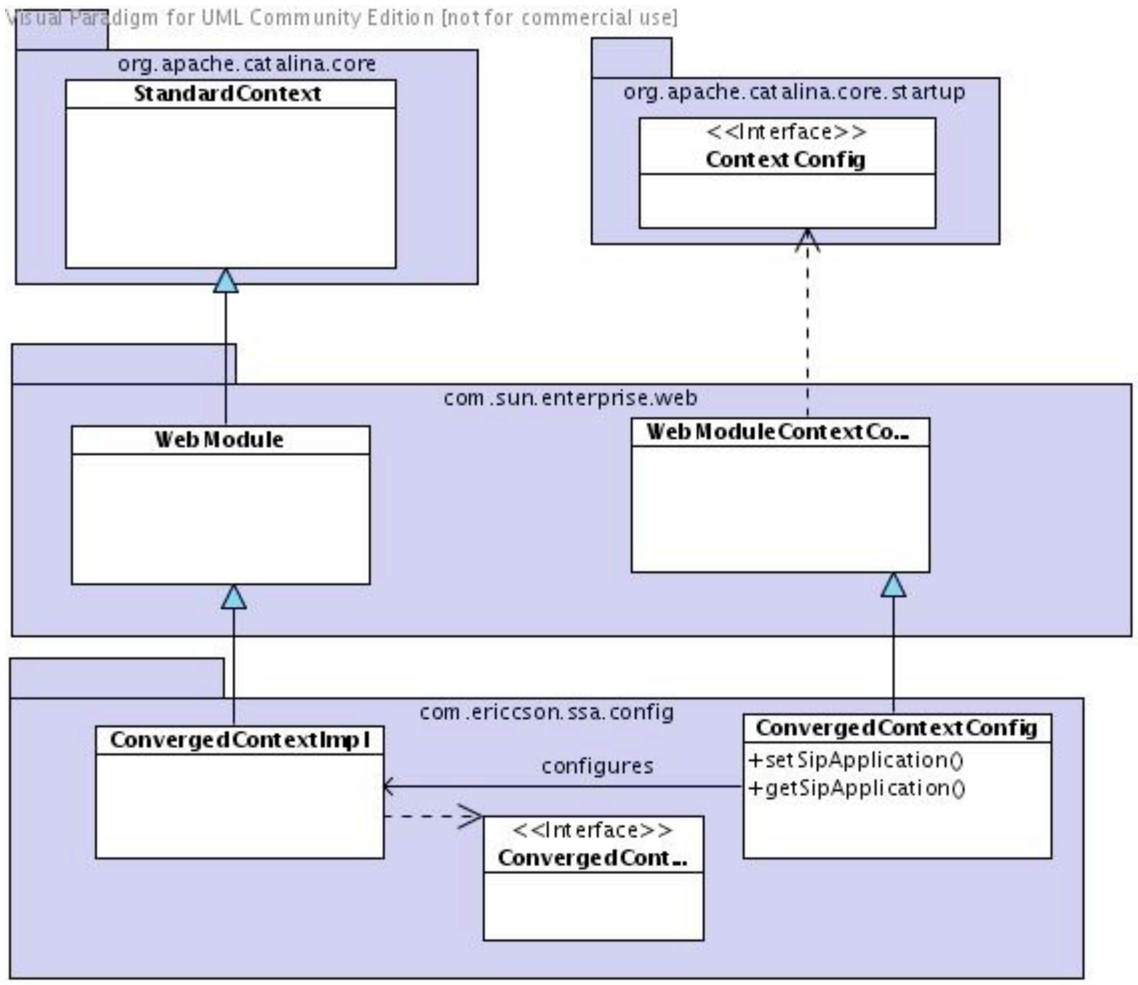
Loading of SIP Applications
The loading of SIP Applications and Converged HTTP and SIP Applications follow the same sequence as mentioned above. The HTTP and SIP components would need share a context.

Two major changes have been proposed to the Web Container in order to support the loading of SIP Applications.
a. Support for extension modules in the Web Container.
b. Custom Context and ContextConfig classes for each

Support for Extensions Modules
The Web Container has been modified to support extension modules defined by the ExtensionModule config beans. The limitation being that the BundleDescriptor of such an extension module should be a sub class of *com.sun.enterprise.deployment.WebBundleDescriptor*

Custom Context and ContextConfig classes
The following classes are introduced to support the loading and configuration of SIP Applications/ Converged SIP Applications.
com.ericcson.ssa.config.ConvergedContextImpl which extends *com.sun.enterprise.web.WebModule* . This class represents the SIP Application being deployed.

com.ericcson.ssa.config.ConvergedContextConfig which extends *com.sun.enterprise.web.WebModuleContextConfig . T*his class represents the Lifecycle listener for configuration of the *ConvergedContext.*

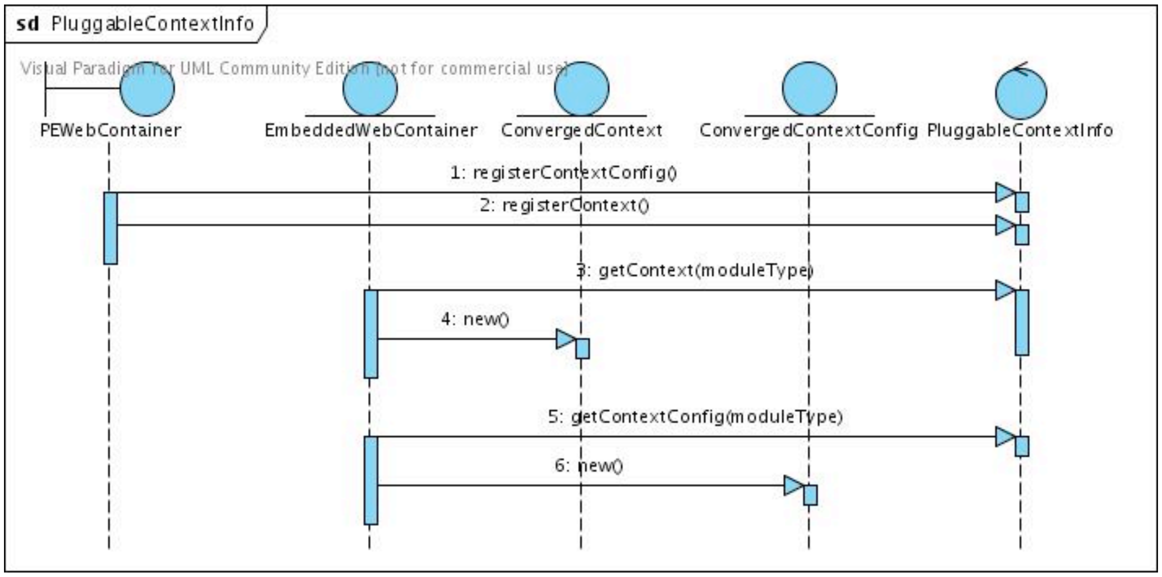


The following class diagram explains these classes.

Pluggable Context SPI

An SPI has been introduced in the *WebContainer* to enable developers to plug in custom *Context* and *ContextConfig* implementations, to support various extension modules. A registry class *com.sun.enterprise.web.PluggableContextInfo* , maps the Context and ContextConfig implementation for each module type.
The following sequence is followed to plug in the custom Context and ContextConfig classes.
1. During the startup of the WebContainer, the custom *Context* and *ContextConfig* implementation classes are registered, with *PluggableContextInfo*, with the moduleType as a the key.
2. During application loading (both at startup and deployment) the appropriate Context and ContextConfig classes for that module Type are looked up from the PluggableContextInfo. This lookup happens in *com.sun.enterprise.web.WebContainer .*

The following sequence diagram explains the sequence of registration and lookup of Context and ContextConfig classes.



2.4 Deployment of Converged Java EE and SIP Applications

The deployment of a Pure SIP / Converged HTTP and SIP application embedded within a .ear (Enterprise Application), follows the same path as any Enterprise Archive. The following limitations are imposed to extend the SPI to work for an embedded extension application.
1. Only modules who Deployment Descriptor object extend the com.sun.enterprise.deployment.BundleDescriptor can be deployed.
2. Only modules that extend the web module and are specified as a web module in the application.xml can be deployed.

The followingclasses in the GlassFish backend are used :
Archivist : *com.sun.enterprise.deployment.archivist.ApplicationArchivist*
When a web module is being processed by an ApplicationArchvist, the ApplicationArchivist would delegate the processing of the archive to the appropriate ExtensionModuleArchivist or the WebArchivist, as the case may be. In the case of all other modules the appropriate Archivist is invoke to process the module.

Deployer : *com.sun.enterprisedeployment.phasing.AppDeployer*
The AppDeployer is responsible for the creation of the Application for the EAR deployment descriptor and the module descriptors of all the modules within.

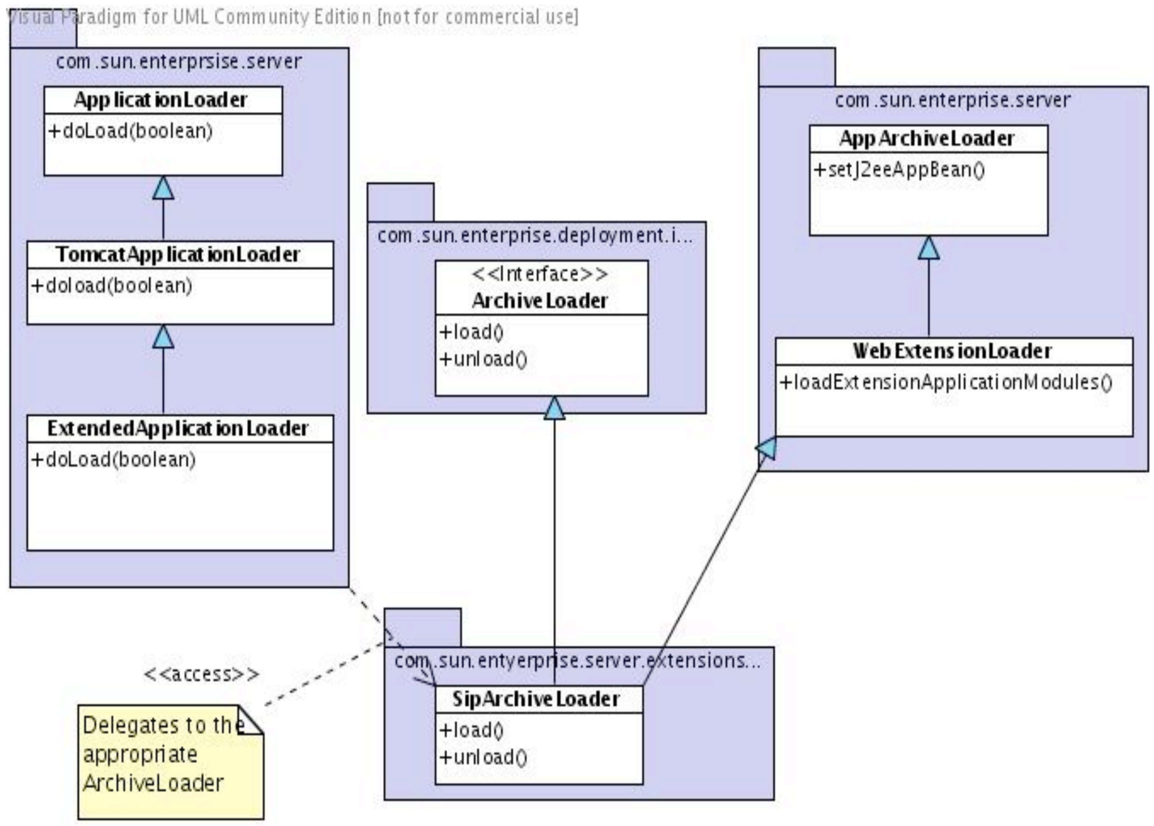
DeploymentDescriptor : *com.sun.enterprise.deployment.Application*
The Applcation object would contain the deployment descriptors for all the ExtensionModules within the EAR. The Application class has been modified to recognize DepoymentDescriptors whose module type correspond to registered Extension Modules

Loader : *com.sun.enterprise.server.ExtendedAppLoader*
The ExtendedAppLoader is the Loader class used for deploying EARs with an embedded Extension module. The ExtendedAppLoader extends the TomcatApplicationLoader which is

the loader used for all the EARs without embedded Extension module. The ExtendedAppLoader would get the appropriate ArchiveLoader for the module type being deployed.

In this case the SipArchiveLoader, handles the loading of the extension module , by delegating to the appropriate method in the Web Container. A helper class AppArchiveLoader has been introduced in the GlassFish deployment backend, to provide support during loading for extension modules embedded within an EAR.

This class diagram below explains the model being followed


Visual Paradigm for UML Community Edition [not for commercial use]
com.sun.enterprsise.server
ApplicationLoader
+doLoad(boolean)
TomcatApplicationLoader
+doload(boolean)
ExtendedApplicationLoader
+doLoad(boolean)
com.sun.enterprise.deployment.i...
<<Interface>>
ArchiveLoader
+load()
+unload()
com.sun.enterprise.server
AppArchiveLoader
+setJ2eeAppBean()
WebExtensionLoader
+loadExtensionApplicationModules()
<<access>>
Delegates to the appropriate ArchiveLoader
com.sun.entyerprise.server.extensions...
SipArchiveLoader
+load()
+unload()

2.5 Annotation Processing

The SPI provides a mechanism for plugging in annotation handlers for processing custom annotation.

2.5.1 Plugging in Custom Annotation Handlers

The PluggableDeploymentInfo would register custom annotation handlers with a AnnotationProcessor object created for a particular module type. The key for registering these annotation handlers is the module type.

2.5.2 Plugging in the Annotation Processor in the deployment code

The standard Java EE annotations would be processed inside the WebDeployer methods (exposed via the WebExtensionDeployer). Once these annotations are processed , the custom annotation processor from the PluggableDeploymentInfo is invoked. A Scanner object is invoked to parse through the archive for any custom annotations and processed via the custom annotation handler registered with the custom annotation processor invoked.

Application Upgrade

The High Availability requirements specify that an applications on instances within a cluster should have the capability to be upgraded without imposing a downtime for the requests being served. It also specifies that each application would need to have a capability to know that an upgrade has happened / is going to happen.

When an application is being upgraded it would receive an event stating that an upgrade is planned. This application would stop allocating resources for new requests and quiesce the existing request.

The usual redeploy mechanism would be used to deploy a newer version of the Application. This application would deployed in a disabled state. A deployment property would be used to indicate that this redeploy action is an Application Upgrade option.

The older application would be disabled and the newer one would be enabled. After the Upgrade (i.e. Enabling of the newer version of the Application) the application would revceive another event stating that the application has been upgraded. These notification events would be generated from the Administration module.

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

<Describe how performance, management status, and diagnostic information is exposed. How does this feature handle dynamic configuration changes?>

4.1 Interfaces

<How is this feature(s) configured by administrator? Does it introduce new commands or modify existing ones? Show syntax of expected administrative commands and response codes. What is the schema for new configuration? Show the DTD snippets. What are their default values? What are the validation rules? List stability level for each of the above [committed|evolving|unstable|standard]. Does it consume interfaces from other projects or sub-systems (imported) or produce interfaces for consumption (exported).

4.1.1 Exported Interfaces

Interface	Defined in	Stability
sun-sip_1-1.dtd	This document	Unstable
com.sun.enterprise.deployment.annotation.AnnotationHandler	This document ?	Unstable

4.1.2 Imported Interfaces

Interface	Defined in	Stability

Interface	Defined in	Stability
sun_domain_1-4.dtd	GlassFish	Stable
com.ericsson.glassfish.sip.startup.SipStartupListener		Project Private

4.1.3 Configuration

Pure SIP Application and Converged HTTP and SIP Application

The configuration and management of the pure SIP Applications and Converged HTTP and SIP Applications would be handled by the extension-module element in the domain.xml. The extension-module element has been defined in sun-domain_1_3.dtd. The following snippet of the DTD shows the extension-module element

```
<!ELEMENT extension-module (description?, property*)>

<!ATTLIST extension-module
   name CDATA #REQUIRED
   location CDATA #REQUIRED
   module-type CDATA #REQUIRED
   object-type %object-type; "user"
   enabled %boolean; "true"
   libraries CDATA #IMPLIED
   availability-enabled %boolean; "false"
   directory-deployed %boolean; "false">
```

The following table lists down the valid scenarios for SIP Applications and Converged HTTP and SIP Applications

Components	Deployment Descriptors	Context Root	Archive extension
SIP Servlets, HTTP Servlets	web.xml, sip.xml, sun-web.xml	As defined by HTTP Servlet Specification	.sar / .war
SIP Servlets	sip.xml, sun-web.xml	Not Applicable	.sar /.war

Table 4.1.2

Converged JavaEE and SIP Application

The configuration of such applications would be done via the j2ee-application element as it is done all other enterprise archives.
A valid converged Java EE and SIP Application consists of any Java EE component and a SIP / Converged SIP Application packages within an Enterprise Archive (.ear). The SIP Application would be specified in the application.xml as a web component.

4.1.3.1 ContextRoot for SIP Applications.

Only converged HTTP and SIP Applications would have a context root defined. The context root would be specified as property named *contextRoot* under the extension-module element

4.1.3.2 Differentiating Converged vs Pure Applications

A property under extension-module named *isConverged* would be used to indicate that an application is converged or not.

5 Packaging, Files, and Location

5.1 Packaging

The following table lists the jar files created by this module, and also jar files in Glassfish affected by this module.

Jar File	Location	New/Updated
comms-appserv-deployment.jar	${com.sun.aas.installRoot}/lib	New
appserv-rt.jar	${com.sun.aas.installRoot}/lib	Updated

In case of native packages like SVR4 packages, sailfin-deployment.jar would be a part of the core package that delivers the SailFin functionality.

6 Quality

CLI

The same CLI commands that are used for deployment of other modules are used for deploying SIP Applications. All the command options need to be tested.
Deploy/Redeploy and Undeploy command should be tested, for success and rollback in case of failure

Application Content

All the valid SIP Applications / Converged HTTP and SIP Application/ Converged Java EE and SIP Applications should be deployed/redployed/undeployed.
Java EE annotations within SIP Servlets should be tested.
JSR289 annotations with SIP Servlets should be tested.

7 Documentation Requirements

All the relevant and valid scenarios for SIP Applications and converged SIP Applications need to be documented.

8 Open Issues

Issue No	Issue	Comment
8.1	The annotation processing extension is yet to be implemented	
8.2	The extension deployment SPI is yet to be implemented for deploying SIP Applications embedded with an .ear	
8.3	There is a comment that the SipModuleDeployEventListener extends a GlassFish internal class and violates the principle of SPI	CLOSED

APPENDIX

A : sun-sip_1-1.dtd

```
<!-- root element for vendor specific sip application (module) configuration -->
<!ELEMENT sun-sip-app (security-role-mapping*, session-config?,
                       ejb-ref*, resource-ref*, resource-env-ref*,
                       service-ref*, message-destination-ref*, class-loader?,
                       property*, message-destination*)>

<!ELEMENT security-role-mapping (role-name, (principal-name | group-name)+)>
<!ELEMENT role-name (#PCDATA)>

<!ELEMENT principal-name (#PCDATA)>
<!ATTLIST principal-name class-name CDATA #IMPLIED>
<!ELEMENT group-name (#PCDATA)>

<!ELEMENT session-config (session-manager?, session-properties?)>

<!ELEMENT session-manager (manager-properties?, store-properties?)>
<!ATTLIST session-manager persistence-type CDATA "memory">

<!ELEMENT manager-properties (property*)>
<!ELEMENT store-properties (property*)>
<!ELEMENT session-properties (property*)>

<!ELEMENT jndi-name (#PCDATA)>

<!ELEMENT resource-env-ref (resource-env-ref-name, jndi-name)>
<!ELEMENT resource-env-ref-name (#PCDATA)>

<!ELEMENT resource-ref (res-ref-name, jndi-name, default-resource-principal?)>
<!ELEMENT res-ref-name (#PCDATA)>

<!ELEMENT default-resource-principal ( name,  password)>

<!--
This node holds information about a logical message destination
-->
<!ELEMENT message-destination (message-destination-name, jndi-name)>

<!--
This node holds the name of a logical message destination
-->
<!ELEMENT message-destination-name (#PCDATA)>

<!--
message-destination-ref is used to directly bind a message destination reference
to the jndi-name of a Queue,Topic, or some other physical destination. It should
only be used when the corresponding message destination reference does not
specify a message-destination-link to a logical message-destination.
-->
<!ELEMENT message-destination-ref (message-destination-ref-name, jndi-name)>

<!--
name of a message-destination reference.
-->
<!ELEMENT message-destination-ref-name (#PCDATA)>

<!--
This text nodes holds a name string.
-->
<!ELEMENT name (#PCDATA)>

<!--
This element holds password text.
```

```
-->
<!ELEMENT password (#PCDATA)>


<!ELEMENT ejb-ref (ejb-ref-name, jndi-name)>
<!ELEMENT ejb-ref-name (#PCDATA)>

<!--
  The class-loader element allows developers to customize the behaviour
  of their web application's classloader.

  attributes
    extra-class-path          List of comma-separated JAR files to be
                              added to the web application's class path
    delegate                  If set to false, the delegation behavior
                              of the web application's classloader complies
                              with the Servlet 2.3 specification,
                              section 9.7.2, and gives preference to classes
                              and resources within the WAR file or the
                              extra-class-path over those higher up in the
                              classloader hierarchy, unless those classes or
                              resources are part of the Java EE platform.
                              If set to its default value of true, classes
                              and resources residing in container-wide library
                              JAR files are loaded in preference to classes
                              and resources packaged within the WAR file or
                              specified on the extra-class-path.
    dynamic-reload-interval   Not supported. Included for backward
                              compatibility with previous Sun Java System
                              Web Server versions
-->
<!ELEMENT class-loader (property*)>
<!ATTLIST class-loader extra-class-path CDATA  #IMPLIED
                       delegate %boolean; 'true'
                       dynamic-reload-interval CDATA #IMPLIED >

<!--
Syntax for supplying properties as name value pairs
-->
<!ELEMENT property (description?)>
<!ATTLIST property name  CDATA  #REQUIRED
                   value CDATA  #REQUIRED>

<!ELEMENT description (#PCDATA)>

<!--
This text nodes holds a value string.
-->
<!ELEMENT value (#PCDATA)>


<!--
                                        W E B   S E R V I C E S
-->
<!--
Runtime settings for a web service reference.  In the simplest case,
there is no runtime information required for a service ref.  Runtime info
is only needed in the following cases :
 * to define the port that should be used to resolve a container-managed port
 * to define default Stub/Call property settings for Stub objects
 * to define the URL of a final WSDL document to be used instead of
the one packaged with a service-ref
-->
<!ELEMENT service-ref ( service-ref-name, port-info*, call-property*, wsdl-override?, service-
impl-class?, service-qname? )>

<!--
Coded name (relative to java:comp/env) for a service-reference
```

```
-->
<!ELEMENT service-ref-name ( #PCDATA )>

<!--
Information for a port within a service-reference.

Either service-endpoint-interface or wsdl-port or both
(service-endpoint-interface and wsdl-port) should be specified.

If both are specified, wsdl-port represents the
port the container should choose for container-managed port selection.

The same wsdl-port value must not appear in
more than one port-info entry within the same service-ref.

If a particular service-endpoint-interface is using container-managed port
selection, it must not appear in more than one port-info entry
within the same service-ref.

The optional message-security-binding element is used to customize the
port to provider binding; either by binding the port to a specific provider
or by providing a definition of the message security requirements to be
enforced by the provider.

-->
<!ELEMENT port-info ( service-endpoint-interface?, wsdl-port?, stub-property*, call-property*,
message-security-binding? )>

<!--
Fully qualified name of service endpoint interface
-->
<!ELEMENT service-endpoint-interface ( #PCDATA )>
<!--
Port used in port-info.
-->
<!ELEMENT wsdl-port ( namespaceURI, localpart )>

<!--
JAXRPC property values that should be set on a stub before it's returned to
to the web service client.  The property names can be any properties supported
by the JAXRPC Stub implementation. See javadoc for javax.xml.rpc.Stub
-->
<!ELEMENT stub-property ( name, value )>

<!--
JAXRPC property values that should be set on a Call object before it's
returned to the web service client.  The property names can be any
properties supported by the JAXRPC Call implementation.  See javadoc
for javax.xml.rpc.Call
-->
<!ELEMENT call-property ( name, value )>

<!--
This is a valid URL pointing to a final WSDL document. It is optional.
If specified, the WSDL document at this URL will be used during
deployment instead of the WSDL document associated with the
service-ref in the standard deployment descriptor.

Examples :

  // available via HTTP
  <wsdl-override>http://localhost:8000/myservice/myport?WSDL</wsdl-override>

  // in a file
  <wsdl-override>file:/home/user1/myfinalwsdl.wsdl</wsdl-override>

-->
```

```
<!ELEMENT wsdl-override ( #PCDATA )>

<!--
Name of generated service implementation class. This is not set by the
deployer. It is derived during deployment.
-->
<!ELEMENT service-impl-class ( #PCDATA )>

<!--
The service-qname element declares the specific WSDL service
element that is being refered to.  It is not set by the deployer.
It is derived during deployment.
-->
<!ELEMENT service-qname (namespaceURI, localpart)>

<!--
The namespaceURI element indicates a URI.
-->
<!ELEMENT namespaceURI (#PCDATA)>

<!--
The message-layer entity is used to define the value of the
auth-layer attribute of message-security-binding elements.

Used in: message-security-binding
-->
<!ENTITY % message-layer    "(SOAP)">

<!--
The message-security-binding element is used to customize the
webservice-endpoint or port to provider binding; either by binding the
webservice-endpoint or port to a specific provider or by providing a
definition of the message security requirements to be enforced by the
provider.

These elements are typically NOT created as a result of the
deployment of an application. They need only be created when the
deployer or system administrator chooses to customize the
webservice-endpoint or port to provider binding.

The optional (repeating) message-security sub-element is used
to accomplish the latter; in which case the specified
message-security requirements override any defined with the
provider.

The auth-layer attribute identifies the message layer at which the
message-security requirements are to be enforced.

The optional provider-id attribute identifies the provider-config
and thus the authentication provider that is to be used to satisfy
the application specific message security requirements. If a value for
the provider-id attribute is not specified, and a default
provider is defined for the message layer, then it is used.
if a value for the provider-id attribute is not specified, and a
default provider is not defined at the layer, the authentication
requirements defined in the message-security-binding are not
enforced.

Default:
Used in: webservice-endpoint, port-info
-->
<!ELEMENT message-security-binding ( message-security* )>
<!ATTLIST message-security-binding
          auth-layer  %message-layer; #REQUIRED
          provider-id CDATA           #IMPLIED >

<!--
The message-security element describes message security requirements
```

```
that pertain to the request and response messages of the containing
endpoint, or port

When contained within a webservice-endpoint this element describes
the message security requirements that pertain to the request and
response messages of the containing endpoint. When contained within a
port-info of a service-ref this element describes the message security
requirements of the port of the referenced service.

The one or more contained message elements define the methods or operations
of the containing application, endpoint, or referenced service to which
the message security requirements apply.

Multiple message-security elements occur within a containing
element when it is necessary to define different message
security requirements for different messages within the encompassing
context. In such circumstances, the peer elements should not overlap
in the messages they pertain to. If there is any overlap in the
identified messages, no message security requirements apply to
the messages for which more than one message-security element apply.

Also, no message security requirements apply to any messages of
the encompassing context that are not identified by a message element.

Default:
Used in: webservice-endpoint, and port-info
-->
<!ELEMENT message-security ( message+, request-protection?, response-protection? )>

<!--
The message element identifies the methods or operations to which
the message security requirements apply.

The identified methods or operations are methods or operations of
the resource identified by the context in which the message-security
element is defined (e.g. the the resource identified by the
service-qname of the containing webservice-endpoint or service-ref).

An empty message element indicates that the security requirements
apply to all the methods or operations of the identified resource.

When operation-name is specified, the security
requirements defined in the containing message-security
element apply to all the operations of the endpoint
with the specified (and potentially overloaded) operation name.

Default:
Used in: message-security
-->
<!ELEMENT message ( java-method? | operation-name? )>

<!--
The java-method element is used to identify a method (or methods
in the case of an overloaded method-name) of the java class
indicated by the context in which the java-method is contained.

Default:
Used in: message
-->
<!ELEMENT java-method ( method-name, method-params? )>

<!--
The operation-name element is used to identify the WSDL name of an
operation of a web service.

Default:
Used in: message
-->
```

```
<!ELEMENT operation-name ( #PCDATA )>

<!--
The request-protection element describes the authentication requirements
that apply to a request.

The auth-source attribute defines a requirement for message layer
sender authentication (e.g. username password) or content authentication
(e.g. digital signature).

The auth-recipient attribute defines a requirement for message
layer authentication of the reciever of a message to its sender (e.g. by
XML encryption).

The before-content attribute value indicates that recipient
authentication (e.g. encryption) is to occur before any
content authentication (e.g. encrypt then sign) with respect
to the target of the containing auth-policy.

An absent request-protection element is the recommended shorthand
for a request-protection element with unspecified values for both the
auth-source and auth-recipient attributes.

Default:
Used in: message-security

 * Expected evolution to support partial message protection:
 *
 * request-protection ( content-auth-policy* )
 *
 * If the request-protection element contains one or more
 * content-auth-policy sub-elements, they define the authentication
 * requirements to be applied to the identified request content. If multiple
 * content-auth-policy sub-elements are defined, a request sender must
 * satisfy the requirements independently, and in the specified order.
 *
 * The content-auth-policy element would be used to associate authentication
 * requirements with the parts of the request or response object identified
 * by the contained method-params or part-name-list sub-elements.
 *
 * The content-auth-policy element would be defined as follows:
 *
 * content-auth-policy ( method-params | part-name-list )
 * ATTLIST content-auth-policy
 *         auth-source (sender | content) #IMPLIED
 *         auth-recipient (before-content | after-content) #IMPLIED
 *
 * The part-name-list and part-name elements would be defined as follows:
 *
 * part-name-list ( part-name* )
 * part-name ( #PCDATA )
 *
-->
<!ELEMENT request-protection EMPTY >
<!ATTLIST request-protection
          auth-source (sender | content) #IMPLIED
          auth-recipient (before-content | after-content) #IMPLIED>

<!--
The response-protection element describes the authentication requirements
that apply to a response.

The auth-source attribute defines a requirement for message layer
sender authentication (e.g. username password) or content authentication
(e.g. digital signature).

The auth-recipient attribute defines a requirement for message
layer authentication of the reciever of a message to its sender (e.g. by
XML encryption).

The before-content attribute value indicates that recipient
authentication (e.g. encryption) is to occur before any
content authentication (e.g. encrypt then sign) with respect
to the target of the containing auth-policy.

An absent response-protection element is the recommended shorthand
for a request-protection element with unspecified values for both the
auth-source and auth-recipient attributes.

Default:
Used in: message-security

 * Expected evolution to support partial message protection:
 *
 * response-protection ( content-auth-policy* )
 *
 * see request-protection element for more details
 *
-->
<!ELEMENT response-protection EMPTY >
<!ATTLIST response-protection
          auth-source (sender | content) #IMPLIED
          auth-recipient (before-content | after-content) #IMPLIED>

<!--
The method-name element contains the name of a service method of a web service
implementation class.

Used in: java-method
-->
<!ELEMENT method-name (#PCDATA)>
<!--
The method-params element contains a list of the fully-qualified Java
type names of the method parameters.

Used in: java-method
-->
<!ELEMENT method-params (method-param*)>

<!--
The method-param element contains the fully-qualified Java type name
of a method parameter.

Used in: method-params
-->
<!ELEMENT method-param (#PCDATA)>
```