AFRL-RI-RS-TR-2009-45
In-House Technical Report
February 2009

PHOENIX ARCHITECTURE

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-45 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
STEVEN D. FARR , Chief Systems & Information Interoperability Branch	JAMES W. CUSACK, Chief Information Systems Division Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
FEB 09	Final	Nov 06 – Jan 09

4. TITLE AND SUBTITLE

PHOENIX ARCHITECTURE

5a. CONTRACT NUMBER
In-House

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Robert Hillman

5d. PROJECT NUMBER
OIM7

5e. TASK NUMBER
IH

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

AFRL/RISE
525 Brooks Rd.
Rome NY 13441-4505

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RISE
525 Brooks Rd.
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-45

12. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW 2009-0469

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This technical report documents the effort, known as Phoenix, that leveraged all of the existing AFRL expertise and knowledge of Information Management (IM), along with colleagues and customers, to define a Service Oriented Architecture (SOA) based IM solution that will have significant meaning now and well into the foreseeable future. The goal was to define an abstract architecture which defines the concepts and interfaces necessary for an effective set of IM service that should be part of any DoD SOA based infrastructure. An important part of the Phoenix development effort was to ensure that the architecture developed aligns with the Air Force and DoD's vision of current and future network-centric operations. The architecture specifications in this document present the results of the requirements analysis and design to achieve information management services in future net-centric environments.

15. SUBJECT TERMS

Information Management, architecture specification, service interfaces

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Robert Hillman
U	U	U	UU	120	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

1.0 Introduction

This document specifies the design of Phoenix Information Management (IM) Services, also referred to as the IM Services. The Phoenix IM Services project is not a development effort in the traditional sense. The goal is to define an abstract concept for an information infrastructure, rather than to build a system. The architecture specifications in this document provide an approach to meeting the needs of information management in future net-centric environments. The architecture specified for the Phoenix IM Services could be used to develop any number of actual implementations.

This document is one of the deliverables for the Phoenix project. The purpose of this document is to present the results of the requirements analysis and design work that has been completed. An iterative, object oriented approach has been used to develop the design. The results are presented here in the form of textual descriptions of the components along with Unified Markup Language (UML) diagrams. UML diagrams describe in detail the actors, actions, interactions, and overall services architecture. The following types of UML diagrams are used to depict the architectural entities: Use Case, Activity, Sequence, and Class.

1.1 Information Management

The definition of information management is "a set of intentional activities to maximize the value of information for achieving the objectives of the enterprise." The primary purpose of information management is to achieve effective information sharing among the many applications and users within an enterprise. In the case of net-centric Command and Control (C2) systems, mission success is tied to application interoperability, i.e., information sharing among edge-user producer applications and edge-user consumer applications.

Three best practices have been identified as crucial to future net-centric C2 systems. These best practices are:

1. Reduce complexity in the edge-user applications by moving it to a shared and supported infrastructure. The infrastructure will provide common necessary functions, such as authentication, authorization, access control, prioritization, and demand-driven information flow. This will free the information provider and consumer applications from having to manage these functions. The infrastructure will provide universal services, such as publish, subscribe and query, that are information-neutral.
2. Increase the ability to control the system by decreasing the number of places that must be modified to implement a change. By moving policy enforcement to the shared infrastructure, changes in policy can be accomplished without changing any of the edge-user applications. Similarly, when the operational environment changes, the infrastructure will be changed to compensate, and the edge-user applications will still function properly.
3. Package information appropriately for dissemination and management. Effective management of information requires that it be characterized sufficiently so it can be interpreted unambiguously. The characterization is called metadata, while the information itself is called the payload. The information infrastructure uses the metadata to know how and where to acquire, store and deliver the payloads.

The goal of the Phoenix project is define such a shared infrastructure that incorporates these best practices and thus allows for both rapid application development and independent evolution of disparate C2 systems.

1.2 Background

The Air Force and the Department of Defense (DoD) have been moving toward a network-centric concept of operations for several years. The advent of Service Oriented Architecture (SOA) based systems has increased the likelihood of actually implementing the conceived overarching Global Information Grid (GIG). SOA-based systems group functionalities around business processes and expose them as packaged, interoperable services. These services allow applications to exchange data as they perform their individual or collaborative business processing. These characteristics of SOAs provide the perfect blend of rigidness and flexibility that is needed for Information Management operations.

The Air Force Research Laboratory's Systems & Information Interoperability Branch (AFRL/RISE) has Information Management at its core. The branch grew out of the pioneering work done in this field by the members of the Joint Battlespace Infosphere (JBI) special project. In turn, that project was kicked off by a review and subsequent publication from the Scientific Advisory Board (SAB) in 1999 stating that the Air Force was weak in the areas of systems integration and information management.

This current effort, known as Project Phoenix, aims to leverage all of the existing in-house knowledge of IM, along with the expertise of RISE's colleagues and customers, to define a SOA-based IM solution that will have significant meaning now and well into the foreseeable future. Part of the Phoenix development effort will be to ensure that it aligns with the Air Force and DoD's vision of current and future network-centric operations.

1.3 Audience

This document is intended for two types of readers: those who are looking for an overview of the key architectural concepts, and those who are developing an implementation of the abstract architecture as specified. For those who are interested in an overview of the key concepts, this document will suffice.

Implementers will need, in addition to this specification, to be familiar with Unified Markup Language (UML), have installed the Visual Paradigm for UML Viewer Edition, and have a copy of the Phoenix.vpp Visual Paradigm project file. Visual Paradigm for UML can be downloaded from http://www.visual-paradigm.com.

1.4 Conventions

This document provides both an architectural specification and a conceptual design for the Phoenix architecture. The architectural specification is a technical specification defined using UML. The conceptual architecture is a less formal description using plain language and diagrams to provide design concepts and objectives.

The term actor is used to denote a component that uses a service. Examples of an actor include a software client or a service, whether it is part of the Phoenix architecture or external to it.

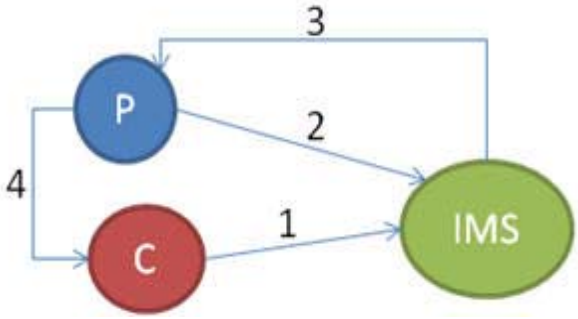
1.4.1 Diagram Conventions

In addition to the UML diagrams, this document uses non-UML diagrams to illustrate high-level concepts. Samples of these diagrams are shown below along with interpretation information.

The diagram below shows a sample communication between Phoenix entities via data channels.

Entity	Meaning	Color
Producer	Produces information.	
Service	Manipulates information.	
Consumer	Consumes information.	
Actor	A generic term that can mean producer, consumer, or service.	
Inquisitor	A type of consumer that queries a service to get information.	

The diagram below is a sample diagram showing labeled information flow.



1. Consumer registers with IMS.
2. Producer requests consumer list.
3. Producer receives consumer list.
4. Producer delivers data directly to consumers.

2.0 Concepts

This section describes the key concepts employed in the Phoenix architecture.

- Information
- Information Types
- Contexts
- Sessions and Session Tracks
- Data Channels
- Data Transports
- Data Transport Factory
- Service Bindings
- Connectors & Stubs
- Filters
- Event Channels
- Control Channels
- Service Orchestration
- Application Service Interaction

2.1 Information

Information is the basic building block for data flowing between and among actors. A complete unit of information consists of:

- An information type identifier
    - Tags the payload with a known and defined information structure
    - May or may not be optional depending on whether or not designers implement the ability to support un-typed information
- Metadata (the description of the payload)
    - May or may not be a subset of the payload
    - May duplicate some or the entire payload
    - Is used for Brokering
    - Is used for Storage and Retrieval
    - May be used by filters
- A payload (the actual data being managed)
    - May consist of known or unknown content
    - May or may not be used by an actor
- An InformationContext (3.1.2.4)
    - Contains attributes that provide additional insights that further describe the information, including any implementation specific attributes

Two simple examples offer different, yet consistent views of information. The first example consists of an information type that contains XML as the payload with certain fields promoted to metadata. The second consists of an information type that contains a binary (image file) payload and XML metadata that describes that image.

Example 1:

In a particular implementation of the Phoenix architecture, an Air Tasking Order (ATO) fragment for a specific reconnaissance mission may be the data of interest. The payload may contain the ATO fragment in XML format, the metadata may contain the mission number and the information type identifier may be mil.af.ato.

Example 2:

For example, in a particular implementation of the Phoenix architecture, a sensor may have captured an image of interest. The information type identifier may be sensor.mil, the payload would be the image itself, and the metadata would describe the image, and possibly, its contents.

Information may be represented in such a way that it contains only pointers to one or both the payload or metadata. Information within the Phoenix architecture is defined by the Information (3.1.2.3) interface.

The Phoenix architecture does not define whether or not instances of information are immutable. This decision is left up to the implementation designers and developers to insure that the implementations of the IM Services are optimally tailored to meet the operational requirements of the stakeholders and their individual Communities of Interest (COIs).

2.2 Information Types

Information types are used to characterize payloads. An information type consists of:

- A unique identifier
- A payload schema
- A metadata schema
- Other implementation specific attributes

The unique identifier for an information type is used by the IM Services to identify what type of information a particular instance represents. This is useful for operations such as metadata or payload validation. Information type identifiers may be implemented such that they represent some kind of implementation specific information type hierarchy. For example, information type identifiers could be implemented to follow Java package standards where child packages are extensions of parent packages.

The payload schema for an information type describes the structure of the information being managed. Each individual payload instance should conform to its corresponding information type's payload schema. For example, an XML payload for type ATO should conform to the payload schema's XML Schema Document (XSD) defined by the ATO information type.

Metadata schemas are similar to the payload schemas. They define the structure of the metadata for the information being managed, i.e., the structure for the metadata describing the payload. Each individual metadata instance should conform to its corresponding information type's metadata schema. For example an XML metadata instance for type ATO should conform to the metadata schema (its XSD) defined by the ATO information type.

It is important to note that structure of the payload and metadata need not be expressed by XML schema documents. XML and XSD were chosen as examples here in an attempt to explain an abstract idea using a common, well-known, and well-understood representation.

Additional attributes for information types may be defined by particular implementations of the architecture. This allows implementation developers to associate data or constructs of any kind with information types. For example, an implementation of the architecture may define special translator classes for each information type. Supporting additional attributes would allow these translator classes to be associated with the information types and stored as part of the information type definition.

Information types are described by the InformationTypeContext (3.2.1.2) interface and stored in a repository managed by the InformationTypeManagementService (3.2.1.3).

2.3 Contexts

A context is a basic construct consisting of a set of key-value pairs that describe the state of an entity defined by the architecture. These entities include data channels, services, actors, etc. Contexts are realized in the architecture using the BaseContext interface (3.1.1.2).

For example, a ServiceContext (3.1.1.8) is used to describe a service. It includes status, name and type attributes.

Contexts are meant to be easily extensible to accommodate the addition of attributes based on operational requirements identified during design of an actual implementation.

2.4 Sessions and Session Tracks

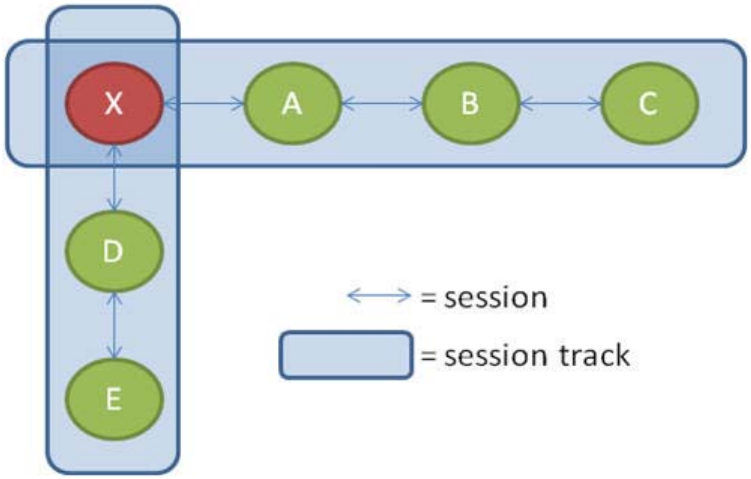
A session is a sustained interaction between two actors.

A session track is an ordered collection of session identifiers showing the sequence of actors involved in a transaction. Session tracks can be used to:

- Show service usage
- Support security and QoS policy decisions
- Determine the originator of a session

It is important to understand the difference between a session and a session track. A session is a single interaction between two actors, whereas a session track is an ordered sequence of session identifiers.

The diagram below shows an actor as the originator of two session tracks.



Sessions and session tracks are defined in the architecture using the SessionContext (3.2.10.2) and SessionTrack (3.1.1.11)interfaces, respectively.

2.5 Channels

Information moves through the architecture using channels. Channels are:

- "Streams" used to read and write data
- High level constructs that do not define wire protocols or data formats

Any actor, either internal or external, must use a channel implementation to move data in order to leverage the full complement of IM capabilities provided by the architecture. However, if an actor wishes to move data without using a Phoenix channel, the channel concept, the interfaces, and the other service constructs do not specifically restrict or prohibit this mode of data transmission.

For example, a web service whose capabilities are being implemented using a Phoenix implementation could be accessed either by the use of Phoenix channels or by direct communication through some exposed web service interface which, in turn, could place the submitted data onto a channel that is read by the implementing Phoenix service.

Channels are realized within the Phoenix architecture by the BaseChannel interface (3.1.3.1). Specific extensions of this interface have been defined for raw byte channels, information channels, and event channels. The control mechanisms for creating and maintaining channel instances are defined by the Transport interface (3.1.3.17).

2.6 Transports

Transports provide life cycle management methods for Channels and encapsulate wire protocols used to move data. Any implementation of the architecture will have a transport defined for each supported wire protocol. Transports are created by the Transport Factory (3.1.3.19).

Transports are realized in the architecture through the Transport interface (3.1.3.17).

2.7 Transport Factory

A transport factory creates instances of Transports based upon some specified criteria. Transport factories are owned and managed by the individual Phoenix services and are used to populate the services' associated Service Bindings.

Transport Factories are realized in the architecture by the TransportFactory interface (3.1.3.19).

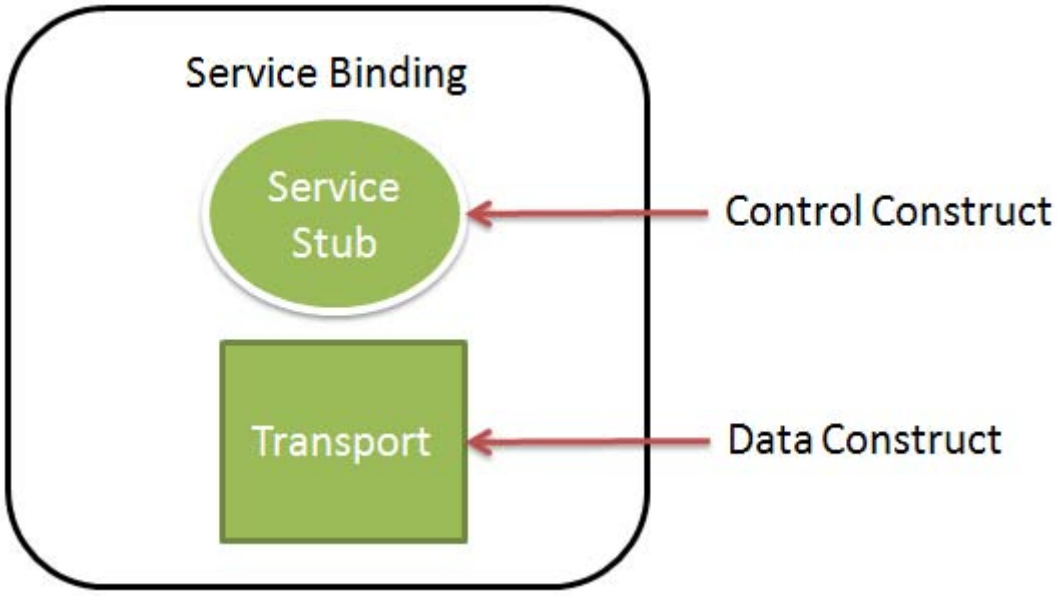
2.8 Service Bindings

A service binding contains the control and data constructs used to interact with the associated service. Each service binding may contain either the stub, the transport, or both of these constructs, depending upon the implemented security policy and their respective permissions.

- Service Stub - This construct, an extension of the BaseServiceStub interface that is specific to the service, i.e., SubmissionServiceStub, represents the control channel connecting the external actor to a Phoenix service. Actors invoke service methods upon these stubs to influence service behavior or request that the service perform an operation.
- Transport - This construct, an instance of the Transport interface, is utilized by the external actor to transmit and receive data to and from a Phoenix service.

Bindings may not be actor specific at implementation time and, as such, their architecture specification supports the notion of them being transferred from one actor to another.

Service Bindings are defined in the architecture by the ServiceBinding interface (3.1.3.16).



2.9 Connectors & Stubs

Services implement and expose their control interfaces via the connector and stub model. A well-known example of this model is the Remote Method Invocation (RMI) model.

The connector represents the service side of a connection between the service and an actor. The service methods that an implementer wishes to expose to an actor are specified within the connector.

The stub represents the actor side of a connection between the service and an actor. Any service method exposed by the connector must be specified by the stub to allow a connection between the actor and the service.

Connectors and stubs are realized in the architecture using the BaseServiceConnector (3.1.1.5) and BaseServiceStub (3.1.1.6) interfaces, respectively.

2.10 Filters

Filters provide a way to manipulate and modify the data moving through the Phoenix architecture. They can be applied to data channels, Information, or any object that works with byte arrays. When a filter is applied to the data entering or leaving a byte array, it applies some operation to the data that may modify it.

Filters can be used for any type of manipulation operation, from compression and decompression, to scrubbing sensitive data.

The Phoenix architecture also defines a very specific child construct of the generic data channel filter called the ResidualProcessFilter (3.1.7.5). The ResidualProcessFilter takes any residual data that has been removed from or modified within the raw data and outputs this residual data on its own data channel. This could be used for things such as splitting data into separate channels for logging filtering operations.

2.11 Event Channels

Event channels are a specific extension of the Channel concept and need to be discussed explicitly because they are used by actors for non-information based communications. Event channels are used to deliver Events to actors. These channels may be negotiated between any two or more actors, or they may be negotiated from an actor to a central Event Notification Service whose sole responsibility is the registration and satisfaction of Event notification requests. Event channels abstract away the mundane details of the Channel and present actors with convenient methods for reading and writing Event messages from and to the channel. Like all other Channels, event channels are created by Transports.

Actors send and receive messages through the EventOutputDataChannel (3.1.6.5) and EventInputDataChannel (3.1.6.4) interfaces, respectively.

2.12 Control Channels

Control channels are abstract concepts that are manifested implicitly by each call upon a service interface. Control channels are not physical constructs within the IMS but instead are embodied by the architecture's use of the connector-stub model for service interface invocations. The connector and stub constructs supply adequate physical components for possible application policies such as security or Quality of Service (QoS).

2.13 Service Orchestration

Service orchestration in the Phoenix abstract architecture has not yet been concretely defined. Some examples exist, such as the necessary chain of services required to provide a complete publish and subscribe workflow, but this chain of services is configured using settings internal to the implementation of the services involved. At this point in the Phoenix development, service orchestration has been left to the implementation designers. These designers must decide what services being implemented can be controlled or tasked by whatever orchestration service or mechanism they define and implement. The architecture's dependence on flexible Contexts to describe state within the services provides implementation designers a measure of flexibility when making these design-time decisions.

Future spirals into the design of the Phoenix abstract architecture may generate a service orchestration service, semantics, and/or other constructs to support this concept.

2.14 Application Service Interaction

Applications are intended to interact with the Phoenix IM Services via Service Bindings. These bindings provide access to one or both of a service's control stub and a transport for creating channels for data transmission and reception that are connected to the corresponding service, depending upon implementation decisions and any applicable runtime policies.

Service Bindings can be made accessible to an application in any number of ways. The two most common that the Phoenix Team repeatedly referred to during the design phase were:

- **Static Access** - The application already has a copy of the required Service Binding within its local address space, perhaps as part of a pre-packaged library.
- **Service Brokering** - The application uses a Service Brokering capability that provides the correct Service Binding as a result of the service brokering process.

To invoke a service method via the control stub, an application retrieves the stub from the Service Binding and invokes the chosen method on the stub. This invocation is funneled through the corresponding service connector and passed on to the service itself, again depending upon implementation decisions and any applicable runtime policies.

To open a Channel between itself and a service, an application retrieves the transport from the service binding and uses that transport to create channels. More detail on this can be found in the Transport, and Channel concept sections.

3.0 Architecture Specification

This section specifies the Phoenix Architecture by means of the interfaces between components. At this level of the design, it is not necessary to define the internal workings of the components. The actual functions, variables and operations will be determined by the implementation designers, whenever an instance of the Phoenix concept is built.

This Phoenix architecture specification is divided into two types of interfaces: component and service. The component and service interfaces have been sub-divided into a total of twenty units by functionality. For example, the component interfaces in the "predicate" group all provide support for predicate definition and processing functions.

For each interface, the specification includes a description of its purpose within the Phoenix architecture. Where appropriate, the specification also includes the object's attributes, its public operations (also called methods), how the interface is used, and a list of related UML diagrams. An operation is defined in this documentation as any action that is performed upon a set of targets by a set of actors.

3.1 Component Interfaces

Component interfaces provide functional pieces that may be used by one or more of the services. The component interfaces have been sub-divided into seven functional groups, ordered by overall importance to the architecture as a whole:

- Core (3.1.1)
- Information (3.1.2)
- Channel (3.1.3)
- Exceptions (3.1.4)
- Predicate (3.1.5)
- Event (3.1.6)
- Filter (3.1.7)

3.1.1 Core

The core group contains the interfaces, contexts, and supporting components that provide the base functionality and meaning behind the Phoenix IM services. The core interfaces are:

- ActivationCallback (3.1.1.1)
- BaseContext (3.1.1.2)
- BasePersistentService (3.1.1.3)
- BaseService (3.1.1.4)
- BaseServiceConnector (3.1.1.5)
- BaseServiceStub (3.1.1.6)
- ConnectorContext (3.1.1.7)
- ServiceContext (3.1.1.8)
- ServiceStatus (3.1.1.9)
- ServiceType (3.1.1.10)
- SessionTrack (3.1.1.11)
- StubContext (3.1.1.12)

3.1.1.1 ActivationCallback

The Activation Callback is intended to provide actors a way to be asynchronously notified when the assigned stub's activation is complete and ready for use. Stubs may not be ready exactly when the actor requests them due to network latencies or other dependencies. This concept was designed explicitly to provide a solution to the service-to- service circular dependency problem.

3.1.1.1.1 Public Operations

`onActivation() : void`

This method is invoked by the assigned stub after stub activation has been completed.

3.1.1.2 BaseContext

This entity is the super class for all Contexts within the architecture. It is primarily an internal data structure that contains key-value pairs for holding data that describes a Context. It also provides several "getter" and "setter" methods for these keys and values.

BaseContext has two required attributes that lay the foundation for all other Context sub-classes: *contextId*, and *attributes*.

1. The *contextId* is the unique identifier for the Context.
2. *attributes* is the Map of attribute names and their associated values.

3.1.1.2.1 Public Operations

`clear() : void`

Clears the attributes map of all entries.

`destroy() : void`

Deletes any references to this object, and clears out any other internal data that the context owns.

`getAllAttributes() : Map<String, Object>`

This method returns a Map that contains the key-value pairs for all the attributes that are stored for this instance of the Context.

`getAttribute(attributeName : String) : Object`

This method returns an Object that represents the value for the given attributeName key name.

Arguments:

- attributeName - The name of the attribute that is to be returned from the Map. This is the key in the Map.

`getAttributes(attributeNames : String[]) : Map<String, Object>`

This method returns a Map that has the keys and values filled in for the given attributeNames key names.

Arguments:

- attributeNames - An array of names (key values) of the attributes to be returned.

`getContextId() : String`

This method returns the unique identifier associated with this instance of specified Context.

`removeAttribute(attributeName : String) : void`

Removes a specific attribute from this Context's attribute Map.

Arguments:

- attributeName - The name of the attribute to be removed.

`removeAttributes(attributeNames : String[]) : void`

Removes the supplied attributes from the attributes Map.

Arguments:

- attributeNames - The names of the attributes to be removed.

`setAttribute(attributeName : String, attributeValue : Object) : void`

This method takes a key-value pair and sets it within the Map maintained by the Context.

Arguments:

- attributeName - The name of the attribute to be added.
- attributeValue - The value for the attribute.

3.1.1.3 BasePersistentService

This interface provides mechanisms for storing and restoring service state in the event of service shutdown and restart.

3.1.1.3.1 Public Operations

`load() : void`

Loads a stored service state from some persistent data store.

`store() : void`

Stores the service state in some persistent data store.

3.1.1.4 BaseService

This interface defines the minimum set of methods required to identify a specific entity as a Phoenix service. It includes a set of generic service maintenance functions that provide a basis for controlling and administering the IM Services at runtime.

3.1.1.4.1 Public Operations

`getModifiableServiceAttributes(sessionTrack : SessionTrack): Map<String, Object>`

Returns the hierarchical view of service attributes that can be modified by external actors. For example, this method could return the Data Channels that the service owns, their settings, and their filters and filter settings.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

`getServiceContext(sessionTrack : SessionTrack) : ServiceContext`

Retrieves the context object that contains the current state of the associated service, as well as its description.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

`getStatus(sessionTrack : SessionTrack) : ServiceStatus`

Retrieves the identifier that signifies the current state of the associated service. The possible values for this identifier are listed in the ServiceStatus enumeration.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

`setServiceAttributes(sessionTrack : SessionTrack, attributeMap : Map<String, Object>) : void`

This method provides a mechanism for influencing service behavior by providing a way to modify service settings at runtime.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- attributeMap - The Map of attribute namess and their corresponding new values.

`start() : void`

This method is used to start a service once it has been implemented and deployed to some platform.

`stop() : void`

This method is used to stop a service after it has already been started.

3.1.1.5 BaseServiceConnector

A base interface defining the minimal methods required to be considered a service connector.

3.1.1.5.1 Public Operations

`activate() : void`

This method is called by the parent service to activate the connector so it is ready for stubs to hit.

`getConnectorContext(sessionTrack : SessionTrack) : ConnectorContext`

This method returns the connector context that is associated with the service.

`updateConnectorContext(sessionTrack : SessionTrack, attributesToUpdate : Map<String, Object%gt;) : void`

This method updates the specified attributes with their new supplied values.

3.1.1.6 BaseServiceStub

Interaction with all services is accomplished using the connector and stub model. A well known example of this model is the Remote Method Invocation (RMI) model. The connector construct represents the service side of a physical connection between the service and an actor. The stub is the actor side of the same physical connection. All methods exposed through the connector are to be implemented within the stub as well. The stub is the physical construct upon which the actor invokes methods. These method invocations are forwarded to the connector by the stub and onward through the connector to the actual service itself. All return values for method invocations follow a reciprocal path through the connector and stub back to the actor.

3.1.1.6.1 Public Operations

`activate(async : boolean) : void`

Activates this stub.

`connect(sessionTrack : SessionTrack, ctx : StubContext) : void`

Connects to the stub. This allows the stub to be ready to be used.

`deactivate() : void`

Deactivates the stub and renders it unusable.

`getStubContext(sessionTrack : SessionTrack) : StubContext`

This method returns the current instance of the StubContext.

`isActivated() : boolean`

This method returns true if its current state is ACTIVATED, and false otherwise.

`registerActivationCallback(callback : ActivationCallback) : void`

This method sets the ActivationCallback for the stub.

`updateStubContext(sessionTrack : SessionTrack, attributesToUpdate : Map<String, Object>) : void`

Updates the specified attributes with the given new values.

3.1.1.7 ConnectorContext

This Context maintains and describes the state of a connector owned by a service. It contains two required attributes: *svcCtx* and *svcRef*.

1. The *svcCtx* is a copy of the Service Context that describes the service associated with the connector.
2. The *svcRef* is a reference or pointer to the actual service instance with which this connector is associated.

3.1.1.7.1 Public Operations

`getService() : BaseService`

This method returns the actual service instance that the connector is holding.

`getServiceContext() : ServiceContext`

This method returns the copy of the service context that describes the serviceto which this connector is connected.

`setService(service : BaseService) : void`

Sets the reference to the associated service for this Context's connector.

Arguments:

- service - The reference to the associated service for this Context's connector.

3.1.1.8 ServiceContext

This Context maintains a description of the service and its capabilities and is also used for storing service state data. The Service Context contains two attributes: *status* and *description*.

1. The *status* is a flag that denotes the current status of this service. The values of this flag are defined by the Service Status enumeration.
2. The *description* is a Map-type attribute that defines additional information about the service. This is the information that is registered with the Service Brokering Service to allow actors to locate services based on some submitted brokering criteria. Minimally the description must contain the name of the service (serviceName), a list of service types that define what service interfaces this service implements (serviceType), and the list of supported Service Bindings (supportedSvcBindings) which provides the collection of service bindings that this service supports. A supported Service Binding is roughly equivalent to a supported protocol, i.e., a service can support a binding for proprietary protocol, IIOP, RMI, etc. The serviceType values are defined in the Service Type enumeration.

3.1.1.8.1 Public Operations

`getFunctionalDescription() : Map<String, Object>`

Retrieves the functional description of the service. This description may include such things as what operations the service supports, and what information types it may or can perform operations upon, among others. The exact contents of this description are left up to the implementation designers to determine.

`getOperationalDescription() : Map<String, Object>`

Retrieves the operational description of the service. This description may contain things like what types of information the service is currently operating upon and/or some performance metrics, among others. The exact contents of this description are left up to the implementation designers to determine.

`getServiceStatus() : ServiceStatus`

Retrieves the identifier for the current status of the described service. Possible values for this identifier are defined in the ServiceStatus enumeration.

`getServiceType() : ServiceType[]`

Retrieves the listing of identifiers that signal what types of functionality the described service supports. Possible values are defined in the ServiceType enumeration.

`getServiceName() : String`

Retrieves the human readable name for the associated service.

`setFunctionalDescription(desc : Map<String, Object>) : void`

Sets the functional description for the service that this context describes.

Arguments:

- desc - The functional description of the service that this context describes.

`setOperationalDescription(desc : Map<String, Object>) : void`

Sets the operational description of the service that this context describes.

Arguments:

- desc - The operational description of the service that this context describes.

`setServiceStatus(status : ServiceStatus) : void`

Sets the status flag for the service that this context describes.

Arguments:

- status - The current status of the service that this context describes.

`setServiceType(type : ServiceType[]) : void`

Sets the array stating what service interfaces the service that this context describes implements.

Arguments:

- type - The array of identifiers for the service interfaces that the service that this context describes implements. Possible values are listed in the ServiceType enumeration.

`setServiceName(serviceName : String) : void`

Sets the human-readable name for the service that this context describes.

Arguments:

- serviceName - The human-readable name for the service that this context describes.

3.1.1.9 ServiceStatus

This enumeration contains the possible states that a Phoenix service can be in at any given time.

3.1.1.9.1 Public Fields

`AVAILABLE`

The service has been started and is ready for use by other actors.

`STARTED`

The service has been started and is finalizing its internal state to make itself ready for use by other actors.

`STARTING`

The service is currently in the process of initializing its internal components.

`STOPPED`

The service has been stopped. The service may be restarted.

`STOPPING`

The service is currently in the process of shutting down its internal components and is no longer able to be used by other actors.

`UNAVAILABLE`

The service has entered an error state that has made it unavailable for use by other actors.

3.1.1.10 ServiceType

This enumeration contains the possible types of Phoenix services that have been defined by the architecture.

3.1.1.10.1 Public Fields

`AUTHORIZATION`

Identifies a service that implements the Authorization Service interface.

`DISSEMINATION`

Identifies a service that implements the Dissemination Service interface.

`EVENT_NOTIFICATION`

Identifies a service that implements the Event Notification Service interface.

`INFORMATION_BROKER`

Identifies a service that implements the Information Brokering Service interface.

`INFORMATION_TYPE_MANAGEMENT`

Identifies a service that implements the Information Type Management Service interface.

`QUERY`

Identifies a service that implements the Query Service interface.

`REPOSITORY`

Identifies a service that implements the Repository Service interface.

`SERVICE_BROKER`

Identifies a service that implements the Service Brokering Service interface.

`SESSION_MANAGEMENT`

Identifies a service that implements the Session Management Service interface.

`SUBMISSION`

Identifies a service that implements the Submission Service interface.

`SUBSCRIPTION`

Identifies a service that implements the Subscription Service interface.

3.1.1.11 SessionTrack

This object contains session identifiers used to track the usage of services and, potentially, to make policy decisions regarding how services may be used by actors.

A SessionTrack instance contains a listing of session identifiers for all actors who have been part of the associated service method invocation chain. Each member of the invocation chain should stamp the SessionTrack instance with their own session identifier before either performing any associated actions or passing along the invocation to another actor.

3.1.1.11.1 Public Operations

`addSessionId(sessionId : String) : void`

Adds the session identifier of the current invocator to the method invocation pedigree list.

Arguments:

- sessionId - The session identifier to add to the pedigree list for this associated method invocation chain.

`getOriginatingSessionId() : String`

Returns the originating actor's session identifier. This is the actor who began the associated method invocation chain.

`getCurrentSessionId() : String`

Returns the session identifier for the actor who last invoked this method.

`getSessionPedigreeList() : String[]`

Returns the complete listing of all session identifiers for all actors who have invoked the associated service method as part of the current method invocation chain.

3.1.1.12 StubContext

Maintains and describes the state of a stub owned by an actor. The Stub Context contains a copy of the ServiceContext, with the attribute name svcCtx, that describes the associated service. This copy of the ServiceContext may be filtered for data that actors may not need to know, or that the associated service does not want actors to know about.

3.1.1.12.1 Public Operations

`getServiceContext() : ServiceContext`

This is a possibly modified copy of the context for the service the stub is associated with. This copy is maintained by the StubContext so that a stub may be able to accurately describe its associated service's capabilities or other data about the service that may be important to share with the stub.

`setServiceContext(ctx : ServiceContext) : void`

Sets the ServiceContext for the stub's associated service.

Arguments:

- ctx - The context for the stub's associated service
- .

3.1.2 Information

The information group provides the interfaces, contexts, and supporting components that define the information to be managed. This group also contains the context that describes the services performing the management operations. The information interfaces are:

- ActionContext (3.1.2.1)
- ConfirmationType (3.1.2.2)
- Information (3.1.2.3)
- InformationContext (3.1.2.4)
- InformationServiceContext (3.1.2.5)

3.1.2.1 ActionContext

The Action Context is used to describe the actions that can be invoked on information of specific types. This Context has two required attributes: infoTypeActions and infoTypeNames. The infoTypeActions attribute is a listing of information actions that may be invoked on information. The entries within this list correlate to the entries within the list of infoTypeNames. The infoTypeNames attribute is a listing of information type identifiers that defines what types the respective information actions are being performed upon.

3.1.2.1.1 Public Operations

`getInfoTypeActions() : List`

This method returns all the information actions that may be invoked on information that corresponds to the Type Names list.

`getInfoTypeNames() : List`

This method returns a list of information type name identifiers that defines what types the respective information Type Actions are being performed upon.

`setInfoTypeActions(actions : List) : void`

Sets the list of actions being performed upon information.

Arguments:

- actions - The list of actions being performed upon information.

`setInfoTypeNames(typeNames : List) : void`

Sets the information type names for this context.

Arguments:

- typeNames - The list of information type names.

3.1.2.2 ConfirmationType

This enumeration contains the possible types of information delivery receipts that can be requested by producers of managed information.

3.1.2.2.1 Public Fields

`CONSUMER_ACK`

This signals that the producer of the managed information wants a delivery receipt stating that the registered consumers for the information have indeed received it.

`NONE`

This signals the Submission Service that the producer wants to blindly submit information to be managed without worrying about whether or not the information was submitted successfully.

`SUBMISSION_ACK`

This signals the Submission Service that the producer wants confirmation of receipt of each instance of information as it is received by the Submission Service.

`SUBMISSION_NACK`

This signals the Submission Service that the producer wants to be notified when one of its information submission attempts fails. The meaning of "submission failure" is left to the implementation designers to define.

3.1.2.3 Information

This interface is used to wrap the data being managed as information. Instances of this interface must understand the following attributes: *metadata*, *payload*, *infoTypeName*, and *informationContext*.

1. The *metadata* is the structured data that describes the actual information being managed. This may be XML, some other markup language, a specific set of bytes, or a collection of attribute-value pairs. The makeup of the context really depends upon the particular implementation of the abstract architecture. The structure and format of the metadata can be different for different types of Information.

2. The *payload* is the actual information being managed.

3. The *infoTypeName* is the type identifier for this instance of managed information.
4. The *informationContext* is the Context that provides additional characterization for this particular piece of information.

3.1.2.3.1 Public Operations

`getInformationContext() : InformationContext`

Retrieves the Context providing additional characterization for this piece of information.

`getInformationTypeName() : String`

Retrieves the identifier for the registered type of information to which this instance of information conforms.

`getMetadata() : Object`

Retrieves the metadata for this instance of information. If none exists for this instance, then the payload must not be missing as well and this field should contain some kind of identifier or pointer that enables the retrieval of this instance's metadata.

`getPayload() : byte[]`

Retrieves the raw data that is being managed by the IM Services. If none exists for this instance then the metadata must not be missing as well and this field should contain some kind of identifier or pointer that enables the retrieval of this instance's payload.

`setInformationContext(ctx : InformationContext) : void`

Sets the Context providing additional characterization for this piece of information.

Arguments:

- ctx - The Context providing additional characterization for this piece of information.

`setInformationTypeName(typeName : String) : void`

Sets the type identifier for this instance of information.

Arguments:

- typeName - The type identifier for this instance of information.

`setMetadata(metadata : Object) : void`

Sets the metadata for this instance of information.

Arguments:

- metadata - The metadata for this instance of information.

`setPayload(payload : byte[]) : void`

Sets the payload for this instance of information.

Arguments:

- payload - The payload for this instance of information.

3.1.2.4 InformationContext

This Context provides a container for holding additional auxiliary data describing or characterizing a piece of information being managed by the Phoenix IM Services. A piece of information may or may not contain an InformationContext. Attributes of this Context include: *persistenceFlag*, *brokeringFlag*, *receiptRequestFlag*, and *matchingSubscriptions*.

1. The *persistenceFlag*, if provided, signifies that the information should be persisted. In the Phoenix architecture this would be used by the Submission Service, telling it to interact with one or more Repository Services.
2. The *brokeringFlag*, if provided, signifies that the information is to be forwarded for brokering purposes. In the Phoenix architecture this would be used by the Submission Service, telling it to interact with one or more Information Brokering Services.
3. The *receiptRequestFlag* is the flag that defines what type delivery confirmation has been requested for this instance of managed information, if any.
4. The *matchingSubscriptions* list is a listing of the IDs for the subscriptions whose predicates match the instance of managed information described by an Information Context. This list is used by consumers or intermediate services to determine which subscription(s) that have been issued match the attributes contained in the Information Context.

3.1.2.4.1 Public Operations

`addMatchingSubscriptionId(subscriptionId : String) : void`

Adds a subscription identifier to this instance of information.

Arguments:

- subscriptionId - The identifier for the subscription that matched the predicate for the instance of information.

`getBrokeringFlag() : int`

Retrieves the flag that signals what brokering mode to use for this instance of managed information.

`getMatchingSubscriptionIds() : String[]`

After an Information Brokering operation, this attribute contains the subscription identifiers for the subset of subscriptions whose registered predicates matched the metadata for this instance of information.

`getPersistenceFlag() : int`

Retrieves the flag that signals what type of persistence mode is being requested for this instance of information. Persistence modes are defined by the individual implementations of the Phoenix architecture.

`getReceiptRequestFlag() : ConfirmationType`

Retrieves the flag that signals what type of delivery confirmation is being requested by the producer of the information, if any.

`setBrokeringFlag(brokeringFlag : int) : void`

Sets the brokering mode flag for this instance of information.

Arguments:

- brokeringFlag - The brokering mode flag for this instance of information.

`setPersistenceFlag(persistenceFlag : int) : void`

Sets the persistence mode flag for this instance of information.

Arguments:

- persistenceFlag - The persistence mode flag for this instance of information.

`setReceiptRequestFlag(flag : ConfirmationType) : void`

Sets the receipt request confirmation flag for this instance of information.

Arguments:

- flag - The receipt request confirmation flag for this instance of information.

3.1.2.5 InformationServiceContext

This context is used to define a set of attributes that are common to the Contexts describing the services that operate upon Information within the Phoenix architecture. This includes the list of supported Information types, the associated service's current and maximum throughput rate (in Information Context instances per second), and a list of the actors who currently have an open and active Data Channel with the associated service.

An Information Service Context supports at least the following attributes: supportedInfoTypes, currentThroughputRate , maxSupportedThroughputRate, and connectedActors.

1. The supportedInfoTypes is a list of type identifiers that the associated Information service currently supports. It is up to the implementation of the abstract architecture to define what is meant by "supporting" an Information type. This is the list of information types that this service is allowed to operate upon.
2. The currentThroughputRate is the aggregate data throughput rate of all Data Channels currently connected to this service. It is up to the implementation of the abstract architecture to define the unit of measure for this variable's value.
3. The maxSupportedThroughputRate is the maximum supported aggregate throughput rate for the service. Again, it is up to the implementation of the abstract architecture to define the unit of measure for this variable's value.
4. The list of connectedActors contains the identifiers for the actors who currently have Data Channels established with the associated service.

3.1.2.5.1 Public Operations

`addConnectedActor(actorId : String) : void`

Adds a newly connected actor to the list of tracked actors.

Arguments:

- actorId - The actor identifier for the newly connected actor.

`getConnectedActors() : List`

Retrieves the list of identifiers for the actors who are currently connected to this service over a data channel.

`getCurrentThroughputRate() : long`

Retrieves the metric describing the current information throughput rate. The actual unit of measure is left up to the implementation.

`getMaxSupportedThroughputRate() : long`

Retrieves the theoretical maximum throughput rate for this service. The actual unit of measure is left up to the implementation.

`getSupportedInformationTypes() : List`

Retrieves the list of identifiers for the information types that are currently supported by the associated parent service.

`removeConnectedActor(actorId : String) : void`

Removes the identified actor from the list of connected actors.

Arguments:

- actorId - The identifier for the actor to be removed.

3.1.3Channel

The channel group provides the interfaces, contexts, and supporting components that define the data and control channels for interacting with the services that perform some kind of operation upon managed information. The channel interfaces are:

- BaseChannel (3.1.3)
- BaseDataServiceConnector (3.1.3.2)
- BaseDataServiceStub (3.1.3.3)
- ByteInputChannel (3.1.3.4)
- ByteOutputChannel (3.1.3.5)
- ChannelContext (3.1.3.6)
- ChannelException (3.1.3.7)
- ChannelFactory (3.1.3.8)
- ChannelState (3.1.3.9)
- DestinationContext (3.1.3.10)
- InformationChannel (3.1.3.11)
- InformationInputChannel (3.1.3.12)
- InformationOutputChannel (3.1.3.13)
- InputChannel (3.1.3.14)
- OutputChannel (3.1.3.15)
- ServiceBinding (3.1.3.16)
- Transport (3.1.3.17)
- TransportContext (3.1.3.18)
- TransportFactory (3.1.3.19)
- TransportType (3.1.3.20)

3.1.3.1 BaseChannel

This interface defines the basic methods that are to be shared by all Channels. Channels are expected to contain a ChannelContext that defines what the channel is and tracks its current state and status. Channels are the means through which information, in whatever format, are moved between Phoenix services, and are the preferred method of moving information, again in whatever format, between the Phoenix Dissemination Service and its registered consumers.

3.1.3.1.1 Public Operations

`destroy() : void`

Destroys this Channel and any underlying constructs and/or bindings. The semantics of this may or may not apply to any specific channel implementation.

Raised Exceptions:

- ChannelException

`getContext() : ChannelContext`

This method retrieves the ChannelContext that describes this Channel instance.

`isInput() : boolean`

This methods returns true if the instance is currently reading data, false otherwise.

`updateContext(attributesToUpdate : Map) : void`

Updates the ChannelContext that describes this Channel. This will potentially alter the Channel instance itself.

Arguments:

- attributesToUpdate - The attributes to be updated along with their associated new values.

3.1.3.2 BaseDataServiceConnector

This interface is an extension of the BaseServiceConnector.

All data services interact via the connector and stub model. A well known example of this model is the Remote Method Invocation (RMI) model. The connector construct represents the service side of a physical connection between the data service and an actor. Any service methods that the implementation designer wishes to expose to all actors are implemented within the connector. All methods exposed through the connector are to be implemented within the corresponding stub as well. Method invocations are forwarded to the connector by the stub and onward through the connector to the actual service itself. All return values for method invocations follow a reciprocal path through the connector and stub back to the actor.

3.1.3.2.1 Public Operations

`(none)`

3.1.3.3 BaseDataServiceStub

This is an extension of the BaseSerivceStub.

The stub, being the actor side of the physical connection provided by the connector-stub model, is the physical construct upon which the actor invokes service methods. These are the member items that are expected:

1. context - A StubContext that describes the stub and its associated service.

3.1.3.3.1 Public Operations

`close(sessionTrack : SessionTrack, channelId : String) : void`

This method closes a channel given a channel Id.

`getChannelContext(sessionTrack : SessionTrack) : ChannelContext`

This method returns the channel context for the given stub.

`getInputChannel(sessionTrack : SessionTrack, ctx : ChannelContext) : InputChannel`

This method returns an input data channel based on the channel context settings.

`getOutputChannel(sessionTrack : SessionTrack, ctx : ChannelContext) : OutputChannel`

This method returns an output data channel based on the channel context settings.

`isConnected(sessionTrack : SessionTrack, channelId : String) : boolean`

This method returns true if the stub is currently is connected, and false otherwise.

`isClosed(sessionTrack : SessionTrack, channelId : String) : boolean`

This method returns true if the stub is closed, and false otherwise.

`isInputEnabled(sessionTrack : SessionTrack, ctx : ChannelContext) : boolean`

Checks if this stub will allow creation of an input data channel of the type described by the given channel context specification.

`isOutputEnabled(sessionTrack : SessionTrack, ctx : ChannelContext) : boolean`

Checks if this stub will allow creation of an output data channel of the type described by the given channel context specification.

`updateChannelContext(sessionTrack : SessionTrack, attributesToUpdate : Map) : void`

Updates the specified attributes with the given new values.

3.1.3.4 ByteInputChannel

A channel interface used to receive bytes.

3.1.3.4.1 Public Operations

`read(b : byte[]) : void`

Reads a number of bytes from the channel into the given byte array. The total number of bytes read from the channel equals the length of the given byte array, unless the underlying protocol semantics cause an early termination to the reading process, i.e., EOL character is read or the channel is closed.

Arguments:

- b - The byte array to read data into.

Raised Exceptions:

- ChannelException

`read(b : byte[], offset : int, length : int) : void`

Reads the given number of bytes from the channel into the given array starting at the given offset position.

Arguments:

- b - The byte array to read data into.
- offset - The array index of the starting position within the given array, b, to read data into.
- length - The number of bytes to read from the channel into the given array, b.

Raised Exceptions:

- ChannelException

3.1.3.5 ByteOutputChannel

A channel interface for writing bytes.

3.1.3.5.1 Public Operations

`write(b : byte[]) : void`

Writes the given bytes to the output channel.

Arguments:

- b - The bytes to be written to the output channel.

Raised Exceptions:

- ChannelException

3.1.3.6 ChannelContext

This context is used to maintain specific parameters associated with the Input and Output Channels for a particular Transport. This context may be needed to supply additional parameters to an intermediary entity that resides between a pair of Input and Output Channels, such as data adaptors (refer to diagram below), guard technologies, or possibly routers such as Sarvega or Layer 7 boxes. Channel Contexts have seven required attributes: *channelType*, *channelState*, *currentThroughputRate*, *maxSupportedThroughputRate*, *destinationContext*, *isTyped*, and *supportedInfoTypeName*.

1. The *channelType* is an identifier whose values are defined by the Transport Type enumeration. This value defines what kind of Channel this Channel Context describes.

2. The *channelState* attribute is an identifier whose values are defined by the Channel State enumeration. This value describes the current state that the Channel associated with this Context is in.

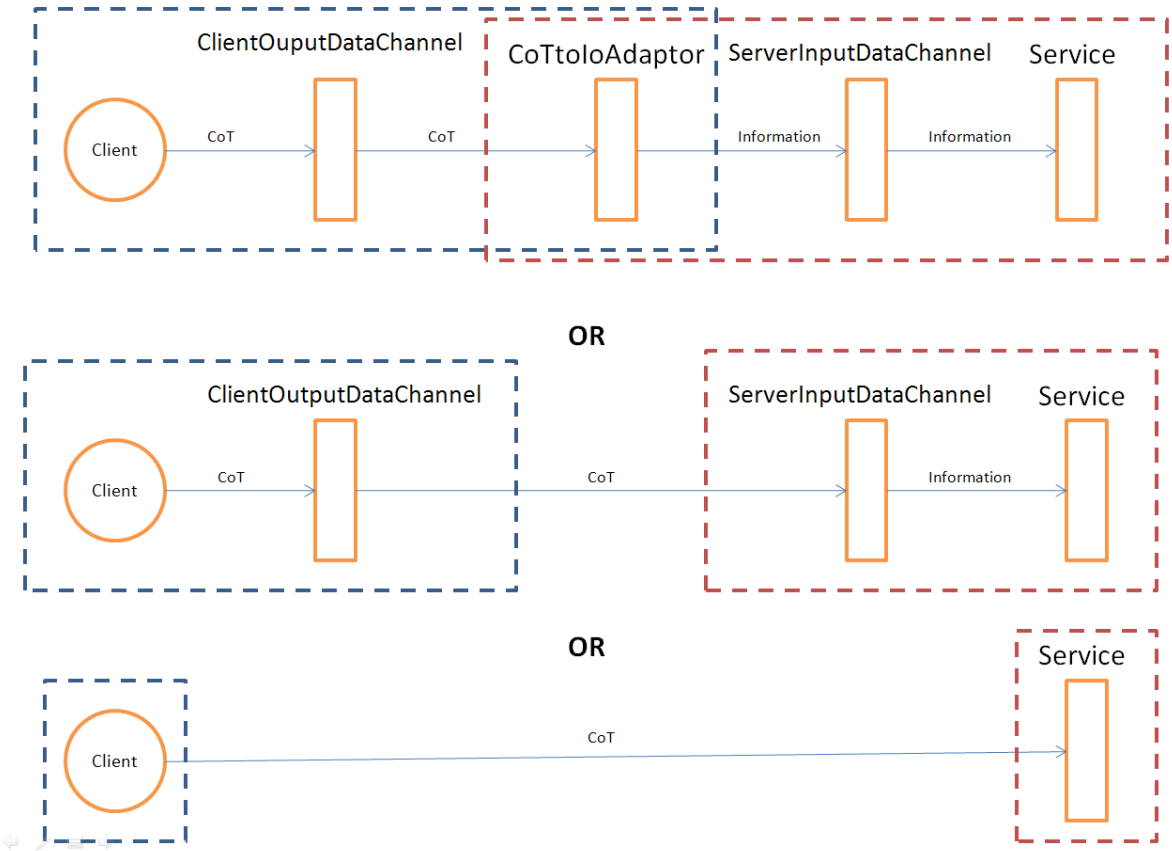
3. The *currentThroughputRate* is the rate at which data is currently flowing through the Channel associated with this Context. The implementations of the abstract architecture must define what the unit of measure is for this attribute.

4. The *maxSupportedThroughputRate* is a place holder for a maximum supported throughput rate for the associated Channel. Again, the implementations of the abstract architecture must define the units for this attribute. It should be noted that this value may be set and enforced via some kind of security or Quality of Service (QoS) policy at runtime.

5. The *destinationContext* contains the URI of the channel endpoint.

6. The *isTyped* flag is used to denote whether or not the associated channel is a typed information channel. A typed information channel is a channel that only supports sending and receiving information of a single, identified information type. An un-typed information channel supports sending and receiving information of any information type.

7. The *supportedInfoTypeName* is the information type identifier of the information type that is supported by the associated channel, if said channel is flagged to be a typed information channel. If the associated information channel is not typed, this value is null.



3.1.3.6.1 Public Operations

`getChannelState() : ChannelState`

This method returns the channelState, whose values are defined by the ChannelState enumeration. This value describes the current state of the Channel associated with this Context.

`getChannelType() : TransportType`

This method returns the channelType whose values are defined by the TransportType enumeration. This value defines what kind of Channel this Context describes.

`getCurrentThroughputRate() : double`

This method returns the current rate at which data is flowing through the Channel associated with this Context. The implementations of the abstract architecture must define what the unit of measure for this attribute is.

`getDestinationContext() : DestinationContext`

Retrieves the DestinationContext that contains the URI for the endpoint of the associated channel.

`getMaxThroughputRate() : double`

This method returns the maximum supported throughput rate for the associated Channel. Again, the implementations of the abstract architecture must define the units used for this attribute. It should be noted that this value may be set and enforced via some kind of security or Quality of Service (QoS) policy at runtime.

`getSupportedInformationTypeName() : String`

Retrieves the information type identifier for the single information type that is supported by the associated channel, if any. If the associated information channel is not typed, this value is null.

`isTyped() : boolean`

Checks if the associated information channel is typed or not. A typed information channel sends and receives information of a single information type. An un-typed information channel can send and receive information of any information type.

`setChannelState(state : ChannelState) : void`

Sets the current state of the channel.

Arguments:

- state - The current state of the channel. Possible values are defined by the ChannelState enumeration.

`setDestinationContext(ctx : DestinationContext) : void`

Sets the DestinationContext that contains the URI for the endpoint of the associated channel.

Arguments:

- ctx - The DestinationContext that contains the URI for the endpoint of the associated channel.

`setSupportedInformationTypeName(infoTypeName : String) : void`

Sets the information type identifier for the single information type that is supported by the associated channel, if any. If the associated information channel is not typed, this value is null.

Arguments:

- infoTypeName - The information type identifier for the single information type that is supported by the associated channel, if any. If the associated information channel is not typed, this value is null.

3.1.3.7 ChannelException

The ChannelException class represents an exception that is specific to Channel operations.

3.1.3.7.1 Attributes

`channelContext`

The ChannelContext for the failed operation.

3.1.3.7.2 Public Operations

`getRelatedChannelContext() : ChannelContext`

Retrieves the ChannelContext that is related to the raised exception.

3.1.3.8 ChannelFactory

This interface creates specific channel constructs that are used to perform the physical read and write operations that actually move the data to be transmitted and received.

3.1.3.8.1 Public Operations

`createByteInputChannel(ctx : ChannelContext) : ByteInputChannel`

Creates an input channel for bytes.

Arguments:

- ctx - The input channel for reading bytes.

Raised Exceptions:

- ChannelException

`createByteOutputChannel(ctx : ChannelContext) : ByteOutputChannel`

Creates an output channel for bytes.

Arguments:

- ctx - The output channel for writing bytes.

Raised Exceptions:

- ChannelException

`createInformationInputChannel(ctx : ChannelContext) : InformationInputChannel`

Creates an input channel for information.

Arguments:

- ctx - The input channel for reading information.

Raised Exceptions:

- ChannelException

`createInformationOutputChannel(ctx : ChannelContext) : InformationOutputChannel`

Creates an output channel for information.

Arguments:

- ctx - The output channel for writing information.

Raised Exceptions:

- ChannelException

`createInputChannel(ctx : ChannelContext) : InputChannel`

Creates an input channel for some construct other than those natively supported by the Phoenix architecture, i.e., a construct other than bytes, events, or information.

Arguments:

- ctx - The input channel for reading constructs not native to the Phoenix architecture.

Raised Exceptions:

- ChannelException

`createOutputChannel(ctx : ChannelContext) : OutputChannel`

Creates an output channel for some construct other than those natively supported by the Phoenix architecture, i.e., a construct other than bytes, events, or information.

Arguments:

- ctx - The output channel for writing constructs not native to the Phoenix architecture.

Raised Exceptions:

- ChannelException

3.1.3.9 ChannelState

This enumeration lists the possible states for a Data Channel.

3.1.3.9.1 Public Fields

`ACTIVE_READ`

The Data Channel is currently reading data.

`ACTIVE_WRITE`

The Data Channel is currently writing data.

`TIME_WAIT`

The Data Channel is currently closed and awaiting destruction.

3.1.3.10 DestinationContext

This context contains the collection of attributes that describe a physical end point associated with an actor. This is currently used for defining the end points of subscriptions and queries but may also be used for defining end points for services. The attributes for Destination Contexts include: consumerURI and inBandConsumer.

The consumerURI is the actual physical location of the consumer on the network. The URI syntax is consistent with the Internet Engineering Task Force (IETF) Request for Comments 1630 and is typically defined as a combination of Internet Protocol (IP) address, port, and wire protocol. In the Phoenix architecture, the URI may also contain other encoded information that may be of use for such things as underlying channel provisioning, etc. The inBandConsumer flag is used to denote the difference between in-band and out-of-band consumers. This will signal the services whether or not this consumer expects delivery of Information through IMS Channels or from other means. This is a boolean flag.

3.1.3.10.1 Public Operations

`getConsumerURI() : URI`

Retrieves the URI that represents the physical location of the consumer on the network.

`isInBandConsumer() : boolean`

Retrieves the flag that specifies whether or not this consumer is an in-band or out-of-band consumer. In-band consumers receive information from Phoenix services directly via channels. Out-of-band consumers receive data from external actors and only use the Information Brokering capability of Phoenix to identify what information is of interest.

`setConsumerURI(uri : URI) : void`

Sets the consumer URI.

Arguments:

- uri - The URI that defines the location of the consumer.

`setInBandConsumerFlag(isInBand : boolean) : void`

Sets the flag for in-band or out-of-band consumer.

Arguments:

- isInBand - The in-band consumer flag.

3.1.3.11 InformationChannel

A Channel that reads and writes information instances. This inherently implies that an implementation of an information channel can successfully serialize or de-serialize information instances.

3.1.3.11.1 Public Operations

`getSupportedInformationTypeName() : String`

Retrieves the information type identifier for the single information type that is supported by this channel, if any. If this channel is not typed, this value is null.

`isTyped() : boolean`

Checks if this channel is typed or not. A typed information channel sends and receives information of a single information type. An un-typed information channel can send and receive information of any information type.

3.1.3.12 InformationInputChannel

This interface defines a specific Channel that provides a way to read variously typed information instances instead of raw bytes.

3.1.3.12.1 Public Operations

`read() : Information`

Reads an instance of information from the Channel.

Raised Exceptions:

- ChannelException

`read(i : Information[]) : void`

Reads a set of information instances from the Channel.

Raised Exceptions:

- ChannelException

3.1.3.13 InformationOutputChannel

This interface defines a specific Channel that provides a way to write variously typed information instances instead of raw bytes.

3.1.3.13.1 Public Operations

`write(info : Information) : void`

Writes an instance of information to the Channel.

Arguments:

- info - The information instance that is to be written to the Channel.

Raised Exceptions:

- ChannelException

3.1.3.14 InputChannel

An input Channel is used to read bytes from the channel, either one at a time or in an array of a given byte size. It is up to the implementation to determine what to do when a broken connection is detected (in terms of any possible buffered bytes and possible recovery semantics).

3.1.3.14.1 Public Operations

`accept() : void`

This is a blocking operation that listens for a connection attempt by a corresponding output channel of a similar kind. Once the connection attempt is successful, this method returns control to the invoking thread of execution and the input channel is ready to be used for reading data.

Raised Exceptions:

- ChannelException

`available() : int`

Gets a best guess as to how much data (information, events, bytes, etc.) is currently available to be read from the Channel.

3.1.3.15 OutputChannel

An output Channel is used to output data to a channel. The format of the data is defined by the specific sub-interfaces of this interface, i.e., information, event, bytes, etc.

3.1.3.15.1 Public Operations

`connect() : void`

Connects this output channel to a corresponding input channel of the same wire protocol.

Raised Exceptions:

- ChannelException

`disconnect() : void`

Disconnects this output channel from the corresponding input channel of the same wire protocol.

Raised Exceptions:

- ChannelException

3.1.3.16 ServiceBinding

This construct contains a copy of a service's control stub and transport. Service bindings are created and managed by individual services. These bindings are then registered with and may be obtained from the Phoenix Service Brokering Service. Bindings can be transferred from one actor to another. Once an actor obtains a service binding it can communicate directly with the associated service via the encapsulated stub or input and output transports.

3.1.3.16.1 Public Operations

`getControlStub() : BaseServiceStub`

This method returns a stub for the given service that this service binding object is holding.

`getTransport() : Transport`

This method returns a data transport that can be used by the actor to create channels for transmitting data to the associated service.

3.1.3.17 Transport

Transports are created by transport factories or server transports. A receiver of data should create a transport via a server transport while a transmitter of data should create a transport directly via the transport factory.

3.1.3.17.1 Public Operations

`getChannelFactory() : ChannelFactory`

Retrieves the ChannelFactory for this transport.

`getTransportContext() : TransportContext`

Retrieves the TransportContext that describes this transport.

`setChannelFactory(factory : ChannelFactory) : void`

Sets the ChannelFactory for this transport.

Arguments:

- factory - The ChannelFactory for this transport.

`setTransportContext(ctx : TransportContext) : void`

Sets the TransportContext that describes this transport.

Arguments:

- ctx - The TransportContext that describes this transport.

3.1.3.18 TransportContext

This context is used by the Transport Factory to maintain information associated with the Transport that an actor wishes to use. For example, an actor could provide a Transport Context requesting a SSL-enabled Transport for secure Channels. This context contains two standard attributes named TransportType and endPointURI. The TransportType attribute's possible values are listed in the Transport Type enumeration. The endPointURI is just that, a URI to the actor on the other end of the transport.

3.1.3.18.1 Public Operations

`getDestinationContext() : DestinationContext`

This method returns the DestinationContext that contains the URI for the endpoint of this transport's associated channels.

`getTransportType() : TransportType`

This method returns the type of Transport that this context describes. Possible values are defined in the TransportType enumeration.

`setDestinationContext(ctx : DestinationContext) : void`

This method sets the DestinationContext that contains the URI for the endpoint of this transport's associated channels.

Arguments:

- ctx - The DestinationContext that contains the URI for the endpoint of this transport's associated channels.

`setTransportType(type : TransportType) : void`

Sets the type of Transport.

Arguments:

- type - The type of Transport. Possible values are defined in the TransportType enumeration.

3.1.3.19 TransportFactory

A transport factory creates instances of Transports based upon some specified criteria.

3.1.3.19.1 Public Operations

`addSupportedTransportType(transportType : String, transportQClassName : String) : void`

Adds a supported transport type.

Arguments:

- transportType - The unique identifier for the transport, i.e., tcp, mockets, ftp.
- transportQClassName - The fully qualified class name of the transport for the identified transport type.

`createTransport(ctx : TransportContext) : Transport`

This method returns the Transport given the TransportContext and its appropriate setting.

Arguments:

- ctx - The TransportContext that describes the Transport that this factory will create.

`getSupportedTransportTypes() : String[]`

Retrieves a listing of the currently supported transport types.

`removeSupportedTransportType(transportType : String) : void`

Removes a transport type from the supported types listing.

Arguments:

- transportType - The unique identifier for the transport, i.e., tcp, mockets, ftp.

3.1.3.20 TransportType

This enumeration contains the possible types of Transports defined by the Phoenix architecture.

3.1.3.20.1 Public Fields

`BYTE`

Identifies a Transport as one that reads/writes raw bytes.

`EVENT`

Identifies a Transport as one that reads/writes event objects.

`INFORMATION`

Identifies a Transport as one that reads/writes information instances.

`OTHER`

Identifies a Transport as one that reads/writes objects or data other than those that are native to the Phoenix architecture, i.e. a transport that supports something other than bytes, events, or information.

3.1.4Exceptions

The exceptions group provides the supporting components that define the high level exceptions that can be raised and handled by the Phoenix IM Services. The exceptions components are:

- ApplicationException (3.1.4.1)
- Exception (3.1.4.2)
- ExceptionType (3.1.4.3)
- MetadataValidationFailedException (3.1.4.4)
- MultipleApplicationException (3.1.4.5)
- PayloadValidationFailedException (3.1.4.6)
- ServiceException (3.1.4.7)
- UnableToCreateServiceException (3.1.4.8)
- UnableToGenerateUidException (3.1.4.9)
- ValidationFailedException (3.1.4.10)

3.1.4.1 ApplicationException

Application exceptions occur due to something an actor or actors attempts to do with the Phoenix IM Services. The Phoenix architecture defines some examples of Application exceptions, but it leaves the implementation designers to complete the hierarchy and definition of Application exceptions.

3.1.4.2 Exception

This class represents the top level of the Phoenix exception hierarchy and should extend the generic Exception class defined by the chosen implementation language. For example, in Java this class would extend the java.lang.Exception class. For implementation languages without the notion of exceptions, the implementation designers will need to either design around this notion or fabricate something that acts as a facade for Exceptions. This class contains three attributes: *exceptionCode*, *exceptionType*, and *exceptionMessage*.

3.1.4.2.1 Attributes

`exceptionCode`

: This is a number of the meaning of the base exception. Implementations can use their own number scheme. This is analogous to the SQL or HTTP error codes.

`exceptionType`

: This defines the type of Phoenix exception that occurred. The possible values for this attribute are defined by the Exception Type enumeration.

`exceptionMessage`

: A message that describes the exception that occurred. This is meant to contain something human readable.

3.1.4.2.2 Public Operations

`getExceptionCode() : int`

Retrieves the code for this exception.

`getExceptionType() : ExceptionType`

Retrieves the type identifier for this exception.

`getExceptionMessage() : String`

Retrieves the message for this exception.

3.1.4.3 ExceptionType

3.1.4.4 MetadataValidationFailedException

A ValidationFailedException sub-class specifically for metadata validation failures.

3.1.4.5 MutlipleApplicationException

The multiple exception is for use when a "bulk" processing operation is undertaken. This special exception class is needed to support the semantics necessary to describe what portions of the "bulk" processing passed and what failed. This class contains two required attributes: *errors*, and *successes*.

3.1.4.5.1 Attributes

`errors`

A Map whose keys are the indices from the submitted List of arguments of the data that caused this error. The values are a string message that describes the errors for the associated piece of data.

`successes`

A Map whose keys are the indices from the submitted List of arguments of the data that caused this exception. The values are the actual return values for the successful invocations for the pieces of submitted data that passed.

3.1.4.5.2 Public Operations

`addError(index : int, errorMsg : String) : void`

Adds an error to the error listing.

Arguments:

- index : int - The index of the "bulk" processing item that caused the described failure.
- errorMsg : String - The human readable error message describing what failed.

`addSuccess(index : int, successMsg : String) : void`

Adds a success to the success listing.

Arguments:

- index : int - The index of the "bulk" processing item that passed.
- successMsg : String - The human readable success message describing what passed.

`getErrors() : Map`

Retrieves the Map that contains the errors and their respective messages.

`getSuccesses() : Map`

Retrieves the Map that contains the successes and their respective messages.

3.1.4.6 PayloadValidationFailedException

A ValidationFailedException sub-class specifically for payload validation failures.

3.1.4.7 ServiceException

This class defines an exception that occurred due to something that one or more of the Phoenix IM Services did. The Phoenix architecture defines some example service exceptions but the majority of the exception hierarchy is left to the implementation designers to complete.

This section specifies the direct descendants of the ServiceException class:

- PredicateFailureException
- UnableToCreateServiceException
- UnableToGenerateUidException
- ValidationFailedException

3.1.4.8 UnableToCreateServiceException

This class is an example of an ServiceException and defines an exception for describing the specific case where the service management layer is unable to create an implementation of one of the Phoenix IM Services.

3.1.4.9 UnableToGenerateUidException

This exception class is an example ServiceException and describes the specific case where a service was unable to generate a new unique identifier for some entity within the implementation of the Phoenix IM Services.

3.1.4.10 ValidationFailedException

A ServiceException that provides a mechanism to track exactly what data failed the associated validation attempt.

3.1.4.10.1 Attributes

`data : byte[]`

The data that failed the validation attempt stored in its binary form to prevent corruption.

`infoTypeName : String`

The identifier for the information type that the data is associated with.

3.1.4.10.2 Public Operations

`getInvalidaData() : byte[]`

Retrieves the invalid data in its binary format.

`getInvalidDataAsString() : String`

Retrieves the invalid data as a String object.

`getInformationTypeName() : String`

Retrieves the information type identifier for the data that failed to validate.

3.1.5 Predicate

The predicate group provides the interfaces, contexts, and supporting components that define the Phoenix architecture's support for predicate definition and processing. The predicate interfaces are:

- Evaluator (3.1.5.1)
- EvaluatorFactory (3.1.5.2)
- InformationPredicateContext (3.1.5.3)
- PredicateContext (3.1.5.4
- PredicateFailureException (3.1.5.5)
- PredicateTypeNotSupportedException (3.1.5.6)
- QueryContext (3.1.5.7)

3.1.5.1 Evaluator

Evaluates predicates against information instances.

3.1.5.1.1 Public Operations

`evaluate(information : Information) : boolean`

Evaluates the predicate against the supplied information.

Arguments:

- information - The information to be evaluated. This could potentially be the full piece of information or a stripped down copy (i.e., the metadata only or the payload only).

Raised Exceptions:

- PredicateFailureException

`getSupportedPredicateType() : String`

Gets the predicate type that this evaluator processes.

`prepareForEvaluation(predicate : PredicateContext, metadataSchema : Schema, payloadSchema : Schema) : void`

Prepares this evaluator for real-time brokering of metadata by performing any required pre-processing of the supplied predicate.

Arguments:

- predicate - The PredicateContext that describes the predicate.
- metadataSchema - The metadata schema for use if needed when translating the predicate into a usable format.
- payloadSchema - The payload schema for use if needed when translating the predicate into a usable format.

Raised Exceptions:

- PredicateFailureException

3.1.5.2 EvaluatorFactory

Creates Evaluator instances based on predicate type.

3.1.5.2.1 Public Operations

`addSupportedPredicateType(predicateType : String, evaluatorQClassName : String) : void`

Adds a new predicate type binding to this factory.

Arguments:

- predicateType - The predicate type identifier.
- evaluatorQClassName - The fully qualified class name of the Evaluator to use for the accompanying predicate type.

`createEvaluator(predicateType : String) : Evaluator`

Creates an Evaluator instance for the supplied predicate type.

Arguments:

- predicateType - The identifier of the predicate type for which the factory will attempt to create a new Evaluator instance for.

Raised Exceptions:

- PredicateTypeNotSupportedException

`getSupportedPredicateTypes() : Map`

Retrieves a Map of predicate types and their associated Evaluator class names.

`removeSupportedPredicateType(predicateType : String) : void`

Removes a predicate type from the list of types supported by this factory.

Arguments:

- predicateType - The predicate type that will no longer be supported.

3.1.5.2.2 Typical Use

The EvaluatorFactory is typically used to create predicate-type-specific Evaluator instances for predicate processing.

3.1.5.3 InformationPredicateContext

This Context contains the attributes and their helper operations that are specific to information brokering and query operations.

3.1.5.3.1 Public Operations

`addDestinationContext(ctx : DestinationContext) : void`

Binds a new consumer location to this predicate.

Arguments:

- ctx - The consumer location to bind to the predicate.

`addInformationTypeName(infoTypeName : String) : void`

Binds this predicate to an information type identifier.

Arguments:

- infoTypeName - The identifier for the information type to apply this predicate to.

`getDestinationContexts() : DestinationContext[]`

Retrieves the entire list of consumers bound to this predicate.

`getInformationTypeNames() : String[]`

Retrieves the entire list of identifiers for the information types that this predicate should be applied to.

`removeDestinationContext(consumerId : String) : void`

Removes the consumer location that is identified.

Arguments:

- consumerId - The identifier for the consumer location to be removed.

`removeInformationTypeName(infoTypeName : String) : void`

Removes an information type identifier.

Arguments:

- infoTypeName - The information type identifier to be removed.

3.1.5.4 PredicateContext

This Context contains all attributes that are common to descriptions of requests for data, also known as predicates.

3.1.5.4.1 Public Operations

`getPredicate() : String`

Retrieves the constraint(s) to be applied over the metadata during the brokering process. This defines what information is of interest to the consumer requesting the brokering operation.

`getPredicateType() : String`

Retrieves the type of predicate that this Context describes.

`getTimeToLive() : long`

Retrieves the amount of time that the inquisitor is willing or able to listen for a response to its query request. This response may be the return of a result set size (synchronous operation) or the start of the result set stream over the corresponding data channel(s) (asynchronous operation).

`setPredicate(predicate : String) : void`

Sets the predicate instance.

Arguments:

- predicate - The predicate instance.

`setPredicateType(predicateType : String) : void`

Sets the type of predicate.

Arguments:

- predicateType - The predicate type identifier.

`setTimeToLive(ttl : long) : void`

Sets the amount of time that the inquisitor is willing or able to listen for a response to its query request. This response may be the return of a result set size (synchronous operation) or the start of the result set stream over the corresponding data channel(s) (asynchronous operation).

Arguments:

- ttl - The amount of time that the inquisitor is willing or able to listen for a response to its query request. This response may be the return of a result set size (synchronous operation) or the start of the result set stream over the corresponding data channel(s) (asynchronous operation).

3.1.5.5 PredicateFailureException

A ServiceException that encompasses all brokering operations that utilize predicates.

3.1.5.5.1 Attributes

`predicate : InformationBrokeringContext`

The Context that describes the predicate that failed evaluation.

3.1.5.5.2 Public Operations

`getFailedPredicate() : InformationBrokeringContext`

Retrieves a copy of the Context that describes the predicate that failed whatever operation that was attempted.

3.1.5.6 PredicateTypeNotSupportedException

Identifies an error that occurred when attempting to create an Evaluator instance for a specific predicate type.

3.1.5.6.1 Attributes

`predicateType`

The predicate type identifier that resulted in the raising of this Exception.

3.1.5.6.2 Public Operations

`getUnsupportedPredicateType() : String`

Retrieves the predicate type identifier that resulted in the raising of this Exception.

3.1.5.7 QueryContext

The Query Context extends the Information Brokering Context. A query is essentially a brokering operation applied to persisted information. The Query Context may contain such things as the query turnaround time and time to live as well as upper and lower limits for result set size.

The Query Context inherits the attributes of the Information Brokering Context. All of these attributes still make sense when applied to persistent Information instead of the live Information flowing through the IMS. In addition, the Query Context defines one attribute specific to query operations: *turnAroundTime*.

1. The *turnAroundTime* is the total amount of time that the inquisitor is willing to wait for the data store to execute the provided query and return the full result set. Once this time limit has been reached any further results will either be actively rejected, ignored or the inquisitor will perform some other action, depending upon the implementation.

QueryContext has two direct descendants: ServiceBrokeringQueryContext and InformationQueryContext.

3.1.5.7.1 Public Operations

`getTurnAroundTime() : long`

Retrieves the maximum amount of time that the inquisitor is willing or able to listen for the complete result set of the query they submitted for execution. This may be implemented such that it contains the time required to execute the query or not.

`setTurnAroundTime(tat : long) : void`

Sets the maximum amount of time that the inquisitor is willing or able to listen for the complete result set of the query they submitted for execution. This may be implemented such that it contains the time required to execute the query or not.

Arguments:

- tat - The maximum amount of time that the inquisitor is willing or able to listen for the complete result set of the query they submitted for execution. This may be implemented such that it contains the time required to execute the query or not.

3.1.6 Event

The event group provides the interfaces, contexts, and supporting components that define the components necessary for supporting the concept of event notification. The event interfaces are:

- DeliveryReceiptEvent (3.1.6.1)
- Event (3.1.6.2)
- EventContext (3.1.6.3)
- EventInputDataChannel (3.1.6.4)
- EventOutputDataChannel (3.1.6.5)
- EventType (3.1.6.6)
- ExceptionEvent (3.1.6.7)
- InformationTypeEvent (3.1.6.8)
- InformationTypeStatus (3.1.6.9)
- ReceiptType (3.1.6.10)

3.1.6.1 DeliveryReceiptEvent

This specific sub-class of Event provides a helper interface for handling Events of type DELIVERY_RECEIPT.

3.1.6.1.1 Public Operations

`getOriginatingActors() : List`

Retrieves the list of actors to whom this DELIVERY_RECEIPT Event applies.

`getReceiptType() : ReceiptType`

The flag that identifies the type of DELIVERY_RECEIPT Event this is.

3.1.6.2 Event

Events are used for passing messages or other pieces of unmanaged data between actors. The Phoenix architecture defines a few instances of Event sub-classes (DeliveryReceiptEvent, InformationTypeEvent, and ExceptionEvent). This hierarchy is meant to be a shell set of common event classes that can be extended by each implementation of the architecture as needed.

3.1.6.2.1 Public Operations

`getEventBody() : byte[]`

Retrieves the body of the Event. This method returns a byte array.

`getEventId() : String`

Retrieves the identifier for this Event.

`getEventType() : EventType`

Retrieves the flag that identifies the Event type. Possible values are identified by the EventType enumeration.

`getOriginatingURI() : URI`

Retrieves the URI of the actor who generated this Event.

3.1.6.3 EventContext

This context describes additional detail about an event. Like all other contexts, the event context may be extended through the definition of additional attributes. For example, if the event type is "subscriber joined", the context may include details such as the specific information type, for which event notification is requested. Most of the attributes for this Context would depend upon the definition of the Event object hierarchy, if any, and its branches. In general, this Context contains only one required attribute, the eventType, whose possible values are dependent upon implementation decisions. Some example possible values for this attribute are defined by the Event Type enumeration.

3.1.6.3.1 Public Operations

`getEventType() : EventType`

Gets the identifier that signals what type of Event this context describes. Possible values are identified within the EventType enumeration.

3.1.6.4 EventInputDataChannel

Event input channels are used to read Event instances from a channel.

3.1.6.4.1 Public Operations

`readEvent() : Event[]`

Reads a set of Event instances from the Data Channel.

3.1.6.5 EventOutputDataChannel

Event output channels are used to write Event instances to a channel.

3.1.6.5.1 Public Operations

`writeEvent(evt : Event) : void`

Writes the specified Event to the Data Channel.

Arguments:

- evt - The Event to be written to the channel.

3.1.6.6 EventType

This enumeration contains the possible types of Events that have been defined by the Phoenix architecture.

3.1.6.6.1 Public Fields

`DELIVERY_RECEIPT`

Identifies a delivery confirmation receipt Event.

`INFO_TYPE_ACTIVE`

Identifies an Event that signifies an information type has become active for an actor.

`INFO_TYPE_INACTIVE`

Identifies an Event that signifies an information type has become inactive for an actor.

3.1.6.7 ExceptionEvent

This Event is used to report Exceptions asynchronously.

3.1.6.7.1 Public Operations

`getException() : Exception`

Retrieves the Exception that is contained within this Event.

3.1.6.8 InformationTypeEvent

An Event that specifies the status of a specific information type for an actor.

3.1.6.8.1 Public Operations

`getStatusType() : InformationTypeStatus`

Retrieves the status of the information type. Possible status values are defined by the InformationTypeStatus enumeration.

3.1.6.9 InformationTypeStatus

This enumeration lists the possible information types for a status value.

3.1.6.9.1 Public Fields

`CONSUMER_FOUND`

There is at least one registered consumer of this information type.

`CONSUMER_NOT_FOUND`

There are no consumers registered to receive this information type.

`PRODUCER_FOUND`

There is at least one registered producer of this information type.

`PRODUCER_NOT_FOUND`

There are no producers registered for this information type.

3.1.6.10 ReceiptType

This enumeration contains the possible types of DELIVERY_RECEIPT Events that have been defined by the Phoenix architecture.

3.1.6.10.1 Public Fields

`CONSUMER_DELIVERY_ACK`

Identifies an Event that signifies the corresponding consumers did receive the associated instance of information.

`SUBMISSION_SERVICE_DELIVERY_ACK`

Identifies an Event that signifies the corresponding Submission Service did receive the associated instance of information.

`SUBMISSION_SERVICE_DELIVERY_NACK`

Identifies an Event that signifies the corresponding Submission Service did not receive the associated instance of information.

3.1.7 Filter

The filter group provides the interfaces, contexts, and supporting components that define the filtering capability described by the Phoenix architecture. The filter interfaces are:

- Filter (3.1.7.1)
- FilterContext (3.1.7.2)
- FilterDirection (3.1.7.3)
- FilterRegistry (3.1.7.4)
- ResidualProcessorFilter (3.1.7.5)

3.1.7.1 Filter

The Phoenix architecture also defines an interface for applying filters. A filter can be applied to Data Channels or any object since it handles byte arrays, as well as Information instances. These filters read the data being written to or read from the associated byte array and apply some operation to it that may result in a modified form of the data. Filters can be used for any type of operation, from compression and decompression to scrubbing sensitive data. The Phoenix architecture also defines a very specific child construct of the generic Data Channel filter that takes any residual data that has been removed from or modified within the raw data and outputs this residual data on its own Data Channel. This could be used for things such as splitting data into separate channels or for logging of filtering operations.

Filters are maintained by Filter Registries.

3.1.7.1.1 Attributes

`filterId : String`

The unique identifier for a specific filter.

`direction : FilterDirection`

The direction of the filter, as defined by the FilterDirection.

`parentChannelCtx : ChannelContext`

The Context that described the parent Data Channel, if indeed this filter has been applied to one.

`filterCtx : FilterContext`

The Context that describes this filter.

3.1.7.1.2 Public Operations

`filter(bytes : byte[]) : byte[]`

Takes the given byte array and performs some type of filtering logic and operation upon it. It returns the filtered bytes.

Arguments:

- bytes - The byte array to be filtered.

`filterInformation(infoCtx : InformationContext) : InformationContext`

Takes the given InformationContext and performs some type of filtering logic and operation upon it. It returns the filtered InformationContext.

Arguments:

- infoCtx - The InformationContext to be filtered.

`getFilterContext() : FilterContext`

Retrieves the Context that describes this filter.

`processResidualData(data : byte[]) : void`

This method processes any residual data that results from the filtering process. A specific sub-class of filter (ResidualProcessorFilter) has been defined by the Phoenix architecture for utilizing this data.

Arguments:

- data - The array of bytes that are left over from the filtering process.

`setDataChannelContext(channelCtx : ChannelContext) : void`

Sets the ChannelContext that describes the output data channel.

Arguments:

- channelCtx - The ChannelContext that describes the output channel.

`update(attributeName : Object, attributeValue : Object) : void`

Updates the a filter setting.

Arguments:

- attributeName - The name of the setting to be updated.
- attributeValue - The new value for the setting.

3.1.7.2 FilterContext

This context contains the implementation-specific attributes that describe a particular filter. It is envisioned that some or all of the attributes may be used to tailor the behavior of the filter.

3.1.7.3 FilterDirection

This enumeration lists the possible directions for a Filter.

3.1.7.3.1 Public Fields

`INPUT`

Identifies a Filter that applies to only input operations.

`INPUT_OUTPUT`

Identifies a Filter that applies to both input and output operations.

`OUTPUT`

Identifies a Filter that applies to only output operations.

3.1.7.4 FilterRegistry

A filter registry tracks and manages Filters that have been applied to a Data Channel or other operation.

3.1.7.4.1 Public Operations

`getFilter(index : int) : Filter`

Returns the filter for the given index.

Arguments:

- index - Returns the Filter at the given index in the filtering chain.

`getFilterCount() : int`

Returns the number of filters in the chain.

`getFilters() : List`

Returns the full list of filters in the chain.

`insertFilter(filter : Filter) : int`

Inserts a Filter into the chain of Filters at the end location. This method returns the new total number of Filters.

Arguments:

- filter - The Filter to be inserted at the end.

`insertFilter(filter : Filter, index : int) : int`

Inserts a Filter into the chain of Filters at a specific location. This method returns the new total number of Filters.

Arguments:

- filter - The Filter to be inserted.
- index - The index at which to insert the new Filter.

`removeFilter(filterId : String) : void`

Removes the Filter with the given identifier.

Arguments:

- filterId - The identifier for the Filter to be removed.

`removeFilter(index : int) : void`

Removes the Filter at the given index in the filtering chain.

Arguments:

- index - The index of the Filter to be removed.

`updateFilter(filterId : String, attributesToUpdate : Map) : void`

Updates the specified attributes for the identified Filter.

Arguments:

- filterId - The identifier for the Filter to be updated.
- attributesToUpdate - The attributes to be updated.

3.1.7.5 ResidualProcessorFilter

This specific sub-class of Filter is used for processing and outputting data left over from the filtering process.

3.1.7.5.1 Attributes

`residualDataChannel : OutputDataChannel`

The OutputDataChannel to which this Filter writes its residual data.

3.1.7.5.2 Public Operations

`setResidualOutputDataChannel(outputDataChannel : OutputDataChannel) : void`

Sets the OutputDataChannel to which the residual data will be written.

Arguments:

- outputDataChannel - The OutputDataChannel to which the residual data will be written.

3.2 Service Interfaces

Service interfaces define the functionality for each service as well as any specific supporting interfaces required by each individual service. The service interfaces have been sub-divided into thirteen functional groups, in order of maturity:

- Information Type Management (3.2.1)
- Submission (3.2.2)
- Information Brokering (3.2.3)
- Dissemination (3.2.4)
- Subscription (3.2.5)
- Query (3.2.6)
- Repository (3.2.7)
- Event Notification (3.2.8)
- Service Brokering (3.2.9)
- Session Management (3.2.10)
- Security (3.2.11)
- Client (3.2.12)
- Information Discovery (3.2.13)

3.2.1 Information Type Management

The information type management group contains the interfaces that define the service that creates, manages, and destroys information type definitions. The Information type management interfaces are:

- InformationTypeActions (3.2.1.1)
- InformationTypeContext (3.2.1.2)
- InformationTypeManagementService (3.2.1.3)
- InformationTypeManagementServiceConnector (3.2.1.4)
- InformationTypeManagementServiceStub (3.2.1.5)
- Schema (3.2.1.6)

3.2.1.1 InformationTypeActions

This enumeration contains the possible actions that may be implemented upon instances of managed information.

3.2.1.1.1 Public Fields

`ARCHIVE`

Store the information instance in a high-capacity, high-latency data store for later retrieval.

`BROKER`

Broker the information for delivery to interested consumers.

`DISSEMINATE`

Deliver the information to interested consumers.

`PERSIST`

Store the information instance in a low-capacity, low-latency data store for later retrieval.

`QUERY`

Query for stored information.

`SUBMIT`

Submit information to be managed.

`SUBSCRIBE`

Register the intent to receive information as it is brokered.

3.2.1.2 InformationTypeContext

This Context describes the attributes necessary to define an information type within the architecture. It is used by the Information Type Management Service to create, delete, and archive information types.

This Context has three required attributes: *typeName*, *metadataSchema*, and *payloadSchema*.

1. The *typeName* is the identifier for the information type as it will be known to the IM Services. This is used by all actors to identify information of this kind.
2. The *metadataSchema* is the definition of the structure of the metadata used to describe the payload. For example, if the metadata for a type of information is to be described by Extensible Markup Language (XML) documents, the *metadataSchema* for that type would be an XML schema document (XSD) stored within a Schema object.
3. The *payloadSchema* is the definition of the structure of the payload. For example, if the payload for a type of information is to be described by Extensible Markup Language (XML) documents, the *payloadSchema* for that type would be an XML schema document (XSD) stored within a Schema object.

3.2.1.2.1 Public Operations

`getInformationTypeName() : String`

Retrieves the unique identifier for this registered information type.

`getMetadataSchema() : Schema`

Retrieves the object defining the structure of the metadata for this information type.

`getPayloadSchema() : Schema`

Retrieves the object defining the structure of the payload for this information type.

`setInformationTypeName(typeName : String) : void`

Sets the information type identifier.

Arguments:

- typeName - The information type identifier.

`setMetadataSchema(schema : Schema) : void`

Sets the metadata schema.

Arguments:

- schema - The metadata schema object.

`setPayloadSchema(schema : Schema) : void`

Sets the payload schema.

Arguments:

- schema - The payload schema object

3.2.1.3 InformationTypeManagementService

The interface for information type management will provide methods for registering, retrieving, and deleting information type definitions.

3.2.1.3.1 Public Operations

`archiveTypeDefinitions(sessionTrack : SessionTrack, typeCtxs : InformationTypeContext[]) : void`

This method archives the definition of a specified information type. This is for the case where information of a certain type is moved to the offline archive, but the description of said type is still needed so that the offline archive can be queried and sense made of the information being returned.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- typeCtxs - The Information Type Contexts describing what information types are to be archived.

Raised Exceptions:

- MultipleApplicationException

`createInformationTypes(sessionTrack : SessionTrack, typeCtxs : InformationTypeContext[]) : void`

Creates a new information type for a specified format of metadata and payload.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- typeCtxs - The Information Type Contexts containing the definitions of the new information types.

Raised Exceptions:

- MultipleApplicationException

`deleteTypeDefinitions(sessionTrack : SessionTrack, typeCtxs : InformationTypeContext[]) : void`

Deletes the specified information type definition and all records of this type from the data store.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- typeCtxs - The Information Type Contexts describing what information types are to be deleted from the registry.

Raised Exceptions:

- MultipleApplicationException

`getTypeDefinition(sessionTrack : SessionTrack, ctx : InformationTypeContext) : InformationTypeContext`

Gets the type definition for the specified information type.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- ctx - The Information Type Context that contains the parameters used to locate the information type definition of interest. These may include the type name identifier, possible hierarchical constraints, or others.

`listInformationTypeIdentifiers(sessionTrack : SessionTrack) : String[]`

Returns a list of IDs for all of the registered information types.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

`updateTypeDefinition(sessionTrack : SessionTrack, typeCtx : InformationTypeContext) : void`

Updates an information type's associated context. The Phoenix Design Team suggests that this method have policy applied such that the information type name, metadata schema, and payload schema variables not be allowed to be modified.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- typeCtx - An InformationTypeContext containing the attributes (and their associated values) to be updated.

Raised Exceptions:

- ApplicationException

3.2.1.3.2 Typical Use

This service is for defining and managing information types. It may or may not utilize some kind of repository to store these definitions. Reading and writing from and to this repository are internal processes of this service and not exposed to external actors by the service interface, except in abstract forms (i.e., via methods such as "getTypeDefinition()" and "createInformationTypes()" respectively).

3.2.1.3.3 Associated Diagrams

3.2.1.3.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0007 Information Type Management

3.2.1.3.3.2 Activity Diagrams

- AD 0015 Information Type Management

3.2.1.3.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0009 Information Type Management

3.2.1.3.3.4 Sequence Diagrams

- (none)

3.2.1.4 InformationTypeManagementServiceConnector

This interface represents a connector for the information type management service.

3.2.1.5 InformationTypeManagementServiceStub

This interface represents a stub for the information type management service.

3.2.1.6 Schema

This class represents a usable version of a schema that defines either a metadata or a payload format for an information type.

3.2.1.6.1 Public Operations

`getDefinitionDocument() : Object`

Retrieves the definition document as an object.

`setDefinitionDocument(schemaDoc : Object) : void`

Sets the definition document using a generic Object representation.

Arguments:

- schemaDoc - The definition document as a generic Object.

`validate(data : Object) : boolean`

Validates an instance of data against the definition document.

Arguments:

- data - The data to be validated.

Raised Exceptions:

- ValidationFailedException

3.2.2 Submission

The submission group contains the interfaces that provide the information submission capability for the Phoenix IM Services. The submission interfaces are:

- SubmissionService (3.2.2.1)
- SubmissionServiceConnector (3.2.2.2)
- SubmissionServiceContext (3.2.2.3)
- SubmissionServiceStub (3.2.2.4)

3.2.2.1 SubmissionService

The Submission Service accepts data from producers and, if necessary, converts it into the Phoenix architecture's supported format for managed information. In addition, the Submission Service forwards the information to other IM services as required. This process uses Data Channels internal to the Submission Service and, because of this, there is no 'submit' method defined in the service's control interface.

The submission service supports the notion of acknowledging acceptance or rejection of submitted data using delivery receipt Events sent over Event Channels. The architecture defines an information "submission" service versus information "publication" service because this service does not guarantee the publication of submitted data (i.e., security and QoS policy constraints). As such, this service may represent a likely Policy Enforcement Point (PEP).

The submission service must provide mechanisms to forward submitted information to information brokering services and repository services, but the architecture makes no guarantees that this is done for any specific piece of submitted data. An implementation of the Submission Service may also provide mechanisms for forwarding submitted information to other IM services as well depending upon the requirements of said implementation.

3.2.2.1.1 Public Operations

(none)

3.2.2.1.2 Typical Use

This service is orchestrated with one or more Information Brokering Services and Dissemination Services via some internal settings that are left to the implementation to define. Engineering information such that it may contain a Context allows for the possibility that each instance of information could be tagged to be handled by the Submission Service differently, i.e., forwarded to an orchestrated set of IM services unique to that instance of information.

As such, the Context containing this service's state may contain the Service Bindings or Data Channels for the services with which this service has been integrated within the publish and subscribe orchestration chain.

3.2.2.1.3 Associated Diagrams

3.2.2.1.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0001 Information Submission

3.2.2.1.3.2 Activity Diagrams

- AD 0001 Information Submission
- AD 0002 Information Submission (Producer-to-Consumer)

3.2.2.1.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0001 Information Submission

3.2.2.1.3.4 Sequence Diagrams

- SQD 0001 Information Submission (Delivery Confirmation – via EDC)
- SQD 0002 Information Submission (Delivery Confirmation – via ENS)
- SQD 0010 Data Channels (Subscription – End-to-End Data Flow)

3.2.2.2 SubmissionServiceConnector

This interface represents a connector for the submission service.

3.2.2.3 SubmissionServiceContext

The Submission Service Context holds the default values for the Submission Service's brokering and persistence modes. Brokering mode is defined as whether or not the Submission Service will automatically forward submitted Information to the Information Broker, unless otherwise specified by an actor. The persistence mode is the flag denoting whether to forward submitted Information to one or more Repository Services, unless otherwise specified by an actor. The Submission Service Context will minimally support two attributes: *defaultPersistenceMode* and *defaultBrokeringMode*.

1. The *defaultPersistenceMode* is the default behavior with regard to any connected Repository services. Any Information that is submitted to the associated Submission Service and is not tagged with a persistence flag will default to this behavior.
2. The *defaultBrokeringMode* is the default behavior with regard to any connected Information Brokering services. Any Information that is submitted to the associated Submission Service that is not tagged with a brokering flag will default to the specified behavior.

3.2.2.3.1 Public Operations

`getDefaultBrokeringMode() : int`

Retrieves the flag describing the default brokering mode for this Submission Service. This flag tells the Submission Service if, when, and possibly how to forward submitted information to one or more connected Information Brokers.

`getDefaultPersistenceMode() : int`

Retrieves the flag describing the default persistence mode for this Submission Service. This flag tells the Submission Service if, when, and possibly how to forward submitted information to one or more connected Repository Services.

`setDefaultBrokeringMode(mode : int) : void`

Sets the default brokering mode flag.

Arguments:

- mode - The default brokering mode.

`setDefaultPersistenceMode(mode : int) : void`

Sets the default persistence mode flag.

Arguments:

- mode - The default persistence mode.

3.2.2.4 SubmissionServiceStub

This interface represents a stub for the submission service.

3.2.3 Information Brokering

The information brokering group contains the interfaces that define and support the information brokering capability of the Phoenix architecture. The information brokering interfaces are:

- ConsumerList (3.2.3.1)
- ConsumerReport (3.2.3.2)
- InformationBrokeringContext (3.2.3.3)
- InformationBrokeringService (3.2.3.4)
- InformationBrokeringServiceConnector (3.2.3.5)
- InformationBrokeringServiceContext (3.2.3.6)
- InformationBrokeringServiceStub (3.2.3.7)

3.2.3.1 ConsumerList

Contains the list of in-band and out-of-band consumers for a subscription.

3.2.3.1.1 Public Operations

`addInBandConsumer(consumer : String) : void`

Adds an in-band consumer to the in-band list.

Arguments:

- consumer - The URI or subscription ID of the in-band consumer being added.

`addOutOfBandConsumer(consumer : URI) : void`

Adds an out-of-band consumer to the out-of-band list.

Arguments:

- consumer - The URI of the out-of-band consumer to be added.

`getInBandConsumers() : String[]`

Retrieves the listing of in-band consumers. Returns a list of subscription IDs.

`getOutOfBandConsumers() : URI[]`

This method retrieves the listing of URI's for the out-of-band consumers.

`setInBandConsumers(inBandConsumers : String[]) : void`

Sets the list of in-band consumers.

Arguments:

- inBandConsumers - The in-band consumers list.

`setOutOfBandConsumers(outOfBandConsumers : URI[]) : void`

Sets the out-of-band consumers list.

Arguments:

- outOfBandConsumers - The out-of-band consumers.

3.2.3.2 ConsumerReport

This enumeration lists the possible types of consumer lists that can be understood by the Phoenix IM Services.

3.2.3.2.1 Public Fields

`IN_BAND_ONLY`

Identifies a list that contains only in-band consumers. The term "in-band" applies to consumers who are using the Phoenix Data Channels to receive information from the IM Services.

`OUT_OF_BAND_ONLY`

Identifies a list that contains only out-of-band consumers. The term "out-of-band" applies to consumers who are not using the Phoenix Data Channels to receive information from the IM Services. These consumers handle their own data reception needs.

`ALL`

Identifies a list that contains both in-band and out-of-band consumers.

3.2.3.3 InformationBrokeringContext

This class is used specifically for registering predicates with the Information Brokering Service as subscriptions. Information brokering is the near real-time process of filtering the information that specific consumers are interested in from a larger set of information as the information is made available to the IM Services.

3.2.3.3.1 Public Operations

(None)

3.2.3.4 InformationBrokeringService

Information brokering is the act of matching submitted information with registered predicates. An information brokering service must support both the ability to forward brokered information for delivery and the ability to report the list of matching predicates for a piece of information without delivery.

The architecture defines the information brokering service in such a way that it supports three distinct brokering use cases:

1. Brokering on-demand via a control method. This method accepts a single instance of information, brokers it, and returns the consumer hit list of predicate identifiers and/or consumer URIs that the information satisfied.
2. Implicit brokering that results in a stream of hit list results being delivered to interested entities via event notification.
3. Implicit brokering that results in a stream of information that is forwarded to some dissemination service(s) for delivery to matching consumers.

Implicit brokering refers to the act of the information broker receiving information via data channels. The brokering service interface does not limit the operations that may be performed upon submitted information during the brokering process.

The process of information brokering is realized through the use of the Evaluator interface defined by the Phoenix architecture. Evaluator instances are created by an EvaluatorFactory. This factory accepts some description of the type of predicate (XPath, etc.) submitted to the broker and creates a new Evaluator instance for that predicate type, if the broker supports said predicate type. This design allows for dynamic plug-ability for the predicate evaluation logic and allows this logic to be maintained by the brokering service itself or another piece of the Phoenix SOA, such as the Information Type Management Service.

3.2.3.4.1 Public Operations

`dropPredicates(sessionTrack : SessionTrack, infoBrokeringCtxIds : String[ ]) : void`

This method is used to drop registered predicates for subscriptions. This will remove the internal brokering context for each identified predicate. Ths method returns no value.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- infoBrokeringCtxIds - The identifiers of the registered Predicate Contexts (predicates) to be dropped.

Raised Exceptions:

- MultipleApplicationException

`getBrokeringContext(sessionTrack : SessionTrack, brokeringId : String) : InformationBrokeringContext`

This method provides a way to access the brokering context object describing a registered predicate. It returns an InformationBrokeringContext that describes the registered predicate. It contains the predicate criteria as well as any other custom

attributes that the registrant utilized to describe what information they are interested in receiving.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- brokeringId - The identifier of the registered Predicate Context (predicate) whose description is going to be retrieved.

```
getConsumers(sessionTrack : SessionTrack, information : Information,
consumerScope : ConsumerReport) : ConsumerList
```

This method is used to immediately receive the results of a brokering operation over the supplied information. It returns a ConsumerList object that contains the URI information and identifiers for the consumers whose registered predicate matches the supplied information.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- information - The information that is to be brokered against the registered predicates.
- consumerScope - The flag that identifies the types of consumers that the invoker is interested in. These are identified by the ConsumerReport enumeration.

Raised Exceptions:

- ApplicationException

```
public registerForConsumerLists(sessionTrack : SessionTrack, predicateCtx :
InformationBrokeringContext) : String
```

This method registers a brokering context (predicate) for an actor that wishes to receive consumer hit lists for information that matches the supplied predicate. A Destination Context is contained within the provided Predicate Context and describes the information necessary to deliver the consumer hit lists of interest to the actor. It returns the identifier for the registered request for consumer lists.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- predicateCtx - The Context object that contains the predicate that the invoker of this method is interested in finding matching consumers for.

Raised Exceptions:

- MultipleApplicationException

```
public registerPredicates(sessionTrack : SessionTrack, predicateCtxs :
InformationBrokeringContext) : String [ ]
```

This method is used by in-band subscribers who wish to receive information as a result of the brokering operation defined by the supplied predicate. It returns the identifiers for the registered predicates.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- predicateCtxs - The Context objects describing the predicates to be registered.

Raised Exceptions:

- MultipleApplicationException

3.2.3.4.2 Typical Use

This service is orchestrated with one or more Submission Services and Dissemination Services via some internal settings that are left to the implementation to define. Engineering information such that it may contain a Context allows for the possibility that each instance of information could be tagged to be handled by the Information Brokering Service differently, i.e., forwarded to an orchestrated set of IM services unique to that instance of information.

As such, the Context containing this service's state may contain the Service Bindings or Data Channels for the Submission and Dissemination service or services with which this service has been integrated within the publish and subscribe orchestration chain.

3.2.3.4.3 Associated Diagrams

3.2.3.4.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities

3.2.3.4.3.2 Activity Diagrams

- AD 0006 Brokering (Information via Control Interface)
- AD 0007 Brokering (Information via Data Channel (Out-of-Band Delivery)
- AD 0008 Brokering (Information via Data Channel)

3.2.3.4.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0003 Brokering (Information via Control Interface)
- CD 0004 Brokering (Information via Data Channel)

3.2.3.4.3.4 Sequence Diagrams

- SQD 0003 Brokering (Information via Control Interface)
- SQD 0004 Brokering (Information via Data Channel)
- SQD 0006 Information Consumption (Subscription Creation & Destruction)
- SQD 0007 Information Consumption (Subscription Creation & Destruction2)
- SQD 0010 Data Channels (Subscription End-to-End Data Flow)

3.2.3.5 InformationBrokeringServiceConnector

This interface represents a connector for the information brokering service.

3.2.3.6 InformationBrokeringServiceContext

This context contains the listing of supported brokering languages as well as any other Information Brokering Service specific attributes. The Information Brokering Service Context contains a single required attribute: *supportedPredicateTypes*.

The *supportedPredicateTypes* attribute is a list of brokering languages, such as XPath, that are supported by the associated Information Brokering service.

3.2.3.6.1 Public Operations

`addSupportedPredicateType(type : String) : void`

Adds a supported predicate type.

Arguments:

- type - The identifier for the predicate type.

`getSupportedPredicateTypes() : List<String>`

Retrieves the list of predicate types that are currently supported by the associated Information Brokering Service.

`removeSupportedPredicateType(type : String) : void`

Removes a supported predicate type.

Arguments:

- type - The identifier for the supported predicate type.

3.2.3.7 InformationBrokeringServiceStub

This interface represents a stub for the information brokering service.

3.2.4 Dissemination

The dissemination group contains the interfaces that define and support the information dissemination capability of the Phoenix architecture. The dissemination interfaces are:

- DisseminationContext (3.2.4.1)
- DisseminationService (3.2.4.2)
- DisseminationServiceConnector (3.2.4.3)
- DisseminationServiceContext (3.2.4.4)
- DisseminationServiceStub (3.2.4.5)

3.2.4.1 DisseminationContext

This context is used internally by the DisseminationService to store the information describing a consumer's location within the network. It contains the DataTransport that has been negotiated between the consumer and the Dissemination Service as well as the Destination Context that defines where to locate the consumer and the subscription identifier that this consumer and specific Data Transport are associated with. The attribute names are: consumerTransport, consumerDestCtx, and subscriptionId.

1. The consumerTransport is the actual Data Transport object that contains the DataChannel(s) that are appropriate to use when communicating with the consumer.
2. The consumerDestCtx is the DestinationContext describing the physical location of the consumer within the network.
3. The subscriptionId is the identifier of the subscription that this context is associated with (transport and destination). This ID is stamped onto every piece of Information before it is transmitted over the Data Channel(s) so that the consumer can determine which subscription the Information should be associated with.

3.2.4.1.1 Public Operations

`getConsumerDestinationContext() : DestinationContext`

Returns the context that identifies the physical end point location of the consumer.

`getConsumerTransport() : DataTransport`

Returns the DataTransport object that is physically connected with the associated consumer.

`getSubscriptionId() : String`

Returns the identifier for the subscription associated with this DisseminationContext.

`setConsumerDestinationContext(ctx : DestinationContext) : void`

Sets the DestinationContext that specifies where the consumer can be found.

Arguments:

- ctx - The DestinationContext that defines the location of the consumer.

`setConsumerTransport(transport : DataTransport ) : void`

Sets the DataTransport that will be used to communicate with the consumer.

Arguments:

- transport - The DataTransport that will be used to communicate with the consumer.

`setSubscriptionId(id : String) : void`

Sets the associated subscription identifier.

Arguments:

- id - The associated subscription identifier.

3.2.4.2 DisseminationService

The Dissemination Service is responsible for taking information and delivering it to registered consumers. No method calls exist for disseminating managed information. The service receives information via Data Channels and delivers the information using its internal implementation code that maps the information instances to separate outgoing Data Channels. In addition, this service may represent a likely Policy Enforcement Point (PEP). The Dissemination Service is responsible for delivering the correct information to the correct consumers in a timely manner. The information to be delivered by this service must be tagged with identifiers that indicate what consumers it should be delivered to. Within the Phoenix architecture, these identifiers are URIs describing the end points or locations of the consumers. Tagging the information to be disseminated with consumer locations instead of another form of consumer identifier allows this service to be utilized as a standalone service.

3.2.4.2.1 Public Operations

`dropConsumers(sessionTrack : SessionTrack, subscriptionIds : String[]) : void`

Removes the specified consumers from the list of registered consumers.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- subscriptionIds - The identifiers of the subscriptions to be dropped.

Raised Exceptions:

- MultipleApplicationException

`getConsumer(subscriptionId : String, sessionTrack : SessionTrack) : DisseminationContext`

Gets the data that identifies a specific consumer from the list of registered consumers. This method returns the DisseminationContext object that contains the URI information for the consumers of the specified subscription.

Arguments:

- subscriptionId - Retrieve the DisseminationContext that describes the consumer with this identifier.
- sessionTrack - The pedigree of the invokers for this method.

`getConsumers(sessionTrack : SessionTrack) : List`

Gets the data that identifies the complete set of registered consumers. This method returns a List of DisseminationContext objects that describe the consumers for all registered subscriptions.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

```
 registerConsumers(sessionTrack : SessionTrack, predicateCtxs :
InformationBrokeringContext[]) : String[]
```

Registers consumers to receive information. This method returns the identifier for a registered consumer.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- predicateCtxs - The array of InformationBrokeringContexts that describe the consumers to be registered.

Raised Exceptions:

- MultipleApplicationException

3.2.4.2.2 Typical Use

This service is typically paired with a Phoenix Submission Service and an Information Brokering Service to provide a complete publish and subscribe capability. This service may also be used on its own to deliver data to consumers. The underlying implementation technology is decided upon by the developers.

This service is orchestrated with one or more Information Brokering Services and Submission Services via some internal settings that are left to the implementation to define. Engineering information such that it may contain a Context allows for the possibility that each instance of information could be tagged to be handled by the Dissemination Service differently, i.e., forwarded to an orchestrated set of IM services unique to that instance of information.

As such, the Context containing this service's state may contain the Service Bindings or Data Channels for the services with which this service has been integrated within the publish and subscribe orchestration chain.

3.2.4.2.3 Associated Diagrams

3.2.4.2.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities

3.2.4.2.3.2 Activity Diagrams

- AD 0010 Information Dissemination

3.2.4.2.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0006 Information Dissemination

3.2.4.2.3.4 Sequence Diagrams

- SQD 0006 Information Consumption (Subscription Creation & Destruction)
- SQD 0007 Information Consumption (Subscription Creation & Destruction2)
- SQD 0010 Data Channels (Subscription End-to-End Data Flow)

3.2.4.3 DisseminationServiceConnector

This interface represents a connector for the information dissemination service.

3.2.4.4 DisseminationServiceContext

The Context for the Dissemination Service holds the list of registered consumers and its metadata. This Context contains at least the following variables: *registeredConsumers*, *maxNumberOfConsumers*, and *numberOfRegisteredConsumers*.

1. The list of *registeredConsumers* contains the Destination Contexts for all consumers who have been registered with the associated Dissemination Service.
2. The *maxNumberOfConsumers* is the maximum number of supported consumers.
3. The *numberOfRegisteredConsumers* is the current number of consumers who have registered with the associated Dissemination Service.

3.2.4.4.1 Public Operations

`addRegisteredConsumer(ctx : DisseminationContext) : void`

Adds a registered consumer's information to the service context.

Arguments:

- ctx - The DisseminationContext containing the details of how to interact with the consumer.

`getMaxSupportedConsumers() : long`

Retrieves the theoretical maximum number of concurrent consumers that the associated parent service can support.

`getNumberofRegisteredConsumers() : long`

Retrieves the current number of registered consumers.

`getRegisteredConsumers() : Map<String, DisseminationContext>`

Retrieves the list of currently registered consumers.

`removeRegisteredConsumer(consumerId : String) : void`

Removes the identified consumer's information from the list of registered consumers.

Arguments:

- consumerId - The identifier for the consumer to be removed.

`setMaxSupportedConsumers(max : long) : void`

Sets the maximum number of concurrently supported consumers for this service.

Arguments:

- max - The maximum number of concurrently supported consumers for this service.

3.2.4.5 DisseminationServiceStub

This interface represents a stub for the information dissemination service.

3.2.5 Subscription

The subscription group contains the helper interfaces that define and support the subscription helper service. This service provides an easy to use front-end for the information brokering service and the dissemination service. The subscription interfaces are:

- SubscriptionService (3.2.5.1)
- SubscriptionServiceConnector (3.2.5.2)
- SubscriptionServiceContext (3.2.5.3)
- SubscriptionServiceStub (3.2.5.4)

3.2.5.1 SubscriptionService

This interface acts as a helper for the Information Brokering Service, Dissemination Service, and Event Notification Service interfaces in an attempt to make registering subscriptions with the IM services a little more convenient.

Instead of actors needing to interact with these services individually, they can instead interact solely with the subscription service to set up and tear down information subscriptions. This service simply acts as a pass through for control method invocations upon the aforementioned services. The subscription service may also act as a Policy Enforcement Point (PEP) and/or a "crumple zone" for these services at implementation time.

3.2.5.1.1 Public Operations

`dropSubscriptions(sessionTrack : SessionTrack, subscriptionIds : String []) : void`

Drops the registered subscriptions for the identified subscribers.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- subscriptionIds - The identifiers for the subscriptions to be removed from the list of registered subscriptions.

Raised Exceptions:

- MultipleApplicationException

`getSubscriptionContext(sessionTrack : SessionTrack, subscriptionId : String) : InformationBrokeringContext`

Retrieves the context object that describes the subscription specified by the given identifier.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- subscriptionId - The subscription identifier associated with the Information Brokering Context that is going to be retrieved.

`registerForConsumerLists(sessionTrack : SessionTrack, predicateCtxs : InformationBrokeringContext []) : String`

Registers to receive asynchronous updates concerning what consumers match the supplied predicate.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- predicateCtxs - The Information Brokering Contexts describing what consumers the registrant is interested in.

Raised Exceptions:

- MultipleApplicationException

```
registerSubscriptions(sessionTrack : SessionTrack, predicateCtxs :
InformationBrokeringContext[]) : String
```

Registers a set of predicates with the Information Brokering Service as new subscriptions and returns the identifiers of the registered subscriptions.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- predicateCtxs - The Information Brokering Contexts describing the predicates to be registered as new subscriptions.

Raised Exceptions:

- MultipleApplicationException

3.2.5.1.2 Typical Use

The Subscription Service is used as a front-end service for the Information Brokering and Dissemination Services. It provides a more user-friendly interface for creating and destroying subscriptions to information that those services. For example, to register a subscription without using the Subscription Service, an actor must register its predicate with the Information Brokering Service and then may (depending upon the implementation design) have to register itself as a consumer with the Dissemination Service. When using the Subscription Service this same actor would only have to call the "registerSubscriptions" method, thus reducing the number of actor-to-service interactions and overall complexity of operations on the actor side.

3.2.5.1.3 Associated Diagrams

3.2.5.1.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0008 Information Consumption (Subscription)

3.2.5.1.3.2 Activity Diagrams

- AD 0016 Information Consumption (Subscription)

3.2.5.1.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0010 Information Consumption (Subscription)

3.2.5.1.3.4 Sequence Diagrams

- SQD 0006 Information Consumption (Subscription – Creation & Destruction)
- SQD 0007 Information Consumption (Subscription – Creation & Destruction2)

3.2.5.2 SubscriptionServiceConnector

This interface represents a connector for the subscription service.

3.2.5.3 SubscriptionServiceContext

This is the context that contains the attributes specific to the Subscription Service. These attributes are an aggregation of those found within the Information Brokering Service and Dissemination Service Contexts. This structure supports the use of the SubscriptionService as a helper service that hides required interactions with the Information Brokering and Dissemination Services. This service probably does not store the list of supported predicate types locally. Instead it should probably pass the request for the supported predicate types list on to the known Information Brokering Service(s).

3.2.5.3.1 Public Operations

`getSupportedPredicateTypes() : List`

Retrieves the list of brokering languages that are currently supported by the connected Information Broker(s).

3.2.5.4 SubscriptionServiceStub

This interface represents a stub for the subscription service.

3.2.6 Query

The query group contains the interfaces that provide the information retrieval capability of the Phoenix IM Services. The query interfaces are:

- DataStoreType (3.2.6.1)
- InformationQueryContext (3.2.6.2)
- QueryService (3.2.6.3)
- QueryServiceConnector (3.2.6.4)
- QueryServiceContext (3.2.6.5)
- QueryServiceStub (3.2.6.6)

3.2.6.1 DataStoreType

This enumeration lists the possible types of data stores that can be connected to a Phoenix Repository Service or Query Service.

3.2.6.1.1 Public Fields

`ARCHIVE`

Identifies a data store as a relatively higher-latency, higher-capacity data store. These data stores are synonymous with traditional database archives. These archives are typically not as readily available as LIVE data stores.

`LIVE`

Identifies a data store as a relatively lower-latency, lower-capacity data store. These data stores, called repositories by the Phoenix architecture, are typically more readily available than ARCHIVE data stores.

3.2.6.2 InformationQueryContext

A QueryContext used specifically for information query operations.

3.2.6.2.1 Public Operations

(None)

3.2.6.3 QueryService

Information retrieval is provided by the query service interface. This service permits actors to retrieve records from the underlying data store. Using a Query Context construct to describe the actual query to be executed allows the architecture to mandate a small set of required query attributes while allowing individual implementations of the IM Services to include additional attributes to tune the query processing of each query service more towards their respective underlying data stores. The query service will support synchronous and asynchronous query execution. For synchronous queries the execute query method provided will return a value representing the number of matching records found. This same method will return nothing when used asynchronously. In all cases the result set of the query will be returned to the consumer via Data Channels.

3.2.6.3.1 Public Operations

`executeQuery(sessionTrack : SessionTrack, ctx : QueryContext) : int`

This method processes the specified query to satisfy some inquisitor's request for information. Actual result sets are delivered via a Data Channel set up between the consumer and the query service. When this method is invoked synchronously, the return value signals to the inquisitor the number of matching records that were found or that an error occurred while processing their query. A value of zero or greater is the number of matching records. Values less than zero are reserved for possible error flags. When used asynchronously, this method does not return a value.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- ctx - The Query Context object that describes what information the inquisitor is searchig for.

Raised Exceptions:

- ApplicationException

`getCounts(sessionTrack : SessionTrack, infoTypeNames : List) : Map`

Retrieves the number of records in the repository for the specified types. This method returns a Map of key-value pairs that define how many records there are for each specified type.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- infoTypeNames - The listing of information type identifiers to retrieve the count(s) for.

3.2.6.3.2 Typical Use

This service is typically used as a SOA-based presence for legacy and non-service-oriented databases and other data stores.

3.2.6.3.3 Associated Diagrams

3.2.6.3.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0009 Information Retrieval (Query)

3.2.6.3.3.2 Activity Diagrams

- AD 0017 Information Retrieval (Query – Asynchronous)
- AD 0018 Information Retrieval (Query – Synchronous)

3.2.6.3.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0011 Information Retrieval (Query)

3.2.6.3.3.4 Sequence Diagrams

- SQD 0008 Information Retrieval (Query – Asynchronous)
- SQD 0009 Information Retrieval (Query – Synchronous)

3.2.6.4 QueryServiceConnector

This interface represents a connector for the information retrieval (query) service.

3.2.6.5 QueryServiceContext

This context describes the attributes specific to the Query Service which includes any default query settings such as timeouts and time to live, the set of query languages supported by the associated Query Service, and the type of data store the associated Query Service is an interface to (i.e., "Is the underlying data store a repository or an archive?"). The Query Service Context has five attributes defined for it by the Phoenix architecture: *dataStoreType*, *defaultMaxresultSetSize*, *supportedPredicateTypes*, *defaultTurnAroundTime*, and *defaultTimeToLive*.

1. The *dataStoreType* is the kind of data store that the underlying data store represents. The possible values for this flag are defined by the Data Store Type enumeration.
2. The *defaultMaxresultSetSize* is the default setting for the maximum number of results to be returned by any single query against the underlying data store.
3. The *supportedPredicateTypes* is a list of the supported query languages, such as XQuery, that the data store supports.
4. The *defaultTurnAroundTime* is the default amount of time that a query has to execute, build its result set, and deliver all results to the consumer(s).
5. The *defaultTimeToLive* is the default amount of time that a query has to execute and build its result set. This is a separate constraint on queries because these operations are typically the most intensive and can cause the most problems at runtime.

3.2.6.5.1 Public Operations

`addSupportedPredicateType(type : String) : void`

Adds a supported predicate type.

Arguments:

- type - The supported predicate type identifier.

`getDataStoreType() : DataStoreType`

Retrieves the value describing the type of data store that the associated parent Query Service is connected to. The range of possible return values are defined in the DataStoreType enumeration.

`getDefaultMaxResultSetSize() : long`

Retrieves the default value for the maximum number of results that the IM Services are willing, or allowed, to return to each individual inquisitor for each individual query.

`getDefaultTimeToLive() : long`

This identifies the amount of time that the inquisitor is willing or able to listen for a response to its query request. This response may be the return of a result set size (synchronous operation) or the start of the result set stream over the corresponding data channel(s) (asynchronous operation).

`getDefaultTurnAroundTime() : long`

This represents the maximum amount of time that the inquisitor is willing or able to listen for the complete result set of the query that was submitted for execution. This may be implemented such that it contains the time required to execute the query or not.

`getSupportedPredicateTypes() : List`

Retrieves the list of brokering languages that are currently supported by the associated Query Service and its underlying data store.

`setDataStoreType(type : DataStoreType) : void`

Sets the type of data store to which the associated query service is connected.

Arguments:

- type - The type of data store that the associated query service is connected to. P ossible values are defined by the DataStoreType enumeration.

`setDefaultMaxResultSetSize(maxSize : long) : void`

Sets the default maximum result set size.

Arguments:

- maxSize - The default maximum result set size.

`setDefaultTimeToLive(ttl : long) : void`

Sets the default amount of time that the inquisitor is willing or able to listen for a response to its query request. This response may be the return of a result set size (synchronous operation) or the start of the result set stream over the corresponding data channel(s) (asynchronous operation).

Arguments:

- ttl - The default amount of time that the inquisitor is willing or able to listen for a response to its query request. This response may be the return of a result set

size (synchronous operation) or the start of the result set stream over the corresponding data channel(s) (asynchronous operation).

`setDefaultTurnAroundTime(tat : long) : void`

Sets the default maximum amount of time that the inquisitor is willing or able to listen for the complete result set of the query they submitted for execution. This may be implemented such that it contains the time required to execute the query or not.

Arguments:

- tat - The default maximum amount of time that the inquisitor is willing or able to listen for the complete result set of the query they submitted for execution. This may be implemented such that it contains the time required to execute the query or not.

`removeSupportedPredicateType(type : String) : void`

Removes the given predicate type from the list of supported predicate types.

Arguments:

- type - The identifier of the predicate type that will no longer be supported.

3.2.6.6 QueryServiceStub

This interface represents a stub for the information retrieval (query) service.

3.2.7 Repository

The repository group contains the interfaces that define the Phoenix IM Services capability to store information within a repository or archive for later retrieval. The repository interfaces are:

- RepositoryService (3.2.7.1)
- RepositoryServiceConnector (3.2.7.2)
- RepositoryServiceContext (3.2.7.3)
- RepositoryServiceStub (3.2.7.4)
- TableType (3.2.7.5)

3.2.7.1 RepositoryService

The Repository Service inserts information into its associated data store(s). There is no actual insert information method defined as part of the service API. Instead, the Repository Service receives information via Data Channels, which it reads from internally, making insertion an internal process. This decision was made to ensure the physical separation of control versus data interactions. The information storage interface is an extension of the information retrieval interface. This follows the assumption that if you can write to a section of disk then you are implicitly able to read from that section as well, i.e., if you can write to the data store, you should be implicitly able to read from the data store as well. This service also provides the ability to delete records from the database.

The Phoenix architecture defines two types of data stores: repositories and archives. Repositories are low-latency high-access data stores that should support higher data read and write rates. Archives are expected to be higher latency, low access data stores that may not be able to support high data rates but can store much more data than repositories. A possible implementation strategy would be to store recent information in a repository while aging data would be moved to an archive.

3.2.7.1.1 Public Operations

`archiveRecordsByQuery(sessionTrack : SessionTrack, queries : QueryContext[]) : void`

This method initiates a pull from the underlying data store and writes the retrieved information to the specified archive. It assumes that the Repository Service will or already has set up a Data Channel with the application that is going to write to the offline repository. This method returns no value.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- queries - The Context objects that specify what information is to be moved and the locations of the archives to move it to.

Raised Exceptions:

- MultipleApplicationException

`archiveRecordsByType(sessionTrack : SessionTrack, infoTypeNames : String []) : void`

This method archives records for the specified information type identifiers. It returns no value.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- infoTypeNames - The information type identifiers for the records to be archived.

Raised Exceptions:

- MultipleApplicationException

`deleteRecordsByQuery(sessionTrack : SessionTrack, queries : QueryContext []) : void`

Deletes the specified records from the underlying data store. This method returns no value.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- queries - The Query Context objects that describe what records are to be deleted.

Raised Exceptions:

- MultipleApplicationException

```
deleteRecordsByType(sessionTrack : SessionTrack, infoTypeNames : String []) :
void
```

Removes all existing records for a specified information type that reside within the underlying data store. Returns no value.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- infoTypeNames - The information type identifiers for the records to be deleted.

Raised Exceptions:

- MultipleApplicationException

3.2.7.1.2 Typical Use

This service is typically paired with the Phoenix Submission Service. It inherits the ability to read stored information from the underlying data store from the Phoenix Query Service interface. This service may be implemented in such a way that it can be used as a wrapper for existing legacy data stores.

3.2.7.1.3 Associated Diagrams

3.2.7.1.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0005 Information Storage (Persistence)

3.2.7.1.3.2 Activity Diagrams

- AD 0011 Information Storage (Persistence)

3.2.7.1.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0007 Information Storage (Persistence)

3.2.7.1.3.4 Sequence Diagrams

- SQD 0005 Information Storage (Persistence)

3.2.7.2 RepositoryServiceConnector

This interface represents a connector for the information storage (repository) service.

3.2.7.3 RepositoryServiceContext

This is the context used to describe the attributes specific to the Repository Service. This context inherits all attributes defined by the Query Service Context, just like the Repository Service interface inherits the methods found in the Query Service interface. The Repository Service Context contains at least three attributes: maxSize, spaceRemaining, and defaultTableType.

1. The maxSize is the maximum size of the complete data store including all tables for all supported Information types. The unit of measure for this variable is left up to the implementations of the abstract architecture.
2. The spaceRemaining is the amount of space remaining for the complete data store. Again, the unit of measure for this variable is left up to the implementers of the abstract architecture.
3. The defaultTableType defines the default type of table for registered Information types. The possible values for this flag are defined by the Table Type enumeration.

3.2.7.3.1 Operations

`getMaxRepositorySize() : long`

Retrieves the theoretical maximum size for the data store that the associated parent Repository Service is connected to. The actual unit of measure is left to the implementation designers to determine.

`getSpaceRemaining() : long`

Retrieves the actual space remaining on the hard drive(s) that the underlying data store is being hosted on.

`getDefaultTableType() : TableType`

Retrieves the flag defining the default type of persistence to perform when inserting information into the underlying data store. Possible values are defined in the Phoenix.Contexts.Services.EnumTypes.TableType enumeration.

3.2.7.4 RepositoryServiceStub

This interface represents a stub for the information storage (repository) service.

3.2.7.5 TableType

This enumeration lists the possible types of tables that can exist within a data store that is connected to a Phoenix Repository Service or Query Service.

3.2.7.5.1 Public Fields

`FIXED_SIZE`

This denotes a data store that stops storing data when the corresponding table reaches a fixed size limit. This limit can be represented as physical disk space, number of records, or something else defined by the implementation designers.

`INFINITE`

The data store keeps on storing data until it suffers a hard or soft failure.

`ROLLING`

The data store keeps inserting data until a set size is reached. Once the limit has been hit some existing data is removed to make room for the new data to be stored. The data removed is determined according to whatever logic or policy that the implementation designers enact.

3.2.8 Event Notification

The event notification group contains the interfaces that define Phoenix's event notification service. The event notification interfaces are:

- EventNotificationService (3.2.8.1)
- EventNotificationServiceConnector (3.2.8.2)
- EventNotificationServiceStub (3.2.8.3)

3.2.8.1 EventNotificationService

A central event delivery capability will be provided by this service. The logic describing when to fire an event and the construction of the event will reside within other IM services, external to this service. Actors may register for delivery of events of a certain type, or by using other implementation-specific criteria. When an event is fired by an IM service, the event notification service will deliver the event to all registered entities whose criteria match the fired event. The Phoenix IM services also support direct event communications between actors via the notion of Event Channels. Direct event communication and the central event notification service have been architected in such a way that both may be supported and utilized by all Phoenix services and actors.

3.2.8.1.1 Public Operations

`notify(sessionTrack : SessionTrack, event : Event) : void`

Notifies registered requestors of the given event.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- event - The event that has been fired.

`preRegisterNotificationRequest(sessionTrack : SessionTrack, registrantIds : List, eventType : EventType, additionalConstraints : Map) : String`

This method tells the ENS that the specified registrants are going to be registering event Notification requests, either for themselves or other parties. This method returns the identifier for the pre-registered request for events.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- registrantIds - A List of identifiers for producers of events that match the constraints specified by this method. These identifiers are either service or actor identifiers.
- eventType - The type of event(s) the the registrants with to receive. Possible types are listed in the EventType enumeration.
- additionalConstraints - Any additional attributes that describe what event(s) the registrants wish to receive.

Raised Exceptions:

- ApplicationException

`registerNotificationRequest(sessionTrack : SessionTrack, recipientURIs : List, eventType : EventType, additionalConstraints : Map) : String`

Registers a request to be notified of specific events as they occur within other services. This method returns an identifier for the specific Notification Request. This ID may be used to modify or supplement the event Notification request. This method returns the identifier for the registered request for events.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- recipientURIs - A listing of URI's for actors who are to be notified of events that match the given criteria for the associated notification request.
- eventType - The type of event(s) the the registrants with to receive. Possible types are listed in the EventType enumeration.
- additionalConstraints - Any additional attributes that describe what event(s) the registrants wish to receive.

Raised Exceptions:

- ApplicationException

3.2.8.1.2 Typical Use

The notion of confirming delivery of information, or reporting the failure of delivery, is a specific Use Case associated with the event notification concept supported by this architecture. There have been four specific instances of delivery receipts identified by the design team:

1. Submission Service Acknowledgement (SACK) – This case covers the act of the Submission Service signaling the producer that it received a specific instance of information.
2. Submission Service Negative Acknowledgement (SNACK) – This case covers the act of the Submission Service signaling the producer that it attempted, but failed, to receive a specific instance of information.
3. Submission Service Muted Acknowledgement (SMACK) – This is the case where the Submission Service provides neither a SACK nor a SNACK to the producer. This is the default case for all submitted information.
4. Consumer Acknowledgement – This is the case where the producer wishes to be notified that the consumers of its submitted information have indeed received it.

Logical combinations of these four instances of delivery receipts follow depending on the settings of the Submission Service and the types of delivery receipt requested by the producer.

3.2.8.1.3 Associated Diagrams

3.2.8.1.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0001 Information Submission
- UC 0006 Event Notification
- UC 0008 Information Consumption (Subscription)

3.2.8.1.3.2 Activity Diagrams

- AD 0012 Event Notification
- AD 0013 Event Notification (Delivery Confirmation – Consumer)
- AD 0014 Event Notification (Delivery Confirmation – Submission Service)

3.2.8.1.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0008 Event Notification

3.2.8.1.3.4 Sequence Diagrams

- SQD 0001 Information Submission (Delivery Confirmation – via EDC)
- SQD 0002 Information Submission (Delivery Confirmation – via ENS)
- SQD 0004 Brokering (Information – via Data Channel)
- SQD 0010 Data Channels (Subscription – End-to-End Data Flow)

3.2.8.2 EventNotificationServiceConnector

This interface represents a connector for the event notification service.

3.2.8.3 EventNotificationServiceStub

This interface represents a stub for the event notification service.

3.2.9 Service Brokering

The service brokering group contains the interfaces that define and support Phoenix's service brokering service. The service brokering interfaces are:

- ServiceBrokeringQueryContext (3.2.9.1)
- ServiceBrokeringService (3.2.9.2)
- ServiceBrokeringServiceConnector (3.2.9.3)
- ServiceBrokeringServiceContext (3.2.9.4)
- ServiceBrokeringServiceStub (3.2.9.5)

3.2.9.1 ServiceBrokeringQueryContext

A QueryContext used specifically for Service Brokering operations.

3.2.9.1.1 Public Operations

(None)

3.2.9.2 ServiceBrokeringService

This service supports the registration of service descriptors and provides a means to broker over these registered descriptions in order to selectively utilize IM services as desired. These descriptions include details such as what operations the service is capable of performing and what information types the service may apply these operations to. Each service description is linked to a ServiceBinding object that contains the constructs embodying Data Channels and Control Channels for their associated service.

3.2.9.2.1 Public Operations

`brokerForServices(sessionTrack : SessionTrack, constraints : ServiceBrokeringQueryContext) : List`

Brokers for a set of services that satisfy the specified constraints. This method returns a list of matching service binding classes, which may be used to connect to their associated services via Control Channels or Data Channels.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- constraints - The defined criteria used to broker for matching services.

`registerService(sessionTrack : SessionTrack, svcCtx : SessionContext) : void`

Registers the given description for the specified service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- svcCtx - The Context object containing the description of the service to be registered.

3.2.9.2.2 Typical Use

This service is designed to be used with any SOA-based service. It depends upon the use of the Phoenix Context classes, but any class or data set may be stored within the internal map that is used to associate each service description with a specific service instance.

3.2.9.2.3 Associated Diagrams

3.2.9.2.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0004 Brokering (Service)

3.2.9.2.3.2 Activity Diagrams

- AD 0009 Brokering (Service)

3.2.9.2.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0005 Brokering (Service)

3.2.9.2.3.4 Sequence Diagrams

0013 Session Management (Creation, Use & Destruction)

3.2.9.3 ServiceBrokeringServiceConnector

This interface represents a connector for the service brokering service.

3.2.9.4 ServiceBrokeringServiceContext

This service's Context contains the listing of ServiceContext instances describing all services registered the Service Brokering Service. This is the listing that is brokered over to find a service or services for an actor wishing to utilize some subset of capabilities provided by the IM Services. The Service Broker Service Context contains a single required attribute, registeredServices, that contains a list of Service Contexts for all registered services. These Service Contexts may be modified versions of those maintained by their parent services.

3.2.9.4.1 Public Operations

`addRegisteredService(ctx : ServiceContext) : void`

Adds a service to the list of registered services.

Arguments:

- ctx - The ServiceContext that describes the service that is being registered.

`getRegisteredServices() : List`

Returns the list of currently registered ServiceContexts.

`removeRegisteredService(serviceId : String) : void`

Removes the specified service from the list of registered services.

Arguments:

- serviceId - The identifier for the service to be removed.

3.2.9.5 ServiceBrokeringServiceStub

This interface represents a stub for the service brokering service.

3.2.10 Session Management

The session management group contains the interfaces that define and support the session management service for the Phoenix architecture. The session management interfaces are:

- ActorContext (3.2.10.1)
- SessionContext (3.2.10.2)
- SessionManagementService (3.2.10.3)
- SessionManagementServiceConnector (3.2.10.4)
- SessionManagementServiceContext (3.2.10.5)
- SessionManagementServiceStub (3.2.10.6)

3.2.10.1 ActorContext

This Context is used to describe an entity that interacts with one or more Phoenix services. It may contain attributes describing security credentials, role set(s), QoS characteristics, and the set of session identifiers of the sessions associated with this actor. The Actor Context is defined as the IM Service's view of the actor and contains state information such as the lists of registered subscription and query predicates for this actor. Actor contexts are tracked by the Session Management Service and utilized by this service to create sessions.

The Actor Context has two required attributes: *sessionId* and *brokeringCtxs*.

1. The *sessionId* is the unique identifier of the session associated with this Actor Context. This is stored here to provide a seamless ability to trace back from the Actor Context to its Session Context.
2. The *brokeringCtxs* attribute is a list of Information Brokering Contexts and/or Query Contexts that describe the actor's currently registered predicates.

3.2.10.1.1 Public Operations

`addPredicateContext(ctx : PredicateContext) : void`

Adds the supplied PredicateContext to the collection of predicates registered by this actor.

Arguments:

- ctx - The PredicateContext to be added.

`getPredicateContexts() : List`

This method retrieves the actor's currently registered subscription and query predicates.

`getSessionId() : String`

This method returns the sessionId that is the unique identifier of the session associated with this Actor Context. This is stored here to provide a seamless ability to trace back from the Actor Context to its Session Context.

`removePredicateContext(predicateCtxId : String) : void`

Removes the identified PredicateContext from the list of registered predicates for this actor.

Arguments:

- predicateCtxId - The identifier for the PredicateContext to be removed.

`setSessionId(sessionId : String) : void`

Sets the session identifier for this actor.

Arguments:

- sessionId - The session identifier.

3.2.10.2 SessionContext

The Phoenix architecture identifies interactions between individual actors by creating sessions for each longer lived interactive information exchange between a subset of the IMS services and actors. Session constructs contain data about the actor for whom the session has been created. Most control methods defined by the Phoenix architecture expect a set of session identifiers to be supplied with each invocation. These describe the pedigree of the control invocation and are useful for authorization purposes. The Design Team has taken the time to clearly define what it expected to be stored within a Session construct in order to maximize the utility of this construct and minimize the potential negative impacts upon the IM Services. The contents of a session construct can be found described within the UML model.

The Session Context is used to describe an actor's Session(s) that have been registered with the Phoenix Session Management Service. It contains a copy of the actor-provided ActorContext. The Session Context is defined as the IM Service's view of a registered actor's intended usage of the services. This context should contain some kind of date-time based attribute that enables transactional updates to Sessions (thread-safe updates).

Session Contexts have three required attributes: *actorContext*, *defaultBroker*, and *lastCommitTime*.

1. The *actorContext* attribute is a copy of or pointer to the Actor Context object associated with this Session Context. This, along with the session identifier contained within the ActorContext, provides a seamless association between an actor and its session(s).
2. The *defaultBroker* is a copy of or pointer to the ServiceBinding object for the Service Broker Service to be used by the Session associated with this Session Context. This is used as the Session's broker for services, unless its value is null. In this case it is assumed that the actor for this Session Context already knows how to communicate with the services it wishes to make use of.
3. The *lastCommitTime* is a timestamp used by the Session Management Service to de-conflict updates to the Session Context. This is necessary because multiple actors using the Phoenix IM Services may try to update the Session at nearly the same time and with vastly different versions of the same Session Context. It is up to the implementation to determine which update call will be used to maintain the state of the context.

3.2.10.2.1 Public Operations

`getActorContext() : ActorContext`

Returns the ActorContext for the actor associated with this session.

`getBroker() : ServiceBinding`

Returns the ServiceBinding object for the Service Broker Service.

`getTimeLastUpdated() : long`

Returns the date-time stamp signifying when this context was last updated. This field is used to track update requests and determine whether or not the requestor currently has an up-to-date copy of the context..

`setActorContext(ctx : ActorContext) : void`

Sets the ActorContext for the actor associated with this session.

Arguments:

- ctx - The ActorContext for the actor associated with this session.

`setBroker(binding : ServiceBinding) : void`

Sets the Service Binding for the ServiceBrokeringService for this session (if applicable).

Arguments:

- binding - The ServiceBinding for the ServiceBrokeringService for this session (if applicable).

`timeStamp() : void`

Stamps this Context with the current date and time.

3.2.10.3 SessionManagementService

This service provides the necessary methods for creating and maintaining actor sessions within the Phoenix architecture. When this service is included in a deployed implementation, each and every actor must register a valid session before they are able to utilize any of the resident services. A service that is implemented to support standalone operations must provide this functionality internally (i.e., implement this interface) along with its main service functionality. A standalone service may be implemented in such a way that the service accommodates the fact that there is no notion of session maintained and that it is on its own as far as session management and validation is concerned. What a standalone service actually does in this case is up to the implementation designers, it may ignore the notion of sessions entirely or it may implement some home-grown solution for session management.

3.2.10.3.1 Public Operations

`createSession(context : ActorContext, brokerBack : boolean) : String`

Creates and maintain session state reflecting an actor's interactions with the IM services. Returns the identifier for the session that was created.

Arguments:

- context - The Context object describing the actor for whom the session will be created.
- brokerBack - Flag telling this service whether the actor has requested a ServiceBinding object for the Service Brokering Service or not.

`destroySession(sessionTrack : SessionTrack) : void`

Destroys the specified session. Returns nothing.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

`forceUpdateSession(sessionTrack : SessionTrack, sessionCtx : SessionContext) : String`

Forces an update of the session context for the identified session regardless of synchronization logic or timestamps. This method returns the, possibly updated, session identifier.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- sessionCtx - The updated Session Context object for the session.

```
getDistinctActors(sessionTrack : SessionTrack) : ActorContext[]
```

This is a listing of the distinct actor ID's for all actors that have registered Sessions with the Session Management Service. Returns an array of ActorContext objects that describe the distinct actors that have created IM session(s).

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

```
getSessionContext(sessionTrack : SessionTrack, sessionId : String) :
SessionContext
```

This method retrieves the SessionContext for the given session identifier. It returns the SessionContext object that describes the session identified by the given identifier.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- sessionId - The identifier of the session whose Session Context is going to be retrieved.

```
setSessionAttribute(sessionTrack : SessionTrack, attributeName : Object,
attributeValue : Object) : void
```

Sets a specific attribute for an identified session. Retrns nothing.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- attributeName - The name of the session attribute to be set.
- attributeValue - The new value for the specified session attribute.

```
updateSessionAttributes(sessionTrack : SessionTrack, attributeMap : Map) :
String
```

This method will update a subset of the session's attributes. Multiple update calls are synchronized by applying logic to the date-time stamp stored within the SessionContext being updated. Returns the, possibly updated, session identifier.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- attributeMap - The Map of attribute names's and their corresponding new values.

3.2.10.3.2 Typical Use

The Session Management Service provides a session management capability similar to those provided by a web server.

3.2.10.3.3 Associated Diagrams

3.2.10.3.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0011 Session Management

3.2.10.3.3.2 Activity Diagrams

- AD 0021 Session Management (Administrator)
- AD 0022 Session Management (User)

3.2.10.3.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0015 Session Management

3.2.10.3.3.4 Sequence Diagrams

- SQD 0013 Session Management (Creation, Use & Destruction)

3.2.10.4 SessionManagementServiceConnector

This interface represents a connector for the session management service.

3.2.10.5 SessionManagementServiceContext

This context contains the list of all active Sessions and the metadata describing them. It also contains the maximum number of active Sessions allowed by this service. This context is used to maintain the state of the active Sessions within the architecture at any given time. The Session Management Service Context contains four attributes: *activeSessions*, *maxNumberOfSessions*, *numberOfSessions*, and *defaultSessionTTL*.

1. The *activeSessions* attribute contains a list of all Session objects for the currently active sessions.
2. The *maxNumberOfSessions* is the maximum number of supported sessions.
3. The *numberOfSessions* is the current number of active sessions being managed by the associated Session Management Service.
4. The *defaultSessionTTL* is the amount of time, units to be determined by implementation, for sessions to stay alive after they have been created or active and then gone inactive. In other words, if a session is idle for this specified amount of time, it may be garbage collected or invalidated depending upon the implementation.

3.2.10.5.1 Public Operations

`getActiveSessions() : List`

Retrieves the list of currently active session contexts.

`getDefaultSessionTimeToLive() : long`

Retrieves the default time limit for this session to stay alive. The implementation of the architecture must determine what this value should be.

`getMaxSupportedSessions() : long`

Retrieves the maximum number of concurrent sessions that can be maintained by the associated implementation.

`getNumberOfSessions() : long`

Retrieves the number of currently active sessions being tracked by the associated implementation of the Session Management Service.

3.2.10.6 SessionManagementServiceStub

This interface represents a stub for the session management service.

3.2.11 Security

The security group contains the interfaces that define and support the Phoenix authorization service.
The security interfaces are:

- AuthorizationContext (3.2.11.1)
- AuthorizationResponse (3.2.11.2)
- AuthorizationResponseType (3.2.11.3)
- AuthorizationService (3.2.11.4)
- AuthorizationServiceConnector (3.2.11.5)
- AuthorizationServiceStub (3.2.11.6)

3.2.11.1 AuthorizationContext

This Context is used to provide the action, target, and any other information needed to submit a request to the Authorization Service. The attributes will be used to determine whether or not a specific actor is authorized to perform the specified action upon a particular target within the architecture. This Context contains two standard attributes: *action* and *target*.

1. The *action* is the operation that has been requested.
2. The *target* is what entity, component, or piece of Information that is to be operated upon.

The abstract architecture lists some example values for each of these attributes, but the values need to be determined by the implementation design team. As an example, they may be based on some security policy implementation such as KAoS.

3.2.11.1.1 Public Operations

`getAction() : Map`

This method returns the action or operation that has been requested.

3.2.11.1.1.1 Example Actions

- ARCHIVE
- BROKER
- CONNECT
- CREATE
- DESTROY
- DISSEMINATE
- PERSIST
- QUERY
- SUBMIT
- SUBSCRIBE

`getTarget() : Map`

Returns the target which is what entity, component, or piece of managed information that is to be operated upon.

3.2.11.1.1.2 Example Targets

- DATA_CHANNEL : BaseDataChannel
- INFORMATION : InformationContext
- SERVICE : BaseService
- SESSION : SessionContext

`setAction(map : Map) : void`

Sets the action that has been requested.

Arguments:

- map - The action that has been requested.

`setTarget(map : Map) : void`

Sets the target (what entity, component, or piece of managed information that is to be operated upon).

Arguments:

- map - The target (what entity, component, or piece of managed information that is to be operated upon).

3.2.11.2 AuthorizationResponse

This interface describes the result of an authorization operation by using the AuthorizationResponseType enumeration (Authorized, Not Authorized, or Indeterminate). It provides a framework for developers to utilize when they need to describe the results of the authorization attempt. There can optionally be an array of reasons for the given response.the response also describes the obligations that should be performed as well.

3.2.11.2.1 Public Operations

`getReasons() : String[]`

Retrieves the listing of reasons for why the authorization attempt resulted in this type of response.

`getResponseType() : AuthorizationResponseType`

Retrieves the response type identifier. Possible values are defined by the AuthorizationResponseType enumeration.

`setReasons(reasons : String[]) : void`

Sets the reasons.

Arguments:

- reasons - The reasons.

`setResponseType(responseType : AuthorizationResponseType) : void`

Sets the reposnse type.

Arguments:

- responseType - The type of response as defined by the AuthorizationResponseType enumeration.

3.2.11.3 AuthorizationResponseType

Identifies the possible types of Authorization Responses defined by the Phoenix architecture.

3.2.11.3.1 Public Fields

`AUTHORIZED`

The operation is authorized for the requesting actor according to the currently specified policy.

`INDETERMINATE`

The Authorization Service is unable to determine if the operation is authorized for the requesting actor.

`NOT_AUTHORIZED`

The operation is not authorized for the requesting actor based on the currently specified policy.

3.2.11.4 AuthorizationService

Actions within a SOA-based environment may be dependent upon some form of security policy or restriction. This security authorization capability should be architected such that it may be a single point of execution or a fully distributed capability that potentially uses a decentralized protocol. An authorization could be requested for any operation by any service. An operation is defined by this documentation as any action that is performed upon a set of targets by a set of actors.

Authorization is provided by a central or distributed service acting as a policy decision point for the chosen implementing security policies. This service does not provide mechanisms for the definition or capture of security policies due to the obvious differences in security policy technologies. To provide such mechanisms would couple this abstract architecture too tightly to a specific implementing policy engine. The process of authentication may be contained within an authorization check undertaken sometime during the process of creating a new session and, as such, no specific interface method has been defined for authentication operations.

3.2.11.4.1 Public Operations

`isAuthorized(sessionTrack : SessionTrack, ctx : AuthorizationContext) : AuthorizationResponse`

Checks if an action is authorized for the given action, target, and actors. Returns the AuthorizationResponse object that describes the result of the authorization check.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- ctx - The Context describing the operation being authorized.

`registerService(svcCtx : ServiceContext) : String`

Registers a service as a legitimate creator of sessions for other actors. Returns a String that is the Service Context identifier as authorized and created by this Authorization Service or it may return the pre-existing Service Context identifier as generated by some other service.

Arguments:

- svcCtx - The Context describing the service being registered as a creator of sessions.

`registerSession(svcCtxId : String, actorSessionCtx : SessionContext) : String`

Registers the given session for use during policy enforcement. This method may return the session identifier of the authorized session as stamped by the Authorization Service, or the pre-existing session identifier as generated by some other service.

Arguments:

- svcCtxId - The identifier for the service that created the given session.
- actorSessionCtx - The Context describing the session to be registered.

`unregisterService(svcCtxId : String) : void`

Removes a service from the list of registered session creators. Returns no value.

Arguments:

- svcCtxId - The identifier for the service that registered as a creator of sessions.

`unregisterSession(sessionTrack : SessionTrack) : void`

Removes the specified session from the internal list of sessions being used during policy enforcement. This method returns no value.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

3.2.11.4.2 Typical Use

This service is envisioned to be used by other services to authorize their operations before they are undertaken. Within the Phoenix IM Services, it is typically assumed that any operation upon information as well as any operations that result in the creation, updating, or destruction of entities should first be authorized.

3.2.11.4.3 Associated Diagrams

3.2.11.4.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0010 Authorization

3.2.11.4.3.2 Activity Diagrams

- AD 0019 Authorization

3.2.11.4.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0013 Authorization

3.2.11.4.3.4 Sequence Diagrams

- SQD 0013 Session Management (Creation, Use & Destruction)

3.2.11.5 AuthorizationServiceConnector

This interface represents a connector for the authorization service.

3.2.11.6 AuthorizationServiceStub

This interface represents a stub for the authorization service.

3.2.12 Client

The client group contains the interfaces that define a service designed to live within a client's address space. The client interfaces are:

- ClientRuntimeService (3.2.12.1)
- ClientRuntimeServiceConnector (3.2.12.2)
- ClientRuntimeServiceContext (3.2.12.3)
- ClientRuntimeServiceStub (3.2.12.4)

3.2.12.1 ClientRuntimeService

This class ensures that there is a service oriented presence on the client-side to support event notification and connectors for reach-back from services to the client. This allows core IM Services the ability to influence external actors' address space providing a possible location for client-side policy enforcement and updating, event notification, or other service-to-external actor interactions. This ability becomes doubly important when operating on a disadvantaged network where actor communications may phase in and out over time due to networking degradation or other operational conditions. In this environment the client runtime service may provide a network buffer at the application level by queuing outgoing data until it can be transmitted or it may provide proxy IM capabilities for the client while it is disconnected from the network.

3.2.12.1.1 Public Operations

(None)

3.2.12.1.2 Typical Use

This service is envisioned to be a policy enforcement point for Quality of Service and/or security applications. It is also possible that it could play a central role in the creation and maintenance of Event Channels and Data Channels.

3.2.12.1.3 Associated Diagrams

3.2.12.1.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0012 Client Reach-Back

3.2.12.1.3.2 Activity Diagrams

- 0023 Client Reach-Back (Local Policy Capture & Enforcement)

3.2.12.1.3.3 Class Diagrams

- CD 0000 Phoenix IM services
- CD 0012 Client Reach-Back

3.2.12.1.3.4 Sequence Diagrams

- (none)

3.2.12.2 ClientRuntimeServiceConnector

This interface represents a connector for the client runtime service.

3.2.12.3 ClientRuntimeServiceContext

The Client Runtime Service Context is used to store and track the registered subscriptions and active queries for an associated external actor, which is what is notionally called a client. The minimum required attributes for this Context are: *activeSubscriptionIds* and *activeQueryIds*.

1. The list of activeSubscriptionIds contains the set of subscription IDs for subscriptions that the associated client actor has registered with the IM Services.
2. The activeQueryIds list contains the IDs for all currently active queries that have been executed by the associated client actor.

3.2.12.3.1 Public Operations

`addActiveQuery(id : String) : void`

Adds an active query to this client's state.

Arguments:

- id - The identifier for the active query.

`addRegisteredSubscription(id : String) : void`

Adds a subscription to this client's state.

Arguments:

- id - The identifier for the registered subscription.

`getActiveQueryIds() : List`

Retrieves the list of identifiers for all active and unfulfilled queries that have been submitted by this client actor.

`getRegisteredSubscriptionIds() : List`

Retrieves the list of identifiers of all subscriptions for this client actor that are currently registered with the Information Broker.

`removeActiveQuery(id : String) : void`

Removes a query from this client's state.

Arguments:

- id - The identifier of the query to be removed.

`removeRegisteredSubscription(id : String) : void`

Removes a subscription from this client's state.

Arguments:

- id - The identifier for the subscription to be removed.

3.2.12.4 ClientRuntimeServiceStub

This interface represents a stub for the client runtime service.

3.2.13 Information Discovery

The information discovery group contains the interfaces that define and support the information discovery capability defined by the Phoenix architecture. The information discovery interfaces are:

- InformationDiscoveryService (3.2.13.1)
- InformationDiscoveryServiceConnector (3.2.13.2)
- InformationDiscoveryServiceStub (3.2.13.3)

3.2.13.1 InformationDiscoveryService

The information discovery service provides a user friendly interface for actors searching for specific descriptions of or information being operated upon by IM Services. It supports a wide variety of inquisitions such as who is submitting information, where information is persisted, how much of it there is, and many other operations too numerous to list here. This service interacts with and acts as a proxy for the other IM Services in order to simplify the interactions required between the actor and each of the IM services.

3.2.13.1.1 Public Operations

`getActorsForInformation(sessionTrack : SessionTrack, actions : InfoTypeActions[], infoTypeNames : String[]) : ActorContext[]`

Gets the ActorContext objects for the actors that are performing the specified actions upon the specified Information types. This method needs to ask the required Information services for their respective ServiceContext objects. The data to satisfy this method is contained within these context objects. Returns an array of ActorContext objects that describe the actors that meet the criteria specified by the invoker of this method.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- actions - The array of action types the invoker of this method is interested in.
- infoTypeNames - The array of information type names that the invoker of this method is interested in.

`getAllTypesSupported(sessionTrack : SessionTrack) : String[]`

Returns all the types of information that are currently valid. Returns an array of information type identifiers that are supported by the known set of IM services.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.

`getInformationTypeActions(sessionTrack : SessionTrack, informationTypeNames : String[]) : ActionContext[]`

Gets the actions being supported for the specified Information Types. Returns the ActionContext object that describes what operations may be performed upon the registered information types.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- informationTypeNames - The array of information type names that the invoker of this method is interested in.

`getInformationTypeDefinition(sessionTrack : SessionTrack, typeName : String) : InformationTypeContext`

Given the type name, it returns the InformationTypeContext object that contains the structured metadata format and/or schema, and any associated attributes. Returns the InformationTypeContext object that contains the definition of the metadata and, possibly, the payload for the specified information type.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- typeName - The name of the information type.

3.2.13.1.2 Typical Use

This service is designed to be paired with some combination of one or more Submission, Dissemination, Information Brokering, or Information Type Management services to provide the ability to discover what information is supported by the implemented IM services, what information is currently available, who is enacting what operations upon what information, etc.

3.2.13.1.3 Associated Diagrams

3.2.13.1.3.1 Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0003 Discovery (Information)

3.2.13.1.3.2 Activity Diagrams

- AD 0005 Discovery (Information)

3.2.13.1.3.3 Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0002 Discovery (Information)

3.2.13.1.3.4 Sequence Diagrams

- SQD 0014 Discovery (Information – Actors, Actions & Information)
- SQD 0015 Discovery (Information – Information Types)

3.2.13.2 InformationDiscoveryServiceConnector

This interface represents a connector for the information discovery service.

3.2.13.3 InformationDiscoveryServiceStub

This interface represents a stub for the information discovery service.

4.0 Reference

4.1 Terms

The table below gives a brief description of important terms used in this document.

Term	Meaning
Actor	A generic entity that utilizes an IM Service in some capacity. Several types of actors have been identified, such as consumers, producers, and inquisitors. An actor may represent either a service or a user of a service.
Archive	A higher latency, lower access data store than a repository. Archives may not be able to support high data rates, but can store much more data than repositories.
Attribute	Variable associated with an object.
Client	An actor that is an application, system, or service that accesses another service that is most likely on another computer system, by way of a network.
Consumer	An actor that receives information from a service.
Distributed	Deals with hardware and software systems containing more than one processing element or storage element, concurrent processes, or multiple programs, running under a loosely or tightly controlled regime.
Edge-user Application	An application at an endpoint.
Endpoint	In Service-oriented architecture, an endpoint is the entry point to a service, a process, or a queue or topic destination.
Information	The basic building block of data containing, at a minimum, an information type identifier, a payload, and metadata. Information may contain other data in addition to the specified minimal set.
Information Architecture	The structural design of shared information environments (from *Information Architecture for the World Wide Web: Designing Large-Scale Web Sites,* by Peter Morville and Louis Rosenfeld).
Information Type	An established category of information, the descriptor for which contains, minimally, a description of the payload and metadata structures and a unique identifier.
Inquisitor	A type of consumer that queries a service to retrieve information.

Metadata	A structured set of data that describes the payload without replicating it wholesale. It is sometimes called Metainformation and is "data about data", of any sort in any media. An item of metadata may describe an individual datum, or content item, or a collection of data including multiple content items. Metadata is used to facilitate the understanding, characteristics, and management of data.
Method	An operation. The public operations defined in this specification would be implemented as methods.
Object	An instance of an interface implementation.
Operation	An operation is defined as any action that is performed upon a set of targets by a set of actors.
Payload	Data of interest to be described and managed. Material transmitted over a network (either computer or telecommunications network) includes both data and information that identifies the source and destination of the material, e.g. headers. The payload is the actual data, or the cargo, carried by the headers.
Predicate	A verb phrase template that describes a property of objects, or a relationship among objects, represented by the variables.
Producer	An actor that submits information to a service.
Repository	A low-latency high-access data store that should support higher data read and write rates than an archive. Repositories typically store less data than archives, favoring access speed over possible result set size.
Schema	A conception of what is common to all members of a class; a general or essential type or form (New Oxford American Dictionary).
Service	A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistently with constraints and policies as specified by the service description (Organization for the Advancement of Structured Information Standards).

4.2 Acronyms

Acronym	Definition
AD	Activity Diagram
AFRL/RISE	Air Force Research Laboratory / Systems & Information Interoperability Branch
API	Application Programming Interface
ATO	Air Tasking Order
C2	Command and Control
CD	Class Diagram
COI	Community of Interest
DoD	Department of Defense
ENS	Event Notification Service
GIG	Global Information Grid
IETF	Internet Engineering Task Force
IM	Information Management
IMS	Information Management Service(s)
IP	Internet Protocol
JBI	Joint Battlespace Infosphere
QoS	Quality of Service
RMI	Remote Method Invocation
SAB	Scientific Advisory Board
SOA	Service-Oriented Architecture
SQD	Sequence Diagram

SSL	Secure Socket Layer
UC	Use Case
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language
XPath	XML Path Language
XSD	XML Schema Document

4.3 Interface Hierarchies

This section shows the parent and child relationships for the components of the Phoenix architecture that are specified in Section 3.0.

BaseContext
- ActionContext
- ActorContext
- AuthorizationContext
- ChannelContext
- ConnectorContext
- DestinationContext
- DisseminationContext
- EventContext
- FilterContext
- InformationContext
- InformationTypeContext
- PredicateContext
  - InformationPredicateContext
    - InformationBrokeringContext
    - InformationQueryContext*
  - QueryContext
    - ServiceBrokeringQueryContext
    - InformationQueryContext*
- ServiceContext
  - ClientRuntimeServiceContext
  - InformationServiceContext
    - DisseminationServiceContext
    - InformationBrokeringServiceContext
    - QueryServiceContext
      - RepositoryServiceContext
    - SubmissionServiceContext
    - SubscriptionServiceContext
  - ServiceBrokeringServiceContext
  - SessionManagementServiceContext
- SessionContext
- StubContext
- TransportContext

BaseChannel
- InputChannel
  - EventInputChannel
  - InformationInputChannel*
- OutputChannel
  - EventOutputChannel
  - InformationOutputChannel*

InformationChannel
- InformationInputChannel*
- InformationOutputChannel*

BaseService

- BasePersistentService
  - AuthorizationService
  - ClientRuntimeService
  - DisseminationService
  - EventNotificationService
  - InformationBrokeringService
  - InformationDiscoveryService
  - InformationTypeManagementService
  - QueryService
    - RepositoryService
  - ServiceBrokeringService
  - SessionManagementService
  - SubmissionService
  - SubscriptionService

- BaseServiceConnector
  - AuthorizationServiceConnector
  - BaseDataServiceConnector
    - DisseminationServiceConnector
    - InformationBrokeringServiceConnector
    - InformationTypeManagementServiceConnector
    - QueryServiceConnector
    - RepositoryServiceConnector
    - SubmissionServiceConnector
    - SubscriptionServiceConnector
  - ClientRuntimeServiceConnector
  - EventNotificationServiceConnector
  - InformationDiscoveryServiceConnector
  - ServiceBrokeringServiceConnector
  - SessionManagementServiceConnector

- BaseServiceStub
  - AuthorizationServiceStub
  - BaseDataServiceStub
    - DisseminationServiceStub
    - InformationBrokeringServiceStub
    - InformationTypeManagementServiceStub
    - QueryServiceStub
    - RepositoryServiceStub
    - SubmissionServiceStub
    - SubscriptionServiceStub
  - ClientRuntimeServiceStub
  - EventNotificationServiceStub
  - InformationDiscoveryServiceStub
  - ServiceBrokeringServiceStub
  - SessionManagementServiceStub
- Event
  - DeliveryReceiptEvent
  - ExceptionEvent
  - InformationTypeEvent

- Exception
  - ApplicationException
    - ChannelException
    - MultipleApplicationException
    - PredicateFailureException

PredicateTypeNotSupportedException
ValidationFailedException
MetadataValidationFailedException
PayloadValidationFailedException
ServiceException
UnableToCreateServiceException
UnableToGenerateUidException

Filter
ResidualProcessorFilter

* Dual Inheritance

4.4 Package Structure & Contents

This section depicts the package structure for the Phoenix architecture. Each package listed here corresponds to a similarly named section within the architecture specification section of this document where the interfaces and classes that comprise these packages are listed and described in detail.

mil
- af
  - rl
    - phoenix
      - channel (3.1.3)
      - client (3.2.12)
      - core (3.1.1)
      - dissemination (3.2.4)
      - event (3.1.6
      - eventnotification (3.2.8)
      - exceptions (3.1.4)
      - filter (3.1.7)
      - information (3.1.8)
      - informationbrokering (3.2.3)
      - informationdiscovery (3.2.13)
      - informationtype (3.2.1)
      - predicate (3.1.5)
      - query (3.2.6)
      - repository (3.2.7)
      - security (3.2.11)
      - servicebrokering (3.2.9)
      - sessionmanagement (3.2.10)
      - submission (3.2.2)
      - subscription (3.2.5)