



AFRL-RI-RS-TR-2011-220

**PHOENIX: SERVICE ORIENTED ARCHITECTURE FOR
INFORMATION MANAGEMENT - ABSTRACT ARCHITECTURE
DOCUMENT**

SEPTEMBER 2011

INTERIM TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

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-2011-220 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
STEVEN D. FARR
Branch Chief

/s/
JULIE BRICHACEK, 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

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**1. REPORT DATE (DD-MM-YYYY)**

SEP 2011

2. REPORT TYPE

Interim Technical Report

3. DATES COVERED (From - To)

JAN 2009 – NOV 2010

4. TITLE AND SUBTITLEPHOENIX: SERVICE ORIENTED ARCHITECTURE FOR
INFORMATION MANAGEMENT - ABSTRACT
ARCHITECTURE DOCUMENT**5a. CONTRACT NUMBER**

In House

5b. GRANT NUMBER

N/A

5c. PROGRAM ELEMENT NUMBER**6. AUTHOR(S)**J. Bryant (ITT), V. Combs (AFRL), J. Hanna (AFRL), G. Hasseler (ATC-NY),
R. Hillman (AFRL), B. Lipa (ITT), J. Reilly (RRC), C. Vincelette (ITT)**5d. PROJECT NUMBER**

S2TS

5e. TASK NUMBER

IH

5f. WORK UNIT NUMBER

03

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)AFRL/RISE, 525 Brooks Road, Rome, NY 13441-4505
ITT, 775 Daedalian Drive, Rome NY 13440
RRC, Ridge Street, Rome NY 13440
ATC-NY, Thornwood Drive, Ithaca NY**8. PERFORMING ORGANIZATION
REPORT NUMBER**

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)Air Force Research Laboratory/Information Directorate
Rome Research Site
26 Electronic Parkway
Rome NY 13441**10. SPONSOR/MONITOR'S ACRONYM(S)**

AFRL/RI

**11. SPONSORING/MONITORING
AGENCY REPORT NUMBER**
AFRL-RI-RS-TR-2011-220**12. DISTRIBUTION AVAILABILITY STATEMENT**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2011-0022

13. SUPPLEMENTARY NOTES**14. ABSTRACT**

This document specifies the architectural 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, from which one or more implementations may be developed. The architecture specifications in this document provide an approach to meeting to needs of information management in future net-centric environments.

15. SUBJECT TERMS**16. SECURITY CLASSIFICATION OF:****a. REPORT**
U**b. ABSTRACT**
U**c. THIS PAGE**
U**17. LIMITATION OF
ABSTRACT**

UU

**18. NUMBER
OF PAGES**

186

19a. NAME OF RESPONSIBLE PERSON

VAUGHN COMBS

19b. TELEPHONE NUMBER (Include area code)

N/A

Table of Contents

Introduction	1
Information Management	1
Background	2
Audience	2
Conventions	2
Diagram Conventions.....	2
Concepts	3
Contexts	4
Information	4
Information Types.....	5
Events.....	6
Actor & Service Interactions	6
Channels.....	7
Control Channels.....	7
Sessions and Session Tracks.....	8
Filters.....	10
Filter Lifecycle	10
Filter Chaining	11
Information Brokering.....	11
Service Orchestration.....	12
Use Case : Information Submission	12
Architecture Specification.....	12
Component Interfaces	12
Core	13
Information	25
Session	33
Channel	35
Expression	50
Frame	53
Event	55
Service Interfaces.....	59

Information Service Interfaces.....	59
Dissemination	59
Information Brokering	62
Information Type	67
Query	72
Repository	78
Submission	83
Utility Service Interfaces	84
Client	85
Event Notification	90
Filter	96
Information Discovery.....	102
Security	104
Service Brokering	109
Session Management.....	113
Subscription	118
Streams Service Interfaces.....	122
Connection	123
Stream Brokering	135
Stream Discovery	148
Stream Repository	152
Reference	163
Terms	163
Acronyms	164
Interface Hierarchies.....	165
Package Structure	168
How To.....	169
How To...Submit Information	169
How To...Subscribe to Information	172
How To...Store Information	176
How To...Query for Information	177

Introduction

This document specifies the architectural 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, from which one or more implementations may be developed. The architecture specifications in this document provide an approach to meeting the needs of information management in future net-centric environments.

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.

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 that 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 to 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.

Background

The Air Force and the Department of Defense (DoD) have been moving towards a network-centric concept of operations. Interest in Service Oriented Architecture (SOA) based systems will help move the concept of the Global Information Grid (GIG) closer to realization. SOA based systems group functionalities around business processes and expose them as packaged, interoperable services. These services allow applications to exchange data while performing individual or collaborative business processing. These characteristics make SOA a perfect blend of rigidity and flexibility for Information Management (IM) operations.

The Air Force Research Laboratory (AFRL) Systems & Information Interoperability Branch (RISE) main research focus has been Information Management. The branch grew out of the pioneering work done in this field by the members of the Joint Battlespace Infosphere (JBI) project. In turn, this 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. The current effort, called Project Phoenix, will leverage all of the existing in-house knowledge of IM, along with the expertise of its external research efforts as well as requirements gathered from customers, to produce a SOA-based IM solution. This solution will be aligned with the Air Force and DoD's vision of current and future network-centric operations.

Audience

This document is intended for two types of readers: those who are implementing the architecture as specified and those who are looking for an overview of the key architectural concepts.

Implementers will need 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>.

For those who are interested in an overview of the key architectural concepts, this document is all that is required.

Conventions

This document provides both a literal and conceptual design of the Phoenix architecture. The literal architecture 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.

Diagram Conventions

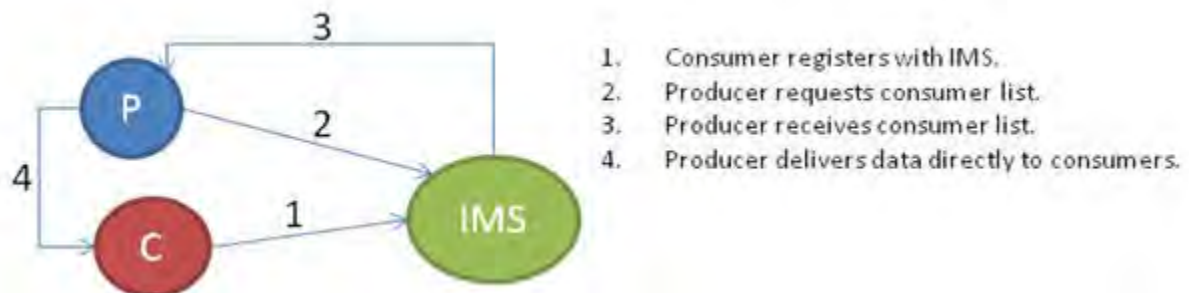
Throughout this document there are a number of non-UML diagrams that are used to illustrate high-level concepts. Samples of these diagrams are shown below along with usage information.

The figure below shows a sample communication between Phoenix entities via 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 figure below is a sample diagram showing labeled information flow.



Concepts

This section specifies the key concepts in the Phoenix architecture:

- Contexts
- Information
- Information Types
- Events

- [Actor & Service Interactions](#)
- [Channels](#)
- [Control Channels](#)
- [Sessions and Session Tracks](#)
- [Filters](#)
- [Information Brokering](#)
- [Service Orchestration](#)

Contexts

The context is one of the basic constructs within the Phoenix architecture. Contexts are used by Phoenix services and components for storing state and configuration settings. Each context contains a set of key-value pairs. Helper methods are defined within many contexts that require specific keys with specific types of values. For example the [Service Context](#) interface defines the set and retrieval methods for a key whose value represents the current service status. This key requires values to be members of the [Service Status](#) enumeration. Contexts are not limited to storing predefined keys and values however. The [Base Context](#) interface, which is the parent interface for all contexts, was designed with idea that it could support setting and retrieving values of any type for any specific key defined by any implementation of the architecture. Contexts are meant to be easily extensible to accommodate additional attributes based on operational requirements within any implementation-level design.

Since contexts are used to store configuration settings for Phoenix entities, they need to support a mechanism for notifying their parent entities, or any other system construct, when a value is changed at runtime. This concept is realized by the definition and application of the [Base Attribute Update Callback](#). When one of these callbacks is applied to a specific key-value pair within a context, any entity that has expressed an interest in being notified when that value changes is notified whenever a change occurs. The most obvious use case for this capability is controlling the configuration setting updates for Quality of Service (QoS) managed information channels. In this case the callback would be the mechanism that alerts the channel that one of its settings has been modified, thus kicking off some processing that effects a change in the way the channel is performing its job.

Contexts are conceptualized to be unique to their context container. This means that any entity in an implementation of the Phoenix Architecture that implements the Context Container interface or one of its sub-interfaces is expected to contain one and only one context instance that is unique to the container. This is done to ensure that different instances of the same implementation classes end up with unique contexts that describe them and their respective state.

Information

Information, the central concept behind IM, is the currency that flows between and among actors. A complete unit of information consists of:

- An **information type** identifier
 - o Tags the **payload** with a known and defined information structure.
 - o May or may not be optional depending on how designers wish to implement the ability to support untyped information.
- **Metadata** (the description of the **payload**)
 - o May or may not be a subset of the **payload**.
 - o May duplicate some or the entire **payload**.
 - o Is used for Brokering.
 - o Is used for Storage and Retrieval.
 - o May be used by filters.
- A **payload** (the actual data being managed)
 - o May consist of known or unknown content.
 - o May or may not be used by an actor.
- An **InformationContext**
 - o 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:

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:

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** 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).

Information Types

Information types are used to uniquely identify classes of information. 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 underlying 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](#) 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](#) 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 (XSD). XML and XSD were chosen as examples in an attempt to explain an abstract idea using a common, well-known, and understood representation.

Additional attributes for information types may be defined by 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 unique metadata generation routines for each information type.

Information types are managed by the [Information Type Management Service](#).

Events

The Phoenix architecture defines events as non-managed data items that facilitate interactions between actors. The key difference between events and information is that the event hierarchy and structures must be defined by the implementation at design time while information types can be defined by any actor with sufficient privileges at runtime.

Some simple examples of events are messages used for confirmation of information submission and delivery, notification of which actors are interested in certain types of information, control messages used to facilitate the creation and destruction of channels, and just about any other interaction or status message you can think of. Events may be managed in some capacity, but they are not managed in the same ways that information instances are. Events are meant to facilitate the transportation of service level data items that are not supposed to be visible to actors. To facilitate event communications the Phoenix architecture defines specific channel interfaces for events.

Actor & Service Interactions

Actors communicate with services via channels. Data to be moved through and processed by the services is transported from the actor to the service by byte, event, frame, and information

channels. Service methods are invoked through control channels. Invocations of service methods are monitored at the service level by the use of actor sessions and session tracks.

Channels

Channels are the central construct for the Phoenix architecture because they facilitate all interactions between all Phoenix actors.

Channel implementations are organized by application and transport protocols. The specific protocols are defined by the Phoenix implementations: for example application protocols may be defined as information, byte, event, and frame while the transport protocols may be Transport Control Protocol (TCP) and Universal Data Protocol (UDP). A specific channel used by a Phoenix actor is a combination of an application protocol and a transport protocol. For example, based on the aforementioned protocols, a channel for an actor may be `information:tcp` which translates to an information channel that uses TCP as its transport.

Channels are defined within the Phoenix architecture by the `BaseChannel` interface. The control mechanisms for creating and maintaining channel instances are defined by the `BaseChannelService` interface. The architecture also defines several standard flavors of channels including byte, event, frame, and information. More varieties of channels are allowed due to the extensibility of the architecture, these were defined due to the associated objects being central components of the architecture (events are tied to the Event Notification Service, information is tied to several services, etc.).

Channel Lifecycle

Channels are created by the service or actor that will utilize them. There is a service level method for configuring an actor output channel. This method exists to allow an actor the ability to ask a service to configure a channel for the actor to use to communicate with that service. For example, if an actor calls this method with a channel context the service may set the host address, host port, and/or the application and transport protocols for the actor's channel to use.

The lifecycle of a channel is similar to that of a Java socket: create, open, utilize, close. The main difference between a Java socket and a Phoenix channel is that a channel may be suspended and resumed without being physically closed. The creating actor is responsible for maintaining the lifecycle of the channel.

Control Channels

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

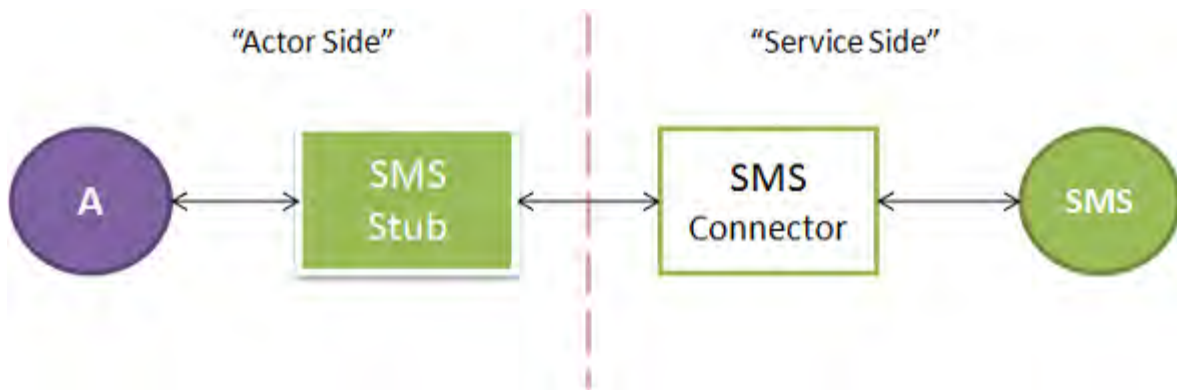


Figure 1 - Connectors and Stubs

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

Control stubs for services may be obtained either through static configuration or some runtime mechanism specific to the connector-stub implementation (for example RMI would require the use of an RMI registry and lookups).

Sessions and Session Tracks

A session is a sustained interaction between an actor and a set of other actors. A session may represent a one-to-one or a one-to-many relationship. Sessions are used to store configuration data about an actor and what actions they are undertaking within the Phoenix information space. Such configuration data can include, but is not limited to:

- The set of information channels (input and output) associated with this actor,
- The set of filters this actor is allowed to utilize,
- Any specific security policy restrictions or rights that apply to this actor only,
- Any other configuration data deemed to be of some use or value that should be associated with a session and stored for later retrieval.

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 used to monitor actors' interactions with other actors by maintaining state that describes what operations an actor is performing. A session track is a construct that holds only session identifiers and is used by all actors to enable policy decisions during control method invocations.

Figure 2 shows an actor (A) as the originator of two session tracks (ST^I , ST^{II}), both using the same session identifier (A^{SID}). Each Phoenix information service (SS, IBS) that has one of its methods invoked via a control channel receives one of these session tracks and annotates it with its own session identifier (IBS^{SID} , SS^{SID}). These services also invoke methods on the Session Management Service (SMS) to update the associated session context as applicable, perhaps with descriptions of information being submitted or subscriptions registered. These services also communicate with the Authorization Service (AS) as required in order to authorize the method invocations made by the actor.

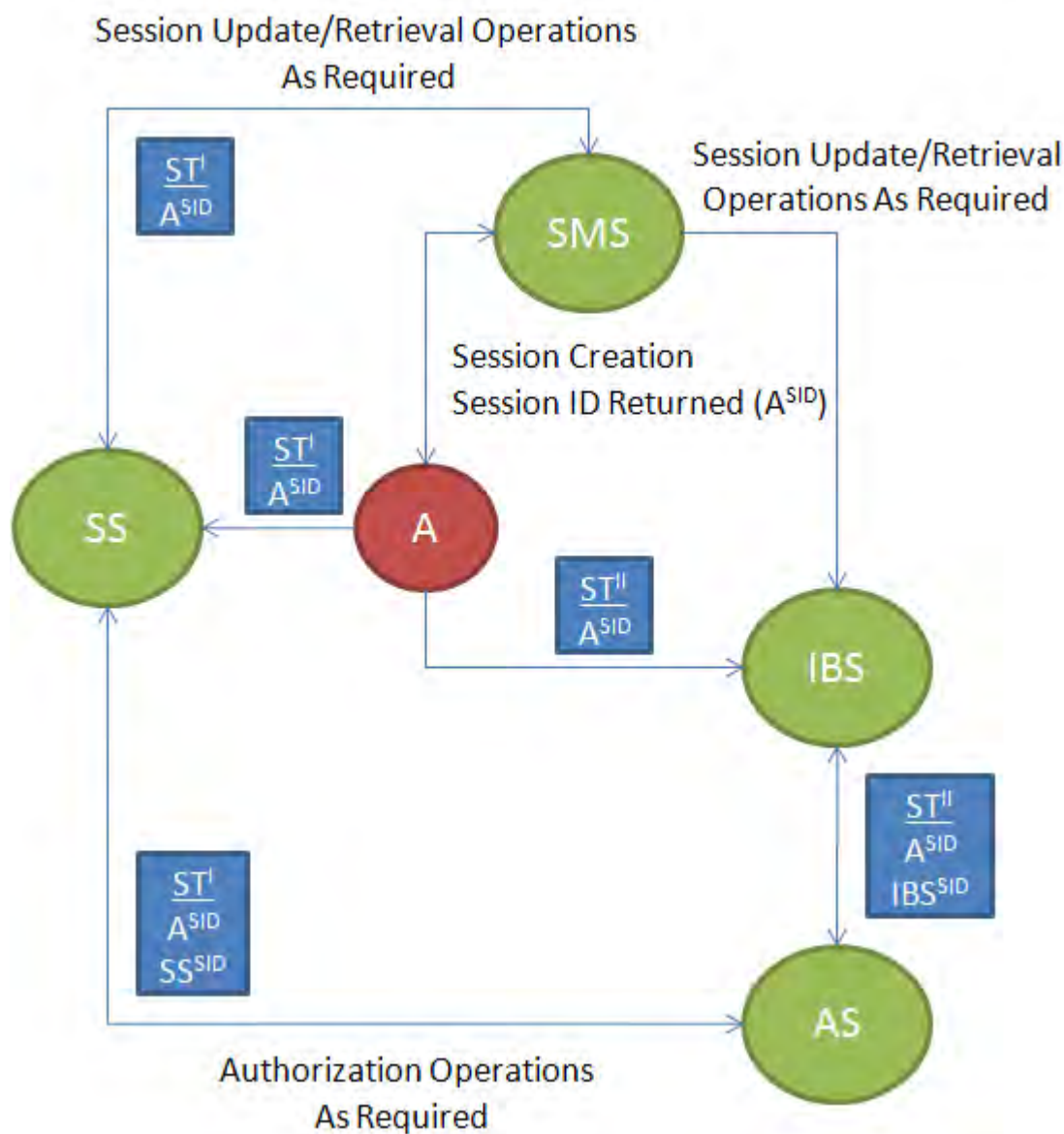


Figure 2 - Sessions & Session Tracks

Diagram Key

A The Actor who creates a session and begins invocation chains using session track instances.

AS The Authorization Service.

IBS The Information Brokering Service.

SID A Session Identifier.

SMS The Session Management Service.

SS The Submission Service.

ST A Session Track.

Sessions and session tracks are defined in the architecture by the [SessionContext](#) and [SessionTrack](#) interfaces, respectively.

Filters

The Phoenix architecture defines an adaptable data filtering capability that may be applied to any component within the architecture. The most obvious use for filters is any interaction between a pair of actors but filters may also be applied to inter-service operations and components such as subscriptions. For example a subscription may have multiple consumers and security may dictate a key-word filter be applied to all outgoing information. In this case it may be optimal for the information filtering to be done within the Information Brokering Service instead of within the Dissemination Service.

The architecture provides an interface for a filtration chain mechanism dubbed Filter Chain. This entity, whose instances are constructed by the factory method available as part of the Filter Management Service, embodies a chain of filters that are be invoked in the order they are linked and the operations that each filter will be performing including each filter's required input and expected output. The Filter interface provides a generic filtration method that takes an Object as its parameter and returns the same. This allows the Phoenix filters their aforementioned flexibility. This transparency comes with a drawback however; each specific filter implementation must know a priori exactly what type of Object it is receiving to perform filtering operations over. This transparency also means that the Filter Chain entities must know exactly what is being returned by each specific filter in order to ensure that the inputs and outputs of chained filters match up exactly.

Filters are defined in the architecture by the [Filter](#) interface.

Filter Lifecycle

It is envisioned by the Phoenix Design Team that all filters are to be registered with at least one [Filter Management Service](#) (FMS) shortly after creation and before they can be utilized. This is enforced by the definition of methods within both the Filter and Filter Chain interfaces that are only visible at the package level, thereby guaranteeing that any code external to the Phoenix filter package cannot invoke them. Filters are able to be created by any actor within the Phoenix environment, but only those actors who can communicate with the FMS may actually utilize them for filtration operations. Any single filter's lifecycle can be summed up by the following steps:

1. Creation,
2. Registration,
3. Utilization,
4. De-registration
5. Destruction.

At creation time a specific filter implementation class is instantiated in some actor's address space. The filter is then registered with one or more FMS's and utilized by any actors who, based on any applicable policies, have the necessary privileges. When the filter is no longer pertinent, or some actor with the necessary privileges deems it necessary, the filter is dropped from the FMS's registry and destroyed.

Filter Chaining

Filters may be chained together into a [Filter Chain](#) (FC), a concept that is embodied within the Phoenix architecture by an interface of the same name. A FC is the entity that an actor invokes the filtration method on to enact a filter operation. This ability is provided by exposing the same generic filtration method signature resident in all filters at the FC interface. Once a FC's filter method has been invoked, the first filter performs its coded operations and then returns its result to the FC. The FC then iterates through each of the subsequent filters in the chain, if any are designated, and returns the final result to the invoking actor. Filter Chain instances are created by the FMS. However, the FMS does not maintain the FC instances. This is the responsibility of the requisitioning actor. However the FMS does carry the burden of being the sanity check point for filter chaining. When FC's are created by the FMS, the service checks each step in the requested filter chain to ensure that the next filter's input type matches the previous filter's output type. If a mismatch is found, it is identified and a detailed exception is thrown by the service back to the requisitioning actor. This operation has been designated as a responsibility of the FMS in order to avoid the regurgitation of heavy logic such as this within the address space of every Phoenix actor that wishes to create a filter chain. By placing this logic and processing in a central (or distributed) service-oriented point, the Phoenix architecture provides an optimized, cohesive, and discoverable solution to the data filtration problem plaguing today's information highways.

Information Brokering

Information brokering is defined as matching information requests to known instances of information. The Phoenix Architecture defines an interface for an [Information Brokering Service](#), but does not place any specific limitations upon or attempt to constrain exactly how this service is implemented.

Subscription is the term used to describe an actor's request for information. Subscriptions may have one or more expressions associated with them. Subscription expressions define filter parameters for selecting information that is of interest to the subscription's consumers and may be as simple as a type identifier or as complex as a regular expression or and XQuery statement. Subscriptions and expressions are defined using the context interfaces located within the '[subscription](#)' and '[expression](#)' packages, respectively.

Such open specification of the [Information Brokering Service](#) in the Phoenix Architecture allows for an infinite number of possible service implementations, thus leaving the way ahead clear for new and emerging brokering technologies such as bulk or priority-based expression processing.

Service Orchestration

Service orchestration is defined by the Design Team to be the chaining or linking of two or more service instances to accomplish some specific task for one or more actors. Such chains can be statically or dynamically constructed based on specific implementation requirements. The decision to support one or both of these is made at implementation design time. The Phoenix service's dependence upon flexible structures such as the context and channel interfaces provides implementation designers with a measure of much-needed flexibility when making such design-time decisions.

The Phoenix Architecture does not define any interfaces for linking services in this way. By leaving this as extensible as possible the architecture allows implementors to design their own solutions customized for their specific requirements.

Use Case : Information Submission

One existing use case of service orchestration is the act of submitting information to the Phoenix services. This is accomplished by interacting with and transmitting information instances to the Submission Service. Once this service receives an information instance it must decide based on local implementation logic and policy if and which Information Brokering and Repository Services the information is going to be forwarded to. These service chains, Submission-to-Information Brokering and Submission-to-Repository, are fundamental examples of simple service orchestrations.

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, when 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 "expression" group all provide support for expression 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 by this documentation as any action that is performed upon a set of targets by a set of actors.

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 functional groups, ordered by overall importance to the architecture as a whole:

- [Core](#)
- [Information](#)
- [Session](#)
- [Channel](#)
- [Expression](#)
- [Frame](#)
- [Event](#)
- [Stream](#)

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, listed alphabetically, are:

- [ActivationCallback](#)
- [BaseAttributeUpdateCallback](#)
- [BaseContext](#)
- [BasePersistentService](#)
- [BaseService](#)
- [BaseServiceConnector](#)
- [BaseServiceStub](#)
- [ConnectionContext](#)
- [ConnectorContext](#)
- [ContextContainer](#)
- [ServiceContext](#)
- [ServiceDescriptorContext](#)
- [ServiceStatus](#)
- [StubContext](#)

ActivationCallback

The Activation Callback is intended to provide actors a way to be asynchronously notified when the assigned `stub`'s activation is complete and it is 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 start-up circular dependency problem, where service A depends on Service B and Service B depends on Service A.

Public Operations

`onActivation() : void`

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

BaseAttributeUpdateCallback

This interface accommodates the case where one or more of the values stored in a context has been changed and said change needs to ripple down to interested parties.

Public Operations

`getAttributesOfInterest() : List<String>`

Retrieves any context attribute keys that the actor is interested in when updated. This method returns the registered context attribute keys of interest

`getCallbackId() : String`

Retrieves the unique ID associated with this callback. This method returns the unique callback ID.

`setAttributesOfInterest(attributesOfInterest : List<String>) : void`

Sets any context attribute keys that the actor is interested in when updated.

Arguments:

- `attributesOfInterest` - The registered context attribute keys of interest

`update(updatedAttributes : Map<String, Object>) : void`

This method is called to notify the interested party that context attributes of registered interest have been updated. Note that a null entry for a specific key value in `updatedAttributes` is associated with a context attribute that has been REMOVED.

Arguments:

- `updatedAttributes` - The attribute value changes provided by notifying context.

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-interfaces: `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.

Public Operations

`addUpdateCallback(updateCallback : BaseAttributeUpdateCallback) : void`

This method allows for the addition of a context callback. The context implementation itself will then, based on the registered attributes of interest contained within the callback, invoke the callback providing the attribute changes.

Arguments:

- `updateCallback` - The callback used to notify interested actor of changes in context attributes.

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

`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 you want returned from the Map. This is the key in the Map.

`getAttributes(attributeNames : List<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` - A list 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`.

`getName() : String`

Retrieve the name for this context. This is an identifier that may be set by applications and used for such things as policy enforcement.

`isForceDeepCopy() : boolean`

This method returns true if a copy operation invoked on this context is forced to return a deep copy of the context and false if the decision whether to deep copy the context is left up to the copier.

`isSanitized() : boolean`

This method returns true if the context has been sanitized at least once. If not it returns false.

`removeAttribute(attributeName : String) : void`

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

Arguments:

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

`removeAttributes(attributeNames : List<String>) : void`

Remove the supplied attributes from the attributes Map.

Arguments:

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

`removeUpdateCallback(callbackId : String) : void`

This method removes a registered context callback.

Arguments:

- callbackId - The unique identifier associated with the registered callback that is to be removed.

`sanitize() : void`

This method removes all attribute values that should not be forwarded to the next actor.

`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 that you want to add.
- attributeValue - The value for the attribute.

`setAttributes(attributes : Map<String, Object>) : void`

This method takes a Map of key-value pairs and sets each one within the Map maintained by the [Context](#).

Arguments:

- attributes - The Map of attributes and their new values.

`setForceDeepCopy(forceDeepCopy : boolean) : void`

Set the flag telling whether or not copying this context forces a deep copy operation.

Arguments:

- forceDeepCopy - True if a copy operation invoked on this context is forced to return a deep copy of the context and false if the decision whether to deep copy the context is left up to the copier.

`setName(name : String) : void`

Set the identifier to be used by applications for things such as policy enforcement.

Arguments:

- name - The identifier to be used by applications for things such as policy enforcement.

BasePersistentService

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

Public Operations

`load() : void`

Load a stored service state from some persistent data store.

Raised Exceptions:

- Exception

`store() : void`

Store the service state in some persistent data store.

Raised Exceptions:

- Exception

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.

Public Operations

`getStatus() : ServiceStatus`

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

Raised Exceptions:

- Exception

`resume() : void`

Resume normal service operations.

Raised Exceptions:

- Exception

`start() : void`

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

Raised Exceptions:

- Exception

`stop() : void`

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

Raised Exceptions:

- Exception

`suspend() : void`

Temporarily suspend service operations.

Raised Exceptions:

- Exception

Protected Operations

`getServiceContext() : ServiceContext`

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

Raised Exceptions:

- Exception

BaseServiceConnector

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

Public Operations

`activate() : void`

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

Raised Exceptions:

- Exception

`deactivate() : void`

This method de-activates the connector, which suspends all invocations from the connector to the actual service. Note that de-activating a connector does NOT destroy it.

Raised Exceptions:

- Exception

`getConnectorContext() : ConnectorContext`

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

Raised Exceptions:

- Exception

BaseServiceStub

All [services](#) are interacted with 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 [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 that the [actor](#) invokes methods upon. 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](#).

Public Operations

`activate(async : boolean) : void`

Activate this stub locally.

Arguments:

- `async` - This parameter tells the stub if the encompassing activation call is synchronous or asynchronous.

Raised Exceptions:

- Exception

`deactivate() : void`

De-activates the stub which means that all invocations from stub to its associated connector are suspended. NOTE that de-activating a stub does NOT destroy it.

Raised Exceptions:

- Exception

`getStubContext() : StubContext`

This method returns the current instance of the [StubContext](#).

Raised Exceptions:

- Exception

`isActive() : boolean`

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

Raised Exceptions:

- Exception

`registerActivationCallback(callback : ActivationCallback) : void`

This method sets the `ActivationCallback` for the stub.

Arguments:

- `callback` - The callback to be fired when the stub is activated.

Raised Exceptions:

- `Exception`

ConnectionContext

This context describes a control channel connection between an actor and a service. It contains the four minimally necessary items for constructing a physical connection between two nodes including the host address, host port, connector identifier, and protocol used for the connection.

Public Operations

`getConnectorAddress() : String`

Retrieve the address of the registry used by the connector.

`getConnectorName() : String`

Retrieve the name of the connector. For use by technologies such as RMI that require a unique identifier.

`getConnectorPort() : int`

Retrieve the port used by the connector and stub.

`getProtocol() : String`

Retrieve the protocol being used by the parent control channel.

`setConnectorAddress(connectorAddress : String) : void`

Set the address of the registry used by the connector.

Arguments:

- `connectorAddress` - The address of the registry used by the connector.

`setConnectorName(connectorName : String) : void`

Set the name of the connector. For use by technologies such as RMI that require a unique identifier.

Arguments:

- `connectorName` - The name of the connector. For use by technologies such as RMI that require a unique identifier.

`setConnectorPort(connectorPort : int) : void`

Set the port used by the connector and stub.

Arguments:

- `connectorPort` - The port used by the connector and stub.

`setProtocol(protocol : String) : void`

Set the protocol being used by the parent control channel.

Arguments:

- protocol - The protocol being used by the parent control channel.

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 that this `connector` is associated with.

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 service that this `connector` is connected to.

`setService(service : BaseService) : void`

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

Arguments:

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

ContextContainer

An interface defining a container for a single context.

Public Operations

`getContext() : BaseContext`

Retrieve the context stored within this container.

`setContext(ctx : BaseContext) : void`

Set the context to be stored by this container.

Arguments:

- ctx - The context to be stored by this container.

`updateContext(attributesAndValues : Map<String, Object>) : void`

Updates the attributes and values stored by the context within this container.

Arguments:

- attributesAndValues - The new attributes and their respective values.

ServiceContext

This [Context](#) maintains a description of the service, its capabilities, and any state data being maintained by the service.

Public Operations

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

Retrieve the functional description of the service. This description may include details such as: what operations the service supports, what information types it may perform operations upon, etc. The exact contents of this description are left up to the implementation designers to determine.

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

Retrieve the operational description of the service. This description may contain details such as: what types of information the service is currently operating upon, performance metrics, etc. The exact contents of this description are left up to the implementation designers to determine.

`getServiceDescriptor() : ServiceDescriptorContext`

Retrieve the descriptor for this service.

`getServiceName() : String`

Retrieve the human readable name for the associated service.

`getServiceStatus() : ServiceStatus`

Retrieve the identifier for the current status of the described service. Possible values for this identifier are defined in the [ServiceStatus](#) enumeration.

`getServiceTypes() : List<String>`

Retrieve the listing of identifiers that signal what types of functionality the described service supports. The values for this field are implementation specific.

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

Set 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`

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

Arguments:

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

`setServiceDescriptor(context : ServiceDescriptorContext) : void`

Set the descriptor for this service.

Arguments:

- context - The descriptor for this service.

`setServiceName(serviceName : String) : void`

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

Arguments:

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

`setServiceStatus(status : ServiceStatus) : void`

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

Arguments:

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

`setServiceTypes(types : List<String>) : void`

Set the list of identifiers stating what service interfaces the service that this context describes implements.

Arguments:

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

ServiceDescriptorContext

This context is used to describe a service. The contents of this context is defined in part by the helper methods enumerated within its interface and in full by the set of schemas stored by the Service Brokering Service. These schemas describe the searchable elements and attributes of service descriptions.

Public Operations

`getServiceName() : String`

Retrieve the service name.

`getServiceTypes() : List<String>`

Retrieve the list of service types that identifies the Phoenix service interfaces that the described service implements.

`listControlChannels() : List<BaseServiceStubInterface>`

Retrieve the list of stubs that embody the control channels for this service.

`listControlChannelTypes() : List<String>`

Retrieve a list of the control channel types (I.E. RMI, PIC, etc.) from this descriptor.

`listSupportedExpressionTypes() : List<String>`

Retrieve the list of expression types supported by this service, if any.

`listSupportedInformationTypeNames() : List<String>`

Retrieve the list of information type names supported by this service, if any.

ServiceStatus

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

Public Fields

AVAILABLE

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

STARTED

The service has been started and is 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.

SUSPENDED

Reflects a service whose normal operations have been temporarily suspended.

UNAVAILABLE

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

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.

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`

Set the [ServiceContext](#) for the [stub's](#) associated [service](#).

Arguments:

- ctx - The [context](#) for the [stub's](#) associated [service](#).

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](#)
- [ConfirmationType](#)
- [Information](#)
- [InformationAction](#)
- [InformationChannel](#)
- [InformationChannelContext](#)
- [InformationContext](#)
- [InformationInputChannel](#)
- [InformationOutputChannel](#)
- [InformationServiceContext](#)

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.

Public Operations

`getInfoTypeActions() : List<InformationAction>`

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

`getInfoTypeNames() : List<String>`

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<InformationAction>) : void`

Set the list of actions being performed upon information.

Arguments:

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

`setInfoTypeNames(typeNames : List<String>) : void`

Set the information type names for this context.

Arguments:

- typeNames - The list of information type names.

ConfirmationType

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

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 their information submission attempts fails. The meaning of "submission failure" is left to the implementation designers to define.

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 piece of information.

Information may also be degraded, meaning that it has been subject to some form of loss of content during its lifecycle. Once information has been degraded, it can never be upgraded. The degradation flag serves as an indicator to Phoenix actors that what they received is not the pristine information instance, but a degraded copy of the original instance. The Phoenix architecture does not prevent said actor from attempting to retrieve the pristine information instance, this is the job of implementation level security components such as authorization policies. Information degradation is thought to be embodied by many different functions, which is why the Information Context contains an integer variable designating the acceptable degradation modes available for a specific instance of information.

Public Operations

`getInformationContext() : InformationContext`

Retrieve the `Context` providing additional characterization for this piece of information.

`getMetadata() : Object`

Retrieve the metadata for this instance of managed data. 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() : Object`

Retrieve 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.

`getTypeName() : String`

Retrieve the type name for this information instance. This type identifier maps to an information type definition.

`isDegraded() : boolean`

Indicate whether the information has been degraded from its original form. Degradation refers to any filtering or other modification that that irreversibly reduces the quality of information: for instance, lossy compression applied to reduce information size. Degradation may be necessary for quality-of-service reasons.

`markAsDegraded() : void`

Set the flag indicating that the information has been degraded. Degradation refers to any filtering or other modification that that irreversibly reduces the quality of information: for instance, lossy compression applied to reduce information size. Degradation may be necessary for quality-of-service reasons. Because degradation is irreversible, once this method is called, the information is irrevocably marked as degraded. There is no way to "unmark" information as degraded. For ease of code maintainability, it is strongly recommended to call this function after every modification that degrades information.

`setMetadata(metadata : Object) : void`

Set the metadata for this instance of managed data.

Arguments:

- `metadata` - The metadata for this instance of managed data.

`setPayload(payload : Object) : void`

Set the payload for this instance of managed data.

Arguments:

- payload - The payload for this instance of managed data.

`setTypeName(typeName : String) : void`

Set the type name for this information instance. This type identifier maps to an information type definition.

Arguments:

- typeName - The type name for the information instance.

InformationAction

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

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.

InformationChannel

An interface for information channels.

Public Operations

`getInformationChannelContext() : InformationChannelContext`

Retrieve the information channel specific context.

`isTyped() : boolean`

Check if this channel is typed or not. A typed information channel sends and receives information of specified information types. An un-typed information channel can send and receive information of any information type.

`listInformationTypeNames() : List<String>`

Retrieve the list of information type names that this channel supports. Any information not of one of these types will cause an exception to be raised.

InformationChannelContext

A channel context specific to typed information channels.

Public Operations

`listInformationTypeNames() : List<String>`

Retrieve the list of information types that this channel supports.

`setInformationTypeNames(typeNames : List<String>) : void`

Set the list of information types that this channel supports.

Arguments:

- `typeNames` - The list of information types that this channel supports.

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](#). Special flags for information include the: *persistenceFlag*, *brokeringFlag*, *receiptRequestFlag*, and *degradableFlag*.

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 *degradableFlag* signals what mode(s) of degradation are allowed for this instance of managed information.

Public Operations

`addConsumers(consumerChannels : List<ChannelContext>) : void`

Stamp this information context with a set of consumers for the information. This is an append operation.

Arguments:

- consumerChannels - The set of consumer channels to use to deliver this information instance.

`getAllowedDegradationMode() : int`

Retrieve the degradation mode flag, which signals what mode(s) of degradation are allowed for this instance of managed information. Semantics of this flag are implementation dependent; at minimum it should be treated like a boolean (zero means no degradation allowed, non-zero means at least one kind of degradation allowed). It is anticipated that in some implementations the degradation flag may be a bitmask encoding multiple degradation modes, all of which are allowed.

`getBrokeringFlag() : int`

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

`getConsumers() : List<ChannelContext>`

After an Information Brokering operation, this attribute contains the list of channels for the consumers that wish to receive the information instance.

`getPersistenceFlag() : int`

Retrieve 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.

`getReceiptRequestFlags() : List<ConfirmationType>`

Retrieve the flags that signal what types of delivery confirmation is being requested by the producer of the information (if any).

`setAllowedDegradationMode(degradableFlag : int) : void`

Set the degradation mode flag for this instance of information. The degradable flag indicates how information-degrading filters are permitted to modify the information (which may be necessary for quality-of-service purposes). Semantics of this flag are implementation dependent; at minimum it should be treated like a boolean (zero means no degradation allowed, non-zero means at least one kind of degradation allowed). In some implementations the value of this flag may be a bitmask encoding multiple degradation modes, all of which are allowed.

Arguments:

- degradableFlag - The degradation mode flag for this instance of information.

`setBrokeringFlag(brokeringFlag : int) : void`

Set the brokering mode flag for this instance of information.

Arguments:

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

`setPersistenceFlag(persistenceFlag : int) : void`

Set the persistence mode flag for this instance of information.

Arguments:

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

`setReceiptRequestFlags(flags : List<ConfirmationType>) : void`

Set the receipt request confirmation flags for this instance of information.

Arguments:

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

InformationInputChannel

An interface for an information input channel.

Public Operations

`read() : Information`

Read a single information instance from the channel.

Raised Exceptions:

- [ChannelException](#)

`read(numberToRead : int) : List<Information>`

Read a set of information instances from the channel.

Arguments:

- numberToRead - The number of information instances to read from the channel. This method will not return until it reads this number of instances.

Raised Exceptions:

- [ChannelException](#)

InformationOutputChannel

This interface defines an information-specific output channel.

Public Operations

`write(information : Information) : void`

Writes an instance of [information](#) to the [Channel](#).

Arguments:

- information - The [information](#) instance that is to be written to the [Channel](#).

Raised Exceptions:

- [ChannelException](#)

`write(information : List<Information>) : void`

Writes an array of instances of [information](#) to the [Channel](#).

Arguments:

- information - The [information](#) instances that are to be written to the [Channel](#).

Raised Exceptions:

- [ChannelException](#)

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 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 channels established with the associated service.

Public Operations

[addConnectedActor\(actorId : String\) : void](#)

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

Arguments:

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

[getConnectedActors\(\) : List<String>](#)

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

[getCurrentThroughputRate\(\) : long](#)

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

[getMaxSupportedThroughputRate\(\) : long](#)

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

`getSupportedInformationTypes() : List<String>`

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

`removeConnectedActor(actorId : String) : void`

Remove the identified actor from the list of connected actors.

Arguments:

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

Session

The session group contains the interfaces that define the constructs used to support session management:

- [ActorContext](#)
- [SessionContext](#)
- [SessionTrack](#)

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, or the [session](#) identifier 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 expressions for this [actor](#). [Actor](#) contexts are tracked by the [Session Management Service](#) and utilized by this service to create sessions.

Public Operations

`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](#).

`setSessionId(sessionId : String) : void`

Set the [session](#) identifier for this [actor](#).

Arguments:

- sessionId - The [session](#) identifier.

SessionContext

The Phoenix architecture identifies interactions between individual [actors](#) by creating [sessions](#) for each semi-permanent 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 is 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 control stub 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 of 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.

Public Operations

[getActorContext\(\)](#) : [ActorContext](#)

Returns the ActorContext for the actor associated with this session.

[getBroker\(\)](#) : [BaseServiceStub](#)

Returns the control stub for the Service Broker Service, if this field has been set.

[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 had an up-to-date copy of the context to begin with.

[setActorContext\(ctx : ActorContext\)](#) : [void](#)

Set the [ActorContext](#) for the actor associated with this session.

Arguments:

- ctx - The [ActorContext](#) for the actor associated with this session.

`setBroker(stub : BaseServiceStub) : void`

Set the control stub for the `ServiceBrokeringService` for this session (if applicable).

Arguments:

- stub - The control stub for the `ServiceBrokeringService` for this session (if applicable).

`timeStamp() : void`

Stamp this `Context` with the current date and time.

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`.

Public Operations

`addSessionId(sessionId : Object) : void`

Add 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() : Object`

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

`getCurrentSessionId() : Object`

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

`getSessionPedigreeList() : List<Object>`

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.

Channel

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`
- `BaseChannelService`

- [BaseChannelServiceConnector](#)
- [BaseChannelServiceStub](#)
- [ByteInputChannel](#)
- [ByteOutputChannel](#)
- [ChannelContext](#)
- [ChannelException](#)
- [ChannelServiceDescriptorContext](#)
- [ChannelState](#)
- [EndPointContext](#)
- [Handler](#)
- [InputChannel](#)
- [InputChannelContext](#)
- [InputHandler](#)
- [OutputChannel](#)
- [ProtocolContext](#)
- [TransportProtocolContext](#)

[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.

Public Operations

[getChannelContext\(\)](#) : [ChannelContext](#)

Retrieve the [ChannelContext](#) that describes this Channel instance.
This method returns a [ChannelContext](#).

[isActive\(\)](#) : [boolean](#)

Depending on if this is an input or output channel, check if the channel has been opened or connected.

[isInput\(\)](#) : [boolean](#)

This method returns true if the channel instance is an implementation of the [InputChannel](#) interface, and false if the channel is an implementation of the [OutputChannel](#) interface.

`updateContext(attributesToUpdate : Map) : void`

Update 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.

BaseChannelService

This service interface extends the Base Service and provides administration methods for managing a service's channels.

Public Operations

`configureActorOutputChannelContext(sessionTrack : SessionTrack, channelCtx : ChannelContext) : ChannelContext`

Associate the given channel context with a service input channel and session identifier (from the SessionTrack). Implementations of this method may automatically create service input channels when an output channel is requested.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelCtx - The context that describes the output channel to be created by the actor.

Raised Exceptions:

- Exception

`createInputChannel(sessionTrack : SessionTrack, channelCtx : ChannelContext) : String`

Create a new input channel for the service to use to communicate with another actor.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelCtx - The configuration to use for the new input channel.

Raised Exceptions:

- Exception

`createOutputChannel(sessionTrack : SessionTrack, channelCtx : ChannelContext) : String`

Create a new output channel for the service to use to communicate with another actor. This method should be used by orchestration services to align services into information pipelines.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelCtx - The configuration to use for the new output channel.

Raised Exceptions:

- Exception

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

Destroy the identified input channel.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelId - The identifier for the channel to be destroyed.

Raised Exceptions:

- Exception

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

Destroy the identified output channel.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelId - The identifier for the channel to be destroyed.

Raised Exceptions:

- Exception

`getAvailableInputChannels(sessionTrack : SessionTrack): List<ChannelContext>`

Retrieve the list of channel contexts that describe the locations of the input channels for this service.

Arguments:

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

Raised Exceptions:

- Exception

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

Return the hierarchical view of service attributes that can be modified by external actors. For example, this method could return the list of identifiers for the channels that the service owns and their settings and the channel filters and their settings.

Arguments:

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

Raised Exceptions:

- Exception

`getServiceAttributeValue(sessionTrack : SessionTrack, attributeName : String) : Object`

Retrieve an attribute's value from the ServiceContext for this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- attributeName - The identifier for the attribute whose value should be retrieved.

Raised Exceptions:

- Exception

`getServiceId(sessionTrack : SessionTrack) : Object`

Arguments:

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

Raised Exceptions:

- Exception

`resumeChannels(sessionTrack : SessionTrack, channelIds : List<String>) : void`

Resume normal operations for the identified channels.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelIds - The set of channel identifiers for the channels to resume normal operations on.

Raised Exceptions:

- Exception

`setServiceAttributeValue(sessionTrack : SessionTrack, attributeName : String, attributeValue : Object) : void`

Set an attribute's value in the ServiceContext for this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- attributeName - The identifier for the attribute whose value should be set.
- attributeValue - The value of the identified attribute.

Raised Exceptions:

- Exception

```
suspendChannels(sessionTrack : SessionTrack, channelIds : List<String>) : void
```

Suspend normal channel operations on the identified channels.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelIds - The list of channel identifiers for the channels to suspend normal operations on.

Raised Exceptions:

- Exception

```
updateChannel(sessionTrack : SessionTrack, channelId : String, updateContext : ChannelContext) : void
```

Update the configuration for the identified channel.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelId - The unique identifier of the channel to be updated.
- updateContext - The channel context containing the settings to be updated.

Raised Exceptions:

- Exception

BaseChannelServiceConnector

This interface extends the Base Service Connector and the Base Channel Service, thereby inheriting, and thus exposing, all methods from that service.

Public Operations

(Inherited from the Base Channel Service)

BaseChannelServiceStub

This interface represents a stub for the base channel service and extends the Base Service Stub and the Base Channel Service. It exposes all methods inherited from the Base Channel Service in addition to those methods listed here that are specific to the stub.

Public Operations

```
connect(sessionTrack : SessionTrack) : void
```

Connects the stub to its corresponding connector.

Arguments:

- sessionTrack - Contains the identifier for the session whose owner is attempting to connect this stub.

Raised Exceptions:

- Exception

`disconnect(sessionTrack : SessionTrack) : void`

Disconnect the stub from its corresponding connector.

Arguments:

- sessionTrack - Contains the identifier for the session whose owner is attempting to disconnect this stub.

Raised Exceptions:

- Exception

`resume(sessionTrack : SessionTrack) : void`

Resume forwarding method invocations to its corresponding connector.

Arguments:

- sessionTrack - Contains the identifier for the session whose owner is attempting to resume operations upon this stub.

Raised Exceptions:

- Exception

`suspend(sessionTrack : SessionTrack) : void`

Suspend the stub, keeping the connection open to the connector but not passing any method invocations to it until the stub is resumed.

Arguments:

- sessionTrack - Contains the identifier for the session whose owner is attempting to suspend operations upon this stub.

Raised Exceptions:

- Exception

ByteInputChannel

A byte specific input channel.

Public Operations

`read(bytes : byte[]) : int`

Read a set of bytes from the channel. This method returns the number of bytes read from the channel.

Arguments:

- bytes - The byte array to read the bytes from the channel into.

Raised Exceptions:

- `ChannelException`

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

Read a set of bytes from the channel beginning at the specified offset and continuing for the specified length. This method returns the number of bytes read from the channel.

Arguments:

- bytes - The byte array to read the bytes from the channel into.
- offset - The offset index for the bytes array. Will start filling the bytes array from this point.
- length - The number of bytes to read from the channel.

Raised Exceptions:

- `ChannelException`

ByteOutputChannel

A channel interface for writing bytes.

Public Operations

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

Write the given bytes to the output channel.

Arguments:

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

Raised Exceptions:

- `ChannelException`

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

Write the given bytes to the output channel using the specified parameters.

Arguments:

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

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

Raised Exceptions:

- [ChannelException](#)

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, guard technologies, or possibly routers such as Sarvega or Layer 7 boxes.

Public Operations

[getApplicationProtocolContext\(\)](#) : [ProtocolContext](#)

Retrieve the context containing the application level networking layer settings. These are expected to be strings that identify the protocol such as "information", "event", "byte", and "frame".

[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.

[getEndPointContext\(\)](#) : [EndPointContext](#)

Retrieve the [EndPointContext](#) that contains the URI for the endpoint of the associated channel.

[getSessionTrack\(\)](#) : [SessionTrack](#)

Retrieve the session track associated with this channel.

[getTransportProtocolContext\(\)](#) : [TransportProtocolContext](#)

Retrieve the context that describes the transport level networking layer section.

[setApplicationProtocolContext\(context : \[ProtocolContext\]\(#\)\)](#) : [void](#)

Set the context containing the application level networking layer settings.

Arguments:

- context - The context containing the application level networking layer settings.

[setChannelState\(state : \[ChannelState\]\(#\)\)](#) : [void](#)

Set the current state of the channel.

Arguments:

- state - The current state of the channel. Possible values are defined by the [ChannelState](#) enumeration.

[setEndPointContext\(context : \[EndPointContext\]\(#\)\)](#) : [void](#)

Set the [EndPointContext](#) that contains the URI for the endpoint of the associated channel.

Arguments:

- context - The [EndPointContext](#) that contains the URI for the endpoint of the associated channel.

`setSessionTrack(sessionTrack : SessionTrack) : void`

Set the session track associated with this channel.

Arguments:

- sessionTrack - The session track associated with this channel.

`setTransportProtocolContext(context : setTransportProtocolContext) : void`

Set the context defining the transport level network settings.

Arguments:

- context - The context defining the transport level network settings.

ChannelException

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

Public Operations

`getChannelContext() : ChannelContext`

Retrieve the [ChannelContext](#) that is related to the raised exception.

`getSessionTrack() : SessionTrack`

Retrieve the [SessionTrack](#) that is related to the raised exception.

ChannelServiceDescriptorContext

A service descriptor context for describing services with input channels.

Public Operations

`listInputChannelContexts() : List<ChannelContext>`

List the contexts describing this service's input channels.

ChannelState

This enumeration lists the possible states for a [Channel](#).

Public Fields

[ACTIVE](#)

Depending on whether the channel is an input or output channel, it is either listening for connections or is connected.

INACTIVE

Depending on whether the channel is an input or output channel, it is either disconnected or has stopped listening for connections.

NEW

The channel has been created but not yet opened or connected.

SUSPENDED

A channel that has been created and activated, but whose channel operations (read or write) have been temporarily suspended. What to do with attempted reads/writes is up to the implementation.

EndPointContext

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 required attributes for EndPointContext instances are: *uri* and *inBandConsumer*.

1. The *uri* 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.
2. 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.

Public Operations

`getHostAddress() : String`

Retrieve the string that contains the host location for this end point. For example, 192.168.2.13

`getHostPort() : int`

Retrieve the port for the end point to communicate on. For example, 2222

`getLocalPort() : int`

Retrieve the local port for the end point to communicate through. For example, 12121

`isInBandConsumer() : boolean`

Retrieve 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.

`setHostAddress(host : String) : void`

Set the string that contains the host location for this end point. For example, 192.168.2.13

Arguments:

- host - The string that contains the host location for this end point. For example, 192.168.2.13

`setHostPort(port : int) : void`

Set the port for the end point to communicate on. For example, 2222

Arguments:

- port - The port for the end point to communicate on. For example, 2222

`setInBandConsumerFlag(isInBand : boolean) : void`

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

Arguments:

- isInBand - The in-band consumer flag.

`setLocalPort(port : int) : void`

Set the local port for the end point to communicate through. For example, 12121

Arguments:

- port - The local port for the end point to communicate through. For example, 12121

Handler

The Handler class is responsible for handling exceptional cases during channel input and output operations.

Public Operations

`handleException(e : Exception) : void`

Handle the provided exception that was raised.

Arguments:

- e - The exception that was raised.

InputChannel<T>

An interface for input channels.

Public Operations

`close(sessionTrack : SessionTrack) : void`

Closes the listening port for this channel. Any connected output channels will be forcibly closed by the host. Invoking this method more than once should do nothing and not generate an exception.

Arguments:

- sessionTrack - The identifier(s) for the actor closing the channel.

Raised Exceptions:

- `ChannelException`

`getInputChannelContext() : InputChannelContext`

Retrieve the input channel context for this channel.

`getNumberActiveConnections() : int`

Retrieve the number of output channels connected to this input channel.

`getNumberAvailableConnections() : int`

Retrieve the number of connections left for output channels to connect to. This is computed by subtracting the number of active connections from the connection limit for the input channel. If no limit is set this method returns a negative value.

`isOpen() : boolean`

Returns true if this input channel is listening for connections from output channels, false otherwise.

`listConnections() : List<ChannelContext>`

Retrieve the list of channel contexts that describe the output channels currently connected to this input channel.

`open(sessionTrack : SessionTrack) : void`

Open the listening port for this channel and make the channel ready to accept connections from output channels. Invoking this method more than once should do nothing and not generate an exception.

Arguments:

- sessionTrack - The identifier(s) for the actor opening the channel.

Raised Exceptions:

- `ChannelException`

`read(reHandler : InputHandler<T>) : void`

Read data from this channel asynchronously using the given handler.

Arguments:

- reHandler - The handler responsible for processing the data received from the channel and any exceptions raised.

InputChannelContext

A channel context specifically for input channels.

Public Operations

`addConnection(context : ChannelContext) : String`

Add a channel context for a connected output channel.

Arguments:

- context - The context for the output channel that has connected to this input channel.

`getConnectionLimit() : int`

Retrieve the maximum number of allowed connections for this input channel. The default for this is the maximum number of connections allowed by the implementation and its supporting hardware.

`listConnections() : List<ChannelContext>`

List the current set of contexts for the output channels that are connected to this input channel.

`removeConnection(id : String) : void`

Remove the channel context for the identified output channel that has disconnected from this input channel.

Arguments:

- id - The identifier for the output channel that has disconnected from this input channel.

`setConnectionLimit(limit : int) : void`

Set the maximum number of allowed connections for this input channel. The default for this is the maximum number of connections allowed by the implementation and its supporting hardware. Setting this value to zero is equivalent to resetting it to the default value.

Arguments:

- limit - The maximum number of allowed connections for this input channel.

`InputHandler<T>`

This class handles the input received over input channels.

Public Operations

`handleObject(object T) : void`

Handle the received object of type T.

Arguments:

- object - The object to handle.

`OutputChannel<T>`

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: information, event, and bytes.

Public Operations

`connect(sessionTrack : SessionTrack) : void`

Connect this output channel to an input channel. Invoking this method more than once should do nothing and not generate an exception.

Arguments:

- sessionTrack - The identifier(s) for the actor connecting the channel.

Raised Exceptions:

- `ChannelException`

`disconnect(sessionTrack : SessionTrack) : void`

Disconnect this output channel from its associated input channel. Invoking this method more than once should do nothing and not generate an exception.

Arguments:

- sessionTrack - The identifier(s) for the actor disconnecting the channel.

Raised Exceptions:

- `ChannelException`

`flushQueue() : void`

Flush the current contents of the output queue.

`isConnected() : boolean`

Check if this output channel is connected to an associated input channel. Returns true if so, False otherwise.

`isQueueEmpty() : boolean`

Check if the current output queue is empty. Returns true if it is empty, False otherwise.

`writeAsync(data : T) : void`

Write the given data to the output channel in an asynchronous fashion.

Arguments:

- data - The data to be written to the output channel.

ProtocolContext

This context is intended to hold quality of service and other as yet undefined attributes related to a channel's network application layer protocol and mechanisms.

Public Operations

`getProtocolId() : String`

Retrieve the identifier for the application level protocol for this string.

`setProtocolId(protocolId : String) : void`

Set the identifier for the application level protocol for this string.

Arguments:

- `protocolId` - The identifier for the application level protocol for this string.

TransportProtocolContext

This context is intended to hold quality of service and other as yet undefined attributes related to a channel's network transport layer protocol and mechanisms.

Public Operations

`getMaxMessageSize() : int`

Retrieve the max message size supported by this protocol.

`isPersistConnection() : boolean`

Retrieve the flag telling whether or nor this protocol uses persistent connections.

`listMessageEncodings() : List<String>`

Retrieve the list of encoders to apply to this protocol.

`setMaxMessageSize(maxSize : int) : void`

Set the max message size supported by this protocol.

Arguments:

- `maxSize` - The max message size supported by this protocol.

`setMessageEncodings(encodings : List<String>) : void`

Set the list of encoders to apply to this protocol.

Arguments:

- `encodings` - The list of encoders to apply to this protocol.

`setPersistConnection(persist : boolean) : void`

Set the flag telling whether or not this protocol uses persistent connections.

Arguments:

- `persist` - The flag telling whether or nor this protocol uses persistent connections.

Expression

The expression group provides the interfaces, contexts, and supporting components that define the Phoenix architecture's support for describing expressions. The expression interfaces are:

- [ExpressionContext](#)
- [ExpressionProcessor](#)
- [ExpressionServiceContext](#)

ExpressionContext

This [Context](#) describes an expression to be used for brokering or query operations.

Public Operations

`getExpression() : String`

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.

`getExpressionType() : String`

The type of [expression](#) that this [Context](#) describes.

`setExpression(expression : String) : void`

Set the expression instance.

Arguments:

- expression - The expression instance.

`setExpressionType(expressionType : String) : void`

Set the type of expression.

Arguments:

- expressionType - The expression type identifier.

ExpressionProcessor

An interface for a generic Phoenix expression processor.

Public Operations

`addExpression(expressionContext : ExpressionContext, informationTypeNames : List<String>) : String`

Register a expression with this expression processor. This method returns the unique identifier for the registered expression.

Arguments:

- expressionContext - The context defining the expression to be registered.
- informationTypeNames - The list of information types that the given expression should be applied to.

Raised Exceptions:

- Exception - if an error occurs during the expression registration process.

`evaluate(objectToEvaluate : Object) : List<String>`

Evaluate an instance of data against the set of registered expressions. This method returns the List of subscription unique identifiers whose expressions the provided object matched.

Arguments:

- `objectToBroker` - The data instance to be evaluated.

Raised Exceptions:

- Exception - if an error occurs during evaluation.

`getSupportedExpressionType() : String`

Retrieve the supported expression type identifier.

`isRegistered(expressionId : String) : boolean`

Check if the specified expression has been registered.

Arguments:

- `expressionId` - The identifier for the expression.

Raised Exceptions:

- Exception - if an error occurs.

`listRegisteredExpressionIds() : List<String>`

Retrieve a listing of the set of identifiers for the currently registered expressions.

Raised Exceptions:

- Exception - if an error occurs.

`removeExpression(expressionId : String) : void`

Remove a registered expression from the set maintained by this processor.

Arguments:

- `expressionId` - The unique identifier for the expression to be removed.

Raised Exceptions:

- Exception - if an error occurs during evaluation.

`updateExpression(expressionId : String, expressionContext : ExpressionContext, informationTypeNames : List<String>) : void`

Update a expression without unregistering and re-registering or even pausing and resuming.

Arguments:

- `expressionId` - The identifier for the expression to be updated.

- `expressionContext` - The context containing the new settings for the expression.
- `informationTypeNames` - The list of information types that the given expression should be applied to.

Raised Exceptions:

- Exception - if an error occurs.

ExpressionServiceContext

An interface for service context's whose services utilize expression processors.

Public Operations

`listSupportedExpressionTypes() : List<String>`

Retrieve the list of expression types that this service supports.

Frame

The frame group contains the interfaces that define and support custom serialization, encapsulation, and stream sequencing (and dissemination) capability of the Phoenix architecture. The frame interfaces are:

- `Frame`
- `FrameInputChannel`
- `FrameOutputChannel`

Frame

The interface used to define frame segments of a stream. A frame is a non-serialized piece of data which has a custom byte-format. It is generally more difficult to manage than information, but higher-performance in its dissemination and processing.

Public Operations

`getHeader() : byte[]`

Gets the header portion of the frame.

`getData() : byte[]`

Gets the data portion of the frame.

`getTotalSize() : int`

Gets the total size of the frame include data, header and internals.

`getStreamId() : String`

Return the string that is used to identify the stream and to query the attributes of the stream.

`setStreamId( streamId : String) : void`

Set a string that is used to identify the stream and to query the attributes of the stream.

Arguments:

- `streamId` - The streamId of this frame

`getFrameNumber() : long`

Returns the frame number in relation to the stream.

`setHeader( header : String) : void`

Sets the header portion of the frame.

Arguments:

- `header` - The header portion of the frame.

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

Sets the data portion of the frame.

Arguments:

- `header` - The data portion of the frame.

`setData( frameNumber : long) : void`

Sets the frame number in relation to the stream.

Arguments:

- `frameNumber` - The frame number of this frame in the sequence of the entire stream.

FrameInputChannel

A frame-specific input channel.

Public Operations

`read() : Frame`

Read a single frame from the channel.

Raised Exceptions:

- `ChannelException`

`read(numberToRead : int) : List<Frame>`

Read a set of frames from the channel. This method will not return until the specified number of frames have been read.

Arguments:

- `numberToRead` - The number of frames to read from the channel.

Raised Exceptions:

- `ChannelException`

FrameOutputChannel

A frame-specific output channel.

Public Operations

`write(frame : Frame) : void`

Writes a frame to the Channel.

Arguments:

- `frame` - The frame to be written to the Channel.

Raised Exceptions:

- `ChannelException`

`write(frames : List<Frame>) : void`

Writes several frames to the Channel.

Arguments:

- `frames` - The frames to be written to the Channel.

Raised Exceptions:

- `ChannelException`

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:

- `Event`
- `EventContext`
- `EventInputChannel`
- `EventOutputChannel`
- `ExceptionEvent`
- `InformationDeliveryEvent`
- `InformationDeliveryType`

- `InformationEvent`
- `InformationTypeEvent`

Event

Events are used for passing messages or other pieces of unmanaged data between `actors`. These pieces of data are separate and distinct from information because they are service-level interactions that are not subject to the same management mechanisms as information instances.

Public Operations

`getBody() : Object`

Retrieve the body for this Event.

`getEventContext() : EventContext`

Retrieve the context for this Event.

`getEventId() : String`

Retrieve the identifier for this Event.

`getFiringActorId() : Object`

Retrieve the identifier for the actor who generated and fired this event.

`setBody(body : Object) : void`

Set the body for this Event.

Arguments:

- `body` - The body for this Event.

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.

Public Operations

`addMatchingRequestIds(requestIds : List<String>) : void`

Add a set of event notification request identifiers to this event.

Arguments:

- `requestIds` - The set of event notification request identifiers to add to this event.

`getMatchingRequestIds() : List<String>`

Retrieve the set of event notification request identifiers for this event, if any exist.

EventInputChannel

An event-specific input channel.

Public Operations

`read() : Event`

Read a single event from the channel.

Raised Exceptions:

- [ChannelException](#)

`read(numberToRead : int) : List<Event>`

Read a set of events from the channel. This method will not return until the specified number of events have been read.

Arguments:

- `numberToRead` - The number of events to read from the channel.

Raised Exceptions:

- [ChannelException](#)

EventOutputChannel

An event-specific output channel.

Public Operations

`write(event : Event) : void`

Write the specified [Event](#) to the [channel](#).

Arguments:

- `event` - The [Event](#) to be written to the channel.

Raised Exceptions:

- [ChannelException](#)

`write(events : List<Event>) : void`

Write the specified [Events](#) to the [channel](#).

Arguments:

- `events` - The [Events](#) to be written to the channel.

Raised Exceptions:

- `ChannelException`

ExceptionEvent

This `Event` is used to report `Exceptions` asynchronously.

Public Operations

`getException() : Exception`

Retrieve the `Exception` that is contained within this `Event`.

InformationDeliveryEvent

This specific sub-interface of `Event` provides a helper interface for handling delivery receipt `Events`.

Public Operations

`getDeliveryType() : InformationDeliveryType`

Retrieve the identifier for the type of delivery that this `Event` is describing.

`getOriginatingActors() : List<String>`

Retrieve the list of identifiers for the `actors` whom this delivery receipt `Event` applied to.

InformationDeliveryType

An enumeration defining the types of information delivery supported by the Phoenix Architecture.

Public Fields

`CONSUMER_RECEIPT`

A delivery type that identifies that a consumer has received the information.

`SUBMISSION_RECEIPT`

A delivery type that identifies that a Submission Service has received the information.

InformationEvent

An interface for events pertaining to Information.

Public Operations

`getInformationId() : String`

Retrieve the identifier for the information instance that this `Event` is associated with.

InformationTypeEvent

An [Event](#) that refers to a specific [Information Type](#). For example, this could be an event alerting an actor that another actor has subscribed to a specific information type.

Public Operations

`getInformationType() : InformationTypeContext`

Retrieve the context describing the [information type](#) associated with this event.

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 the following functional groups:

Information Service Interfaces

Information services directly manipulate information or information type definitions.

- [Dissemination](#)
- [Information Brokering](#)
- [Information Type Management](#)
- [Query](#)
- [Repository](#)
- [Submission](#)

Dissemination

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

- [DisseminationService](#)
- [DisseminationServiceConnector](#)
- [DisseminationServiceContext](#)
- [DisseminationServiceStub](#)

DisseminationService

The Dissemination Service extends the Base Channel Service and is responsible for accepting information and delivering it to consumers. Information forwarded to the Dissemination Service

from an actor is expected to contain a list of channel definitions as part of the resident information context. These definitions come in the form of fully defined channel contexts. The Dissemination Service may, if necessary, create channels based on these definitions and use them to deliver information to consumers.

Public Operations

`getChannelContexts(sessionTrack : SessionTrack, channelNames : List<String>) : List<ChannelContext>`

Retrieve the contexts describing the channels with the given names.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelNames - The names of the channels to retrieve the channel contexts for.

Raised Exceptions:

- Exception

`isManaged(sessionTrack : SessionTrack, channelName : String) : boolean`

Check if the identified channel is being managed by this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelName - The identifier for the channel.

Raised Exceptions:

- Exception

`listChannelNames(sessionTrack : SessionTrack) : List<String>`

Retrieve the set of names of the channels being managed by this Dissemination Service.

Arguments:

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

Raised Exceptions:

- Exception

Typical Use

This service is typically used in concert with single or multiple Information Brokering or Repository Services. These services perform brokering and query execution operations and forward information to be disseminated to the DS. This service may also be used in concert with other IM services, or as a standalone information dissemination capability, providing that the utilizing actors conform to the operational semantics of the DS.

Associated Diagrams

Use Cases

- UC 0000 Phoenix IM Capabilities

Activity Diagrams

- AD 0008 Information Dissemination

Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0006 Information Dissemination

Sequence Diagrams

- SQD 0005 Brokering (Information - via Information Channel (In-Band Delivery))

DisseminationServiceConnector

This interface extends the Dissemination Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Dissemination Service)

DisseminationServiceContext

This context holds the settings specific to the Dissemination Service and its operations.

Public Operations

`getMaxSupportedConsumers() : long`

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

`setMaxSupportedConsumers(max : long) : void`

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

Arguments:

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

DisseminationServiceStub

This interface extends the Dissemination Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Dissemination Service)

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](#)
- [InformationBrokeringService](#)
- [InformationBrokeringServiceConnector](#)
- [InformationBrokeringServiceContext](#)
- [InformationBrokeringServiceStub](#)

ConsumerList

Contains the list of in-band and out-of-band consumers for a subscription. In-band consumers are defined as those connected directly to one or more Phoenix services using channels. These consumers receive information directly from Phoenix services through the channels. Out-of-band consumers are those that are not connected directly to any Phoenix services through channels. If they receive information being managed by the Phoenix services they are receiving it through some mechanism other than a Phoenix channel.

Public Operations

`addInBandConsumers(consumers : List<String>) : void`

Adds a list of in-band consumers to the in-band list.

Arguments:

- consumers - The list of consumer identifiers to add to this ConsumerList.

`addOutOfBandConsumers(consumers : List<URI>) : void`

Add a set of out-of-band consumers to be added to the out-of-band list.

Arguments:

- consumers - The set of out-of-band consumer URI's to add to the out-of-band list.

`getInBandConsumers() : List<String>`

Retrieve the listing of in-band consumers. Returns a list of subscription ID's.

`getOutOfBandConsumers() : List<URI>`

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

InformationBrokeringService

This service extends the Base Channel Service and the Subscription Service interfaces and provides the information brokering capability of the Phoenix services. **Information** brokering is the act of matching submitted **information** with registered expressions. An **information** brokering **service** must support both the ability to forward brokered **information** for delivery and the ability to report the list of matching expressions 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 expression 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** for delivery to matching consumers.

Implicit brokering refers to the act of the brokering information received over information channels. The brokering **service** interface does not limit the operations that may be performed upon submitted **information** during the brokering process.

Brokering of information is done through the use of an expression processor implementation. The processor is part of the configuration defined in the Information Brokering Service's context.

Public Operations

```
dropSubscriptions(sessionTrack : SessionTrack, subscriptionIds : List<String>) : void
```

This method is used to drop subscriptions. This will remove the internal context for each identified expression. No return value for this method.

Arguments:

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

Raised Exceptions:

- Exception

```
getSubscriptionContexts(sessionTrack : SessionTrack, subscriptionIds : List<String>) : List<SubscriptionContext>
```

This method provides a way to access the context objects describing registered subscriptions. It returns the set of **SubscriptionContexts** that describe the registered subscriptions. These contexts contain the expression 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.

- subscriptionIds - The identifiers of the registered subscriptions whose descriptions are going to be retrieved.

Raised Exceptions:

- Exception

```
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 expression 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 expressions.
- consumerScope - The flag that identifies the types of `consumers` that the invoker is interested in. These are identified by the `ConsumerReport` enumeration.

Raised Exceptions:

- Exception

```
listRegisteredSubscriptionIds(sessionTrack : SessionTrack) : List<String>
```

List the identifiers for the set of currently registered subscriptions.

Arguments:

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

Raised Exceptions:

- Exception

```
registerSubscriptions(sessionTrack : SessionTrack, subscriptionContexts :
List<SubscriptionContext>) : List<String>
```

This method is used to register subscriptions. It returns the identifiers for the registered subscriptions.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- subscriptionContexts - The `Context` objects describing the subscriptions to be registered.

Raised Exceptions:

- Exception

```
resumeSubscriptions(sessionTrack : SessionTrack, subscriptionIds : List<String>)  
: void
```

Resume brokering operations over the identified subscriptions.

Arguments:

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

Raised Exceptions:

- Exception

```
suspendSubscriptions(sessionTrack : SessionTrack, subscriptionIds :  
List<String>) : void
```

Temporarily suspend brokering operations over the identified subscriptions.

Arguments:

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

Raised Exceptions:

- Exception

```
updateSubscription(sessionTrack : SessionTrack, subscriptionId : String,  
subscription : SubscriptionContext) : void
```

Update the identified expression.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- subscriptionId - The identifier for the subscription to be updated.
- subscription - The context containing the new settings for the subscription. It is up to each implementation to decide if this context is to contain all new values, with empty values assumed to be removed, or empty values assumed to be untouched.

Raised Exceptions:

- Exception

Typical Use

This service is typically used in concert with single or multiple Submission and Dissemination Services to provide a complete publish and subscribe system.

Associated Diagrams

Use Cases

- UC 0000 Phoenix IM Capabilities

Activity Diagrams

- AD 0005 Brokering (Information - via Control Interface)
- AD 0006 Brokering (Information - via Channels)

Class Diagrams

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

Sequence Diagrams

- SQD 0003 Brokering (Information - via Control Interface)
- SQD 0004 Brokering (Information - Expression Registration (In-Band Consumer))
- SQD 0005 Brokering (Information - via Information Channel (In-Band Delivery))
- SQD 0006 Brokering (Information - Expression Registration (Out-of-Band Producer))
- SQD 0007 Brokering (Information - via Information Channel (Out-of-Band Delivery))

InformationBrokeringServiceConnector

This interface extends the Information Brokering Service and Subscription Service interfaces, thereby exposing all of their methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Information Brokering Service and the Subscription Service)

InformationBrokeringServiceContext

This context contains any Information Brokering Service specific attributes.

Public Operations

None.

InformationBrokeringServiceStub

This interface extends the Information Brokering Service and the Subscription Service, thereby inheriting and exposing all of their methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Information Brokering Service and Subscription Service)

Information Type

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:

- [InformationTypeContext](#)
- [InformationTypeManagementService](#)
- [InformationTypeManagementServiceConnector](#)
- [InformationTypeManagementServiceStub](#)
- [Schema](#)
- [ValidationFailedException](#)

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.

Public Operations

`getInformationTypeName() : String`

Retrieve the unique identifier for this registered [information type](#).

`getMetadataSchema() : Schema`

Retrieve the object defining the structure of the [metadata](#) for this [information type](#).

`getPayloadSchema() : Schema`

Retrieve the object defining the structure of the `payload` for this `information type`.

`setInformationTypeName(typeName : String) : void`

Set the `information type` identifier.

Arguments:

- `typeName` - The `information type` identifier.

`setMetadataSchema(schema : Schema) : void`

Set the metadata `schema`.

Arguments:

- `schema` - The metadata `schema` object.

`setPayloadSchema(schema : Schema) : void`

Set the payload `schema`.

Arguments:

- `schema` - The payload `schema` object.

InformationTypeManagementService

The interface for `information type` management service will provide methods for registering, retrieving, and deleting `information type` definitions. This service interface extends the Base Channel Service interface.

Public Operations

`archiveTypeDefinitions(sessionTrack : SessionTrack, typeCtxs : List<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 we still need the description of said type so that we can query the offline archive and make sense 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:

- `Exception`

`createInformationTypes(sessionTrack : SessionTrack, typeCtxs : List<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:

- Exception

```
deleteTypeDefinitions(sessionTrack : SessionTrack, typeCtxs :  
List<InformationTypeContext>) : void
```

Delete 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:

- Exception

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

Get 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.

Raised Exceptions:

- Exception

```
listInformationTypeIdentifiers(sessionTrack : SessionTrack) : List<String>
```

Returns a list of IDs for all of the registered [information types](#).

Arguments:

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

Raised Exceptions:

- Exception

```
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:

- Exception

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).

Associated Diagrams

Use Cases

- [UC 0000 Phoenix IM Capabilities](#)
- [UC 0007 Information Type Management](#)

Activity Diagrams

- [AD 0011 Information Type Management](#)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)
- [CD 0009 Information Type Management](#)

Sequence Diagrams

- (none)

InformationTypeManagementServiceConnector

This interface extends the Information Type Management Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Information Type Management Service)

InformationTypeManagementServiceStub

This interface extends the Information Type Management Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Information Type Management Service)

Schema

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

Public Operations

`getDefinitionDocument() : Object`

Retrieve the definition document as an object.

`setDefinitionDocument(schemaDoc : Object) : void`

Set the definition document using a generic Object representation.

Arguments:

- `schemaDoc` - The definition document as a generic Object.

Raised Exceptions:

- `Exception`

`validate(data : Object) : boolean`

Validate an instance of data against the definition document.

Arguments:

- `data` - The data to be validated.

Raised Exceptions:

- `ValidationFailedException`

ValidationFailedException

An exception that provides a mechanism to track exactly what data failed the associated validation attempt.

Attributes

`invalidObject : Object`

The data that failed the validation attempt stored as a generic Object.

Public Operations

`getInvalidObject() : Object`

Retrieve the invalid object.

Query

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

- [DataStoreType](#)
- [InformationQueryContext](#)
- [QueryService](#)
- [QueryServiceConnector](#)
- [QueryServiceContext](#)
- [QueryServiceStub](#)

DataStoreType

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

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.

InformationQueryContext

A context used specifically for describe queries for information.

Public Operations

`addConsumerChannel(consumer : ChannelContext) : void`

Add a consumer channel to receive the results of this query.

Arguments:

- consumer - The context describing the consumer channel to receive the results of this query.

`addExpression(expression : ExpressionContext) : void`

Add an expression to this query.

Arguments:

- expression - The expression to be added to this query.

`addInformationTypeName(typeName : String) : void`

Add an information type name for this query to be applied to.

Arguments:

- infoTypeName - The information type name that this query will be applied to.

`getAllResultsReturnedTime() : long`

Retrieve the amount of time the inquisitor is willing to wait for its query to return all of its result set.

`getExecutionModeFlag() : int`

Retrieve the flag describing the query execution mode. Currently this is envisioned as: Synchronous (0) or Asynchronous (1).

`getFirstResultReturnedTime() : long`

Retrieve the amount of time the consumer is willing to listen for the first result that matches the associated expression.

`listConsumerChannels() : List<ChannelContext>`

Retrieve the entire list of consumer channels bound to this query.

`listExpressions() : List<ExpressionContext>`

Retrieve the list of expressions for this query.

`listInformationTypeNames() : List<String>`

Retrieve the information type names that this query applies to.

`setAllResultsReturnedTime(arrt : long) : void`

Set the amount of time the inquisitor is willing to wait for its query to return all of its result set.

Arguments:

- arrt - The amount of time the inquisitor is willing to wait for its query to return all of its result set.

`setExecutionModeFlag(modeFlag : int) : void`

Set the flag describing the query execution mode. Currently this is envisioned as: Synchronous (0) or Asynchronous (1).

Arguments:

- modeFlag - The flag describing the query execution mode. Currently this is envisioned as: Synchronous (0) or Asynchronous (1).

`setFirstResultReturnedTime(frft : long) : void`

Set the amount of time the consumer is willing to listen for the first result that matches the associated expression.

Arguments:

- frft - The amount of time the consumer is willing to listen for the first result that matches the associated expression.

QueryService

This service extends the Base Channel Service interface and provides an information retrieval capability. 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 leaving the door wide open for 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 information channels.

Public Operations

`cancelQuery(sessionTrack : SessionTrack, queryId : String) : boolean`

Cancel a currently executing query. Executing queries are defined as queries that have any processor cycles associated with them, i.e. a query is not done executing until all results (if any) are delivered to the Dissemination Service for delivery.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- queryId - The unique identifier for the query to be canceled.

Raised Exceptions:

- Exception

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

This method processes the specified query to satisfy some inquisitor's request for [information](#). Actual result sets are delivered via one or more information channels set up between the [consumer](#) and the query [service](#). When this method is invoked synchronously, the return value signals the inquisitor the estimated number of matching records that were found or that an error occurred while processing their query. A value of zero or greater is the estimated number of matching records while values less than zero are reserved for possible error flags. The returned value is an estimation because, when a query is executed against a live database more records that match the query could be

inserted while the query is executing or while the results are being returned to the inquisitor. When used asynchronously, this method does not return a value.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- queryCtx - The [Query Context](#) object that describes what [information](#) the inquisitor is searching for.

Raised Exceptions:

- Exception

```
getCounts(sessionTrack : SessionTrack, infoTypeNames : List<String>) : Map<String, Integer>
```

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.

Raised Exceptions:

- Exception

```
listActiveQueryIds(sessionTrack : SessionTrack) : List<String>
```

List the unique identifiers for the currently executing queries. Executing queries are defined as queries that have any processors cycles associated with them, i.e. a query is not done executing until all results (if any) are delivered to the Dissemination Service for delivery.

Arguments:

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

Raised Exceptions:

- Exception

Typical Use

This service is coordinated with one or more other Query Services to provide read access into data stores supported by Repository Services. Since the Repository Service extends this service interface that means a Query Service may end up talking directly to a Repository Service.

Associated Diagrams

Use Cases

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

Activity Diagrams

- AD 0013 Information Retrieval (Query)

Class Diagrams

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

Sequence Diagrams

- SQD 0009 Information Retrieval (Query – Synchronous)
- SQD 0010 Information Retrieval (Query – Asynchronous)

QueryServiceConnector

This interface extends the Query Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Query Service)

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*, *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 *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).
4. 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.

Public Operations

`getStoreType() : DataStoreType`

Retrieve 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.

`getDefaultAllResultsReturnedTime() : long`

Retrieve the default amount of time the Query Service is willing to let an inquisitor wait for their query to return all of its result set.

`getDefaultFirstResultReturnedTime() : long`

Retrieve the amount of time the Query Service is willing to let the consumer wait for the first result that matches their associated query.

`getDefaultMaxResultSetSize() : long`

Retrieve 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.

`setStoreType(type : DataStoreType) : void`

Set the type of data store that the associated query service is connected to.

Arguments:

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

`setDefaultAllResultsReturnedTime(arrt : long) : void`

Set the default amount of time the Query Service is willing to let an inquisitor wait for their query to return all of its result set.

Arguments:

- `arrt` - The default amount of time the Query Service is willing to let an inquisitor wait for their query to return all of its result set.

`setDefaultFirstResultReturnedTime(frft : long) : void`

Set the amount of time the Query Service is willing to let the consumer wait for the first result that matches their associated query.

Arguments:

- `frft` - The amount of time the Query Service is willing to let the consumer wait for the first result that matches their associated query.

`setDefaultMaxResultSetSize(maxSize : long) : void`

Set the default maximum result set size.

Arguments:

- `maxSize` - The default maximum result set size.

QueryServiceStub

This interface extends the Query Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Query Service)

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](#)
- [RepositoryServiceConnector](#)
- [RepositoryServiceContext](#)
- [RepositoryServiceStub](#)
- [TableType](#)

RepositoryService

The Repository Service extends the Query Service and provides the ability to manage information in 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 channels 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.

Public Operations

`archiveRecords(sessionTrack : SessionTrack, query : QueryContext) : int`

Archive records that match the provided query. If consumer channels are specified then write all records to be archived to the channels (remote archive), if no consumer channels are specified archive to disk in a repository implementation agnostic way (local archive). There are several possibilities that arise from using a QueryContext for this operation:

1. If an expression and information types are both specified the expression is applied to only the specified types and the matching records are archived.

2. If an expression is specified but information types are not the expression is applied to all supported types and the matching records are archived.
3. If no expression is provided but a set of information types are specified all records of the specified types are archived.
4. If neither an expression nor a set of information types are specified nothing happens and an exception is thrown. This is done because we specifically want to lock out the possibility of the default case being to archive all records for all supported types.

Returns the total number of records archived.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- query - The context that defines the subset of records to be archived.

Raised Exceptions:

- Exception

`deleteRecords(sessionTrack : SessionTrack, query : QueryContext) : int`

Delete records that match the provided query. Any consumer channels defined for the provided query are ignored. There are several possibilities that arise from using a QueryContext for this operation:

1. If an expression and information types are both specified the expression is applied to only the specified types and the matching records are deleted.
2. If an expression is specified but information types are not the expression is applied to all supported types and the matching records are deleted.
3. If no expression is provided but a set of information types are specified all records of the specified types are deleted.
4. If neither an expression nor a set of information types are specified nothing happens and an exception is thrown. This is done because we specifically want to lock out the possibility of the default case being to delete all records for all supported types.

Returns the total number of records deleted.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- query - The context that defines the subset of records to be deleted.

Raised Exceptions:

- Exception

`removeInformationStore(sessionTrack : SessionTrack, infoTypeName : String) : void`

This method will tell the service to permanently remove the data store for the identified information type. This method should fail if the repository is currently storing data for the specified information type (i.e. "end" method must be called first, before a "remove" call is executed).

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- infoTypeName - The name of the information type to remove the resident data store for.

Raised Exceptions:

- Exception

```
startStoringInformation(sessionTrack : SessionTrack, typeContext :  
InformationTypeContext) : void
```

This method causes the service to begin storing information of the identified type. If the type does not have a location (XML container, database table, etc) to store the information in, one will be created. If a location already exists, that existing location will be appended to. If the desired functionality is to create a new store for an already registered type, an actor should call the archive method, which will move the existing data store contents to another location.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- typeContext - The context that describes the information type to start information for. This context may be a partial or complete copy of the type definition. Complete copies are required if the repository service has never stored information of the identified type.

Raised Exceptions:

- Exception

```
stopStoringInformation(sessionTrack : SessionTrack, infoTypeName : String) :  
void
```

This method will tell the service to stop storing information of the identified type. Any further information instances of this type that are received will be ignored (dropped out of memory at processing time).

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- infoTypeName - The name of the information type to stop storing information instances for.

Raised Exceptions:

- Exception

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.

Associated Diagrams

Use Cases

- [UC 0000 Phoenix IM Capabilities](#)
- [UC 0005 Information Storage \(Persistence\)](#)

Activity Diagrams

- [AD 0009 Information Storage \(Persistence\)](#)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)
- [CD 0007 Information Storage \(Persistence\)](#)

Sequence Diagrams

- [SQD 0008 Information Storage \(Persistence\)](#)

RepositoryServiceConnector

This interface extends the Repository Service interface (and by extension the Query Service interface), thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Query Service and the Repository Service)

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 implementations 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.

Operations

`getMaxRepositorySize() : long`

Retrieve 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`

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

`getDefaultTableType() : TableType`

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

RepositoryServiceStub

This interface extends the Repository Service (and by extension the Query Service), thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Repository Service and Query Service)

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](#).

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.

Submission

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

- [SubmissionService](#)
- [SubmissionServiceConnector](#)
- [SubmissionServiceContext](#)
- [SubmissionServiceStub](#)

SubmissionService

The Submission Service extends the Base Channel Service and accepts data from [producers](#), converts it into managed [information](#), and forwards it to other IM [services](#) as required. This process uses channels internal to the Submission Service and, because of this, there is no 'submit' method defined in the [service](#)'s control interface.

This [service](#) receives [information](#) via channels and may involve converting accepted data into the Phoenix architecture's supported format for managed [information](#). 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). 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 the implementation.

Public Operations

(none)

Typical Use

The Submission Service is the typical entry point for information to be disseminated. As such, the SS is used in conjunction with any number of other Phoenix information services including the Information Type Management, Information Brokering, Dissemination, and Repository Services. The SS may also be coordinated with one or more Session Management and Authorization Services to support session monitoring and authorization operations.

Associated Diagrams

Use Cases

- UC 0000 Phoenix IM Capabilities
- UC 0001 Information Submission

Activity Diagrams

- AD 0001 Information Submission

Class Diagrams

- CD 0000 Phoenix IM Services
- CD 0001 Information Submission

Sequence Diagrams

- SQD 0001 Information Submission
- SQD 0002 Information Submission (Submission ACK via ENS)

SubmissionServiceConnector

This interface extends the Submission Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Submission Service)

SubmissionServiceContext

The Submission Service Context any values or entities of specific interest or importance to the Submission Service.

Public Operations

None.

SubmissionServiceStub

This interface extends the Submission Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Submission Service)

Utility Service Interfaces

Utility services provide capabilities that do not directly manipulate information instances, such as session management or service brokering. They form both the backbone infrastructure of the Phoenix architecture and provide additional service capabilities that extend the usefulness of the IM system as a whole.

- [Client](#)
- [Event Notification](#)
- [Filter](#)
- [Information Discovery](#)
- [Security](#)
- [Service Brokering](#)
- [Session Management](#)
- [Subscription](#)

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](#)
- [ClientRuntimeServiceConnector](#)
- [ClientRuntimeServiceContext](#)
- [ClientRuntimeServiceStub](#)

ClientRuntimeService

The Client Runtime Service extends the Base Channel Service and 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.

Public Operations

`createPublisherChannel`

```
createPublisherChannel(handler : Handler) : String
```

This method creates a new connection to the Submission Service. By allowing an actor to specify the callback object for the channel to use, the actor is able to be made aware of communication failures. This method returns the identifier for the created channel.

Arguments:

- handler - The exception handler for this channel.

Raised Exceptions:

- Exception

`publish(information : Information, waitTimeInMs : long) : void`

This method is for blocking publications. If delivery receipts are requested this method will block until all receipts are returned. If the wait time is expended without return of the expected delivery receipts an exception is thrown.

Arguments:

- information - The information instance to be published.
- waitTimeInMs - The total time to publish the information instance and to wait for all delivery receipts, in milliseconds.

Raised Exceptions:

- Exception

`publishAsync(information : Information) : void`

This is a non-blocking publish method that is used in a fire-and-forget manner. The invoker does not care about neither the return value from the operation nor any raised exceptions. Fire and forget assumes that delivery receipts will be ignored if the corresponding flags were set at publication time.

Arguments:

- information - The information instance to be published.

Raised Exceptions:

- Exception

`publishAsync(information : Information, handler : Handler< ? >) : void`

A non-blocking publish method that provides a handler interface to handle exceptions.

Arguments:

- information - The information instance to be published.
- handler - The handler to process any return value or raised exception. This handler is expected to handle any event notification operations associated with delivery receipts.

Raised Exceptions:

- Exception

```
subscribe(expressions : String[], expressionTypes : String[],
informationTypeNames : List<String>) : void
```

This subscribe method is used to setup a subscription for information, but not the consumption of matching information. For example, it may be used to setup a subscription on the behalf of another actor. This method will generate consumer delivery receipt events as required, depending on service configuration.

Arguments:

- expressions - The expressions to match information instances against.
- expressionTypes - The types of expressions. The entries in this array are expected to map to entries in the expressions array on a 1-to-1 basis.
- informationTypeNames - The list of information type names that this subscription will be applied to. If null or empty the subscription will be applied to all information types known to the Information Brokering Service it is registered with.

Raised Exceptions:

- Exception

```
subscribeAsync(expressions : String[], expressionTypes : String[],
informationTypeNames : List<String>, resultsHandler : InputHandler<Information>) :
String
```

This method is used to subscribe to information and setup the handler that will process matching results as they are delivered to the consumer.

Arguments:

- expressions - The expressions to match information instances against.
- expressionTypes - The types of expressions. The entries in this array are expected to map to entries in the expressions array on a 1-to-1 basis.
- informationTypeNames - The list of information type names that this subscription will be applied to. If null or empty the subscription will be applied to all information types known to the Information Brokering Service it is registered with.
- resultsHandler - The handler for processing matching information instances as they are delivered to the consumer. This handler is expected to generate any required delivery receipts.

Raised Exceptions:

- Exception

```
query(expression : String, expressionType : String, informationTypeNames :
List<String>) : List<Information>
```

Execute a synchronous query in the traditional manner, where the return of this method is the actual result set of the query.

Arguments:

- expression - The query expression to execute.
- expressionType - The type of query expression, i.e. "SQL" or "XQuery".
- informationTypeNames - The names of the information types that this query should be executed over. If null or empty the expression will be executed over the set of types that the executing Query Service is aware of.

Raised Exceptions:

- Exception

```
queryAsync(expression : String, expressionType : String, informationTypeNames : List<String>) : void
```

This method performs a fire-and-forget query execution where the actual consumers of the query result set are determined by another mechanism.

Arguments:

- expression - The query expression to execute.
- expressionType - The type of query expression, i.e. "SQL" or "XQuery".
- informationTypeNames - The names of the information types that this query should be executed over. If null or empty the expression will be executed over the set of types that the executing Query Service is aware of.

Raised Exceptions:

- Exception

```
queryAsync(expression : String, expressionType : String, informationTypeNames : List<String>, controlHandler : Handler, resultsHandler : InputHandler<Information>) : String
```

Execute a query in asynchronous fashion, delivering the result set to the invoker.

Arguments:

- expression - The query expression to execute.
- expressionType - The type of query expression, i.e. "SQL" or "XQuery".
- informationTypeNames - The names of the information types that this query should be executed over. If null or empty the expression will be executed over the set of types that the executing Query Service is aware of.
- controlHandler - The handler for control interactions such as checking query status or canceling the query.
- resultsHandler - The handler for processing the query result set as it is delivered.

Raised Exceptions:

- Exception

Typical Use

This [service](#) is envisioned to be a policy enforcement point for Quality of Service (QoS) and/or security applications. It will also function as a proxy for clients who do not wish to implement complete Phoenix clients within their code.

Associated Diagrams

Use Cases

- [UC 0000 Phoenix IM Capabilities](#)
- [UC 0012 Client Reach-Back](#)

Activity Diagrams

- [0018 Client Reach-Back \(Local Policy Capture & Enforcement\)](#)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)
- [CD 0012 Client Reach-Back](#)

Sequence Diagrams

- None.

ClientRuntimeServiceConnector

This interface extends the Client Runtime Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Client Runtime Service)

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 we notionally call 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.

Public Operations

```
addActiveQuery(id : String) : void
```

Add an active query to this client's state.

Arguments:

- id - The identifier for the active query.

`addRegisteredSubscription(id : String) : void`

Add a subscription to this client's state.

Arguments:

- id - The identifier for the registered subscription.

`getActiveQueryIds() : List<String>`

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

`getRegisteredSubscriptionIds() : List<String>`

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

`removeActiveQuery(id : String) : void`

Remove a query from this client's state.

Arguments:

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

`removeRegisteredSubscription(id : String) : void`

Remove a subscription from this client's state.

Arguments:

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

ClientRuntimeServiceStub

This interface extends the Client Runtime Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Client Runtime Service)

Event Notification

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

- [EventDescriptorContext](#)
- [EventNotificationRequestContext](#)

- [EventNotificationService](#)
- [EventNotificationServiceConnector](#)
- [EventNotificationServiceContext](#)
- [EventNotificationServiceStub](#)

EventDescriptorContext

A context for describing a registered event.

Public Operations

`getRegistrationId() : String`

Retrieve the registration ID for the described Event class.

`getDescription() : String`

Retrieve the human readable and understandable description for the described Event class.

EventNotificationRequestContext

A context for registering for event notifications.

Public Operations

`addConsumerChannel(channel : ChannelContext) : void`

Add a consumer channel context to a specific subset of events.

Arguments:

- channel - The channel context

`addEventDescriptorId(descriptorId : Object) : void`

Add an Event class's registration ID to limit this request to a specific sub-set of events.

Arguments:

- descriptorId - The String containing the descriptor ID.

`addFiringActorId(firingActorId : Object) : void`

Add a specific actor's ID to limit this request to a specific sub-set of actors.

Arguments:

- firingActorId - The String containing the ID for the firing actor.

`getConsumerChannels() : List<ChannelContext>`

Retrieve the set of consumer channel contexts.

`getEventDescriptorIds() : List<String>`

Retrieve the set of registration IDs for the events this request is limited to.

`getFiringActorIds() : List<Object>`

Retrieve the set of actor IDs for the actors this request is limited to.

`removeConsumerChannel(contextId : String) : void`

Removes a channel context based on the ID.

Arguments:

- contextId - The ID of the channel context that is to be removed.

`removeEventDescriptorId(descriptorId : Object) : void`

Remove an event class's registration ID from this request.

Arguments:

- descriptorId - The String denoting the descriptor ID to be removed.

`removeFiringActorId(firingActorId : Object) : void`

Remove an actor ID from this request.

Arguments:

- firingActorId - The String denoting the actor ID to be removed.

EventNotificationService

This service extends the Base Channel Service and provides a central [event](#) delivery capability. The logic describing when to fire an [event](#) and the construction of said [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 said [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](#).

Public Operations

`deleteEventDescriptor(sessionTrack : SessionTrack, eventInstance : descriptorId : String) : void`

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- descriptorId - The identifier of the event descriptor that is to be deleted.

Raised Exceptions:

- Exception

```
dropNotificationRequests( sessionTrack : SessionTrack, requestIds :  
List<String>) : void
```

Drop the identified notification requests.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- requestIds - The list of identifiers for the notification requests to be dropped.

Raised Exceptions:

- Exception

```
getEventRegistrationId(sessionTrack : SessionTrack, eventInstance : Event) :  
String
```

Retrieve the registration identifier for a specific event class, if registered. If event class is not registered, throw an exception.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- eventInstance - A sample instance of the event to look up the registration identifier for.

Raised Exceptions:

- Exception

```
listRegisteredEventDescriptors(sessionTrack : SessionTrack) :  
List<EventDescriptorContext>
```

Arguments:

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

Raised Exceptions:

- Exception

```
notify(sessionTrack : SessionTrack, events : List<Event>) : void
```

Notify consumers of matching requests of the given `events`.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- events - The `events` that have been fired.

Raised Exceptions:

- Exception

```
registerEventDescriptor(sessionTrack : SessionTrack, eventInstance : Event,  
description : String) : String
```

Arguments:

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

Raised Exceptions:

- Exception

```
registerNotificationRequest(sessionTrack : SessionTrack, requestCtxs :  
List<EventNotificationRequestContext>) : List<String>
```

Register a request to be notified of specific **events** as they occur within other **services**. This method returns a list of identifiers for the specific Notification Requests.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- requestCtxs - The list of contexts describing the event notification requests to be registered with the service.

Raised Exceptions:

- Exception

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**.

Associated Diagrams

Use Cases

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

Activity Diagrams

- [AD 0010 Event Notification](#)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)
- [CD 0008 Event Notification](#)

Sequence Diagrams

- [SQD 0002 Information Submission \(Submission ACK via ENS\)](#)
- [SQD 0007 Brokering \(Information – via Information Channel \(Out-of-Band Delivery\)\)](#)

EventNotificationServiceConnector

This interface extends the Event Notification Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Event Notification Service)

EventNotificationServiceContext

A context specifically for the [Event Notification Service](#).

Public Operations

`getRegisteredEventDescriptors() : List<EventDescriptorContext>`

Retrieve a listing of the registered event descriptors supported by the associated [Event Notification Service](#).

`setRegisteredEventDescriptors(eventDescriptors : List<EventDescriptorContext>) : void`

Set the listing of registered event descriptors supported by the associated [Event Notification Service](#).

Arguments:

- eventDescriptors - The listing of registered event descriptors supported by the associated [Event Notification Service](#).

EventNotificationServiceStub

This interface extends the Event Notification Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Event Notification Service)

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](#)
- [FilterChain](#)
- [FilterChainContext](#)
- [FilterContext](#)
- [FilterManagementService](#)
- [FilterManagementServiceConnector](#)
- [FilterManagementServiceStub](#)

Filter

Filters may be used by the Phoenix architecture to provide a mechanism by which data or information may be transformed, enhanced, degraded, or processed in some way as it flows through the IM system. Filters may be applied within a given service, component, or channel. Filters may be used to apply security, QoS, or other policies. Filters are envisioned to support any type of operation, from compression and decompression to removing sensitive data before transmission.

Filters are created by actors and registered with the [Filter Management Service](#).

Public Operations

`activate() : void`

Activate the mechanisms used by the filtering logic for this filter.

Raised Exceptions:

- Exception - if an error occurs during activation.

`deactivate() : void`

Deactivate the mechanisms used by the filtering logic for this filter.

Raised Exceptions:

- Exception - if an error occurs during deactivation.

`filter(object : Object) : Object`

Take the given Object and perform some type of filtering logic and operation upon it. It returns the filtered Object.

Arguments:

- object - The Object to be filtered.

Raised Exceptions:

- Exception - if an error occurs during filtering.

`isObjectModified() : boolean`

Check if this filter modifies filtered objects. Return true if it does, False otherwise.

`filter(object : Object) : Object`

Take the given Object and perform some type of filtering logic and operation upon it. It returns the filtered Object.

Arguments:

- object - The Object to be filtered.

Package Operations

`setNextFilter(filter : Filter) : void`

Set the next filter in the chain. This operation is used at filtering time to facilitate the automatic execution of the next filter in the chain, if any exists.

- filter - The next filter in the chain.

FilterChain

A filter chain tracks and manages [Filters](#) that have been applied to an operation.

Public Operations

`activate() : void`

Activate the mechanisms used by the filtering logic for this filter chain.

Raised Exceptions:

- Exception - if an error occurs during activation.

`deactivate() : void`

Deactivate the mechanisms used by the filtering logic for this filter chain.

Raised Exceptions:

- Exception - if an error occurs during deactivation.

`filter(object : Object) : Object`

Take the given Object and perform some type of filtering logic and operation upon it. It returns the filtered Object.

Arguments:

- object - The Object to be filtered.

Raised Exceptions:

- Exception - if an error occurs during filtering.

`getFilterCount() : int`

Returns the number of filters in the chain.

`updateFilter(filterId : String, attributesToUpdate : Map<String, Object>) : 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.

`isObjectModified() : boolean`

Check if any of the filters in this chain modifies the filtered objects. Return true if one or more filters in the chain do modify the objects, False otherwise.

Package Operations

`appendFilter(filter : Filter) : int`

Insert a Filter into the chain of Filters at the end of the chain. This method returns the index of the appended filter.

Arguments:

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

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

Insert 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 to insert the new Filter at.

`removeFilter(filterId : String) : void`

Remove the `Filter` with the given identifier.

Arguments:

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

`removeFilter(index : int) : void`

Remove the `Filter` at the given index in the filtering chain.

Arguments:

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

FilterChainContext

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

Public Operations

`getFilterNameList() : List<String>`

Retrieve the names of the filters to be used to make the filter chain.

`getInputType() : String`

Retrieve the identifier for the type of Object being passed to the first Filter as input.

`getOutputType() : String`

Retrieve the identifier for the type of Object being returned by the last Filter as output.

`setFilterNameList(filterNames : List<String>) : void`

Set the names of the filters to be used to make the filter chain.

- `filterNames` - The names of the filters to be used to make the filter chain.

`setInputType(inputType : String) : void`

Set the identifier for the type of Object being passed to the first Filter as input.

- `inputType` - The identifier for the type of Object being passed to the first Filter as input.

`setOutputType(outputType : String) : void`

Set the identifier for the type of Object being returned by the last Filter as output.

- `outputType` - The identifier for the type of Object being returned by the last Filter as output.

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.

Public Operations

`getInputType() : String`

Retrieve the identifier for the type of Object being passed to the Filter as input.

`getModifiesObject() : boolean`

Get the flag that tells whether or not the associated filter modifies filtered objects.

`getOutputType() : String`

Retrieve the identifier for the type of Object being returned by the Filter as output.

`setInputType(inputType : String) : void`

Set the identifier for the type of Object being passed to the Filter as input.

- `inputType` - The identifier for the type of Object being passed to the Filter as input.

`setModifiesObject(modifiesObject : boolean) : void`

Set the flag that tells whether or not the associated filter modifies filtered objects.

- `modifiesObject` - The flag that tells whether or not the associated filter modifies filtered objects.

`setOutputType(outputType : String) : void`

Set the identifier for the type of Object being returned by the Filter as output.

- `outputType` - The identifier for the type of Object being returned by the Filter as output.

FilterManagementService

This service extends the Base Channel Service and is responsible for maintaining the registry of filters to be used by actors and for creating orchestrated chains of these filters for use by actors during filtering operations.

Public Operations

`createFilterChain(sessionTrack : SessionTrack, orchestrationCtx : FilterChainContext) : FilterChain`

This method creates an orchestrated chain of filters based on the provided description of the filtering operations to be undertaken.

Arguments:

- `sessionTrack` - The pedigree of the invokers for this method.
- `orchestrationCtx` - The context that contains the description of the filtering operations to be undertaken by the created chain of orchestrated filters.

Raised Exceptions:

- Exception

`dropFilters(sessionTrack : SessionTrack, filterIds : List<String>) : void`

Drop the identified filters from the registry.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- filterIds - The identifiers of the filters to be removed from the registry.

Raised Exceptions:

- Exception

`listRegisteredFilters() : List<FilterContext>`

Retrieve the list of contexts that describe the registered filters.

`registerFilters(sessionTrack : SessionTrack, filters : List<Filter>) : void`

Add the provided filters to the registry.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- filters - The filters to be added to the registry.

Raised Exceptions:

- Exception

Typical Use

It is envisioned that this service will be used by actors to register filter implementations in order to facilitate the sharing of information shaping capabilities. This registry also enables the discovery of information shaping capabilities. This service is also used by actors to create customized filter chains for filtering data. An actor uses the [FilterChainContext](#) to describe the filtering operations they wish to occur. This service accepts that context, looks up the corresponding filter instances, checks the proposed orchestration chain for validity, and returns the formed orchestration chain if everything checks out. If a filter is not found or a validation check fails, an exception is thrown.

Associated Diagrams

Use Cases

- (None.)

Activity Diagrams

- (None.)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)

- (None.)

FilterManagementServiceConnector

This interface extends the Filter Management Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Filter Management Service)

FilterManagementServiceStub

This interface extends the Filter Management Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Filter Management Service)

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](#)
- [InformationDiscoveryServiceConnector](#)
- [InformationDiscoveryServiceContext](#)
- [InformationDiscoveryServiceStub](#)

InformationDiscoveryService

The Information Discovery Service (IDS) extends the Base Channel Service and provides a simple interface for "discovering" what information types are known to the Information Management (IM) services and what services are supporting which types.

Public Operations

```
getSupportingServices(sessionTrack : SessionTrack, informationTypeName : String,  
svcTypeExpression : ExpressionContext) : List<ServiceDescriptorContext>
```

Retrieve a set of contexts that describe the services that support the identified information type and their available control channels. The returned contexts should describe where to find the service and how to connect to it.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- informationTypeName - The information type to find supporting services for.
- svcTypeExpression - An expression to be applied to the service types of the services found to be supporting the provided information type name.

Raised Exceptions:

- Exception

```
getTypeDefinitions(sessionTrack : SessionTrack, expression : ExpressionContext)  
: List<InformationTypeContext>
```

Retrieve the type definitions whose information type names match the provided expression.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- expression - The expression to be applied to the known information type names.

Raised Exceptions:

- Exception

Typical Use

This service is used in conjunction with other Phoenix services to provide a simple interface for finding information and the services processing it.

Associated Diagrams

Use Cases

- [UC 0000 Phoenix IM Capabilities](#)
- [UC 0003 Discovery \(Information\)](#)

Activity Diagrams

- [AD 0004 Discovery \(Information\)](#)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)
- [CD 0002 Discovery \(Information\)](#)

Sequence Diagrams

- None.

InformationDiscoveryServiceConnector

This interface extends the Information Discovery Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Information Discovery Service)

InformationDiscoveryServiceContext

A service context specific to the Information Discovery Service.

Public Operations

None.

InformationDiscoveryServiceStub

This interface extends the Information Discovery Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Information Discovery Service)

Security

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

- [AuthorizationContext](#)
- [AuthorizationService](#)
- [AuthorizationServiceConnector](#)
- [AuthorizationResponse](#)
- [AuthorizationResponseType](#)
- [AuthorizationServiceStub](#)
- [MultipleAuthorizationContext](#)

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](#).

Public Operations

`getAction() : Object`

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

Example Actions

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

`getSessionId() : Object`

Get the session identifier of the actor who will be performing the action being authorized.

`getTarget() : Object`

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

Example Targets

- DATA_CHANNEL : [BaseChannel](#)
- INFORMATION : [Information](#)

- SERVICE : [BaseService](#)
- SESSION : [SessionContext](#)

[setAction\(action : Object\) : void](#)

Set the action that has been requested.

Arguments:

- action - The action that has been requested.

[setSessionId\(sessionId : Object\) : void](#)

Set the session identifier of the actor who will be performing the action being authorized.

Arguments:

- sessionId - The session identifier of the actor who will be performing the action being authorized.

[setTarget\(target : Object\) : void](#)

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

Arguments:

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

[AuthorizationService](#)

This service extends the Base Channel Service, thereby inheriting all its methods.

Actions within a SOA-based environment may be dependent upon some form of security policy or restriction. This security authorization capability should be designed such that it may be a single point of execution or a fully distributed and 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.

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.

Raised Exceptions:

- Exception

`updatePolicy(sessionTrack : SessionTrack, policy : Object) : void`

Update the policy that this Authorization Service reasons upon.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- policy - An Object to update the policy with. What this is and how it is utilized are up to the Authorization Service implementations.

Raised Exceptions:

- Exception

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.

Associated Diagrams

Use Cases

- [UC 0000 Phoenix IM Capabilities](#)
- [UC 0010 Authorization](#)

Activity Diagrams

- [AD 0014 Authorization](#)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)
- [CD 0013 Authorization](#)

Sequence Diagrams

- [SQD 0011 Session Management \(Creation & Destruction\)](#)

[AuthorizationServiceConnector](#)

This interface extends the Authorization Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Authorization Service)

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.

Public Operations

`getReasons() : List<String>`

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

`getResponse() : AuthorizationResponseType`

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

Package Operations

`addReason(reason : String) : void`

Add a reason to this response.

Arguments:

- `reason` - The reason to add to this response.

`setResponse(responseType : AuthorizationResponseType) : void`

Set the response type.

Arguments:

- `responseType` - The type of response. Possible values are defined by the `AuthorizationResponseType` enumeration.

AuthorizationResponseType

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

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.

AuthorizationServiceStub

This interface extends the [Authorization Service](#), thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the [Authorization Service](#))

MultipleAuthorizationContext

This context represents a container object for a set of operations to be authorized as a single unit. This means that if one operation is not authorized or is indeterminate then the whole operation set is marked as such.

Public Operations

```
addAuthorizationRequest(action : Object, target : Object, sessionId : Object) :  
void
```

Add an authorization request to the set that represents the chain of operations to be authorized as a single unit.

Arguments:

- action - The action for a specific authorization request in the chain.
- target - The target for a specific authorization request in the chain.
- sessionId - The session identifier for the actor who will be performing the action being authorized.

```
getAuthorizationRequests() List<AuthorizationContext>
```

Retrieve the set of authorization requests that represents the chain of operations to be authorized.

Service Brokering

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

- [ServiceBrokeringQueryContext](#)

- [ServiceBrokeringService](#)
- [ServiceBrokeringServiceConnector](#)
- [ServiceBrokeringServiceContext](#)
- [ServiceBrokeringServiceStub](#)
- [ServiceDescriptorSchema](#)

ServiceBrokeringQueryContext

A context used specifically for Service Brokering operations.

Public Operations

`getExpression() : ExpressionContext`

Retrieve the expression for this query.

`listStubProtocols() : List<String>`

List the control channel protocols whose corresponding stubs this query context is requesting.

`setExpression(expression : ExpressionContext) : void`

Set the expression for this query.

Arguments:

- `expression` - The expression for this query.

`setStubProtocols(protocols : List<String>) : void`

Set the control channel protocols whose corresponding stubs this query context is requesting.

Arguments:

- `protocols` - The control channel protocols whose corresponding stubs this query context is requesting.

ServiceBrokeringService

This service extends the Base Channel Service and is responsible for maintaining a registry of service descriptions associated with service control stubs. Control stubs are the entities used by Phoenix actors to interact with services. Service descriptions may include fields such as what specific information types a particular service supports or where it is physically located. The control stub returned by a service brokering operation is used to invoke control methods for such operations as setting up information channels.

Public Operations

```
brokerForServices(sessionTrack : SessionTrack, constraints :  
ServiceBrokeringQueryContext) : List<ServiceDescriptorContext>
```

Broker for a set of [services](#) that satisfy the specified constraints. This method returns a list of matching service descriptors.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- constraints - The defined criteria used to broker for matching [services](#).

Raised Exceptions:

- Exception

```
getServiceDescriptorSchema(sessionTrack : SessionTrack, serviceType : String) :  
ServiceDescriptorSchema
```

Retrieve the schema being used by the Service Brokering Service to describe the contents of a service descriptor for a specific service type.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- serviceType - The identifier of the descriptor schema to retrieve.

Raised Exceptions:

- Exception

```
registerService(sessionTrack : SessionTrack, svcDescriptorCtx :  
ServiceDescriptorContext) : void
```

Register the given description for the specified [service](#).

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- svcDescriptorCtx - The [Context](#) object containing the description of the [service](#) to be registered.

Raised Exceptions:

- Exception

Typical Use

This service provides a central registry for Phoenix services. It may be used in conjunction with any or all of the other Phoenix services.

Associated Diagrams

Use Cases

- [UC 0000 Phoenix IM Capabilities](#)
- [UC 0004 Brokering \(Service\)](#)

Activity Diagrams

- [AD 0007 Brokering \(Service\)](#)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)
- [CD 0005 Brokering \(Service\)](#)

Sequence Diagrams

- None.

ServiceBrokeringServiceConnector

This interface extends the Service Brokering Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Service Brokering Service)

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.

Public Operations

`addRegisteredService(ctx : ServiceContext) : void`

Add a service to the list of registered services.

Arguments:

- `ctx` - The [ServiceContext](#) that describes the service that is being registered.

`getRegisteredServices() : List<ServiceContext>`

Returns the list of the currently registered [ServiceContexts](#).

`removeRegisteredService(serviceId : String) : void`

Remove the specified service from the list of registered services.

Arguments:

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

ServiceBrokeringServiceStub

This interface extends the Service Brokering Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Service Brokering Service)

ServiceDescriptorSchema

An interface that defines a container object for the schemas used by the Service Brokering Service to describe Phoenix services. These schemas are deliberately not tied to any specific implementing technology to uphold the abstract nature of this architecture.

Public Operations

`getDefinitionDocument() : Object`

Retrieve the actual schema from this container object that represents the description for a specific type of service.

Package Operations

`setDefinitionDocument(definitionDoc : Object) : void`

Set the actual schema for this container object that represents the description for a specific type of service.

- `definitionDoc` - The actual schema for this container object that represents the description for a specific type of service.

Session Management

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

- [SessionManagementService](#)
- [SessionManagementServiceConnector](#)
- [SessionManagementServiceContext](#)
- [SessionManagementServiceStub](#)

SessionManagementService

This [service](#) extends the Base Channel Service and 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.

Public Operations

`createSession(actorCtx : ActorContext, brokerBack : boolean) : Object`

Create and maintain [session](#) state reflecting an [actor](#)'s interactions with the IM [services](#). Returns the identifier for the [session](#) that was created.

Arguments:

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

Raised Exceptions:

- Exception

`destroySession(sessionTrack : SessionTrack, sessionId : Object) : void`

Destroy the specified [session](#).

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- sessionId - The identifier for the session to be destroyed.

Raised Exceptions:

- Exception

`getDistinctActors(sessionTrack : SessionTrack) : List<ActorContext>`

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

Arguments:

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

Raised Exceptions:

- Exception

```
getSessionAttributeValue(sessionTrack : SessionTrack, sessionId : Object,
attributeName : Object) : void
```

Retrieve a specific attribute for an identified `session`.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- sessionId - The identifier of the `session` whose `Session Context` is going to be modified.
- attributeName - The name of the `session` attribute to be set.
- attributeValue - The new value for the specified `session` attribute.

Raised Exceptions:

- Exception

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

This method retrieves the `SessionContext` for the given `session` identifier. 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.

Raised Exceptions:

- Exception

```
getSessionContexts(sessionTrack : SessionTrack, criteria : Object) :
List<SessionContext>
```

Retrieve a set of sessions that share some criteria.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- criteria - The search criteria that all returned session contexts must match. This is an object to allow implementations to choose their own way of characterizing and searching the session contexts.

Raised Exceptions:

- Exception

```
listRegisteredSessionIds(sessionTrack : SessionTrack) : List<String>
```

Arguments:

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

Raised Exceptions:

- Exception

```
setSessionAttributeValue(sessionTrack : SessionTrack, sessionId : Object,  
attributeName : String, attributeValue : Object) : void
```

Set a specific attribute for an identified [session](#).

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- sessionId - The identifier of the [session](#) whose [Session Context](#) is going to be modified.
- attributeName - The name of the [session](#) attribute to be set.
- attributeValue - The new value for the specified [session](#) attribute.

Raised Exceptions:

- Exception

Typical Use

The Session Management Service provides a [session](#) management capability similar to those provided by a web server.

Associated Diagrams

Use Cases

- [UC 0000 Phoenix IM Capabilities](#)
- [UC 0011 Session Management](#)

Activity Diagrams

- [AD 0016 Session Management \(Administrator\)](#)
- [AD 0017 Session Management \(User\)](#)

Class Diagrams

- [CD 0000 Phoenix IM Services](#)
- [CD 0014 Session Management](#)

Sequence Diagrams

- [SQD 0011 Session Management \(Creation & Destruction\)](#)

[SessionManagementServiceConnector](#)

This interface extends the Session Management Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Session Management Service)

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.

Public Operations

`getActiveSessions() : List<SessionContext>`

Retrieve the list of currently active session contexts.

`getDefaultSessionTimeToLive() : long`

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

`getMaxSupportedSessions() : long`

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

`getNumberOfSessions() : long`

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

SessionManagementServiceStub

This interface extends the Session Management Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Session Management Service)

Subscription

The subscription group contains the interfaces that define the constructs used to support edge actor subscription operations:

- [BrokeringResultType](#)
- [ConsumerReport](#)
- [SubscriptionContext](#)
- [SubscriptionService](#)
- [SubscriptionServiceConnector](#)
- [SubscriptionServiceContext](#)
- [SubscriptionServiceStub](#)

BrokeringResultType

This enumeration describes the possible results of an information brokering operation.

Public Fields

CONSUMER_LIST

This flag represents the expression registrant's desire to receive consumer hit lists for their submitted information via an [Event Channel](#).

INFORMATION

This flag represents the expression registrant's desire to receive information matching the submitted expression via a [Dissemination Service](#).

ConsumerReport

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

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 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 channel 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.

SubscriptionContext

This class is used specifically for registering expressions with the [Information Brokering Service](#) as subscriptions. Information brokering is the concurrent 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.

Public Operations

`addConsumerChannel(consumer : ChannelContext) : void`

Add a consumer channel to receive the results of this subscription.

Arguments:

- consumer - The context describing the consumer channel to receive the results of this subscription.

`addExpression(expression : ExpressionContext) : void`

Add an expression to this subscription.

Arguments:

- expression - The expression to add to this subscription.

`addInformationTypeName(typeName : String) : void`

Add an information type name for this subscription to be applied to.

Arguments:

- infoTypeName - The information type name that this subscription will be applied to.

`getBrokeringResultType() : BrokeringResultType`

Retrieve the brokering result type flag.

`getConsumerReportType() : ConsumerReport`

Retrieve the type of ConsumerReport, if any, to be returned as a result of a brokering operation upon the associated expression.

`getFirstMatchFoundTime() : long`

Retrieve the amount of time the consumer is willing to listen for the first result that matches the associated subscription.

`listConsumerChannels() : List<ChannelContext>`

Retrieve the entire list of consumer channels bound to this subscription.

`listExpressions() : List<ExpressionContext>`

Retrieve the entire list of expressions for this subscription.

`listInformationTypeNames() : List<String>`

Retrieve the information type names that this subscription applies to.

`setBrokeringResultType(resultType : BrokeringResultType) : void`

Set the brokering result type flag.

Arguments:

- `resultType` - The brokering result type flag.

`setConsumerReportType(reportType : ConsumerReport) : void`

Set the type of `ConsumerReport`, if any, to be returned as a result of a brokering operation upon the associated expression.

Arguments:

- `reportType` - The type of `ConsumerReport`, if any, to be returned as a result of a brokering operation upon the associated expression.

`setFirstMatchFoundTime(fmft : long) : void`

Set the amount of time the consumer is willing to listen for the first result that matches the associated subscription.

Arguments:

- `fmft` - The amount of time the consumer is willing to listen for the first result that matches the associated subscription.

SubscriptionService

This service extends the Base Channel Service and manages subscriptions for information. This edge-facing service coordinates subscription registration across a set of Information Brokering Services.

Public Operations

`dropSubscriptions(sessionTrack : SessionTrack, subscriptionIds : List<String>) : void`

This method is used to drop subscriptions.

Arguments:

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

Raised Exceptions:

- `Exception`

`registerSubscriptions(sessionTrack : SessionTrack, subscriptionContexts : List<SubscriptionContextInterface>) : List<String>`

This method is used to register subscriptions for evaluation. This method returns the registered subscription identifiers.

Arguments:

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

Raised Exceptions:

- Exception

```
resumeSubscriptions(sessionTrack : SessionTrack, subscriptionIds : List<String>)
: void
```

Resume evaluation of the identified subscriptions.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- subscriptionIds - The identifiers of the subscriptions to resume evaluation over.

Raised Exceptions:

- Exception

```
suspendSubscriptions(sessionTrack : SessionTrack, subscriptionIds :
List<String>) : void
```

Temporarily suspend evaluation of the identified subscriptions.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- subscriptionIds - The identifiers of the subscriptions to suspend evaluation over.

Raised Exceptions:

- Exception

```
updateSubscription(sessionTrack : SessionTrack, subscriptionId : String,
subscriptionContext : SubscriptionContext) : void
```

Update the identified subscription.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- subscriptionId - The identifier of the subscription to be updated.
- subscriptionContext - The context containing the new settings for the subscription.

Raised Exceptions:

- Exception

Typical Use

This service is used in conjunction with a set of Information Brokering Services to provide load balancing for subscription expression evaluation operations.

Associated Diagrams

Use Cases

- [UC 0000 Phoenix IM Capabilities](#)

Activity Diagrams

-

Class Diagrams

- [CD 0000 Phoenix IM Services](#)

Sequence Diagrams

-

SubscriptionServiceConnector

This interface extends the Subscription Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Subscription Service)

SubscriptionServiceContext

An interface for a context for the Subscription Service.

Public Operations

None.

SubscriptionServiceStub

This interface extends the Subscription Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Subscription Service)

Streams Service Interfaces

Stream services are those that manage streams and actor's interactions with streams.

- [Connection](#)
- [Stream Brokering](#)
- [Stream Discovery](#)
- [Stream Repository](#)

Connection

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

- [ConnectionService](#)
- [ConnectionServiceConnector](#)
- [ConnectionServiceContext](#)
- [ConnectionServiceStub](#)
- [ConnectionGroup](#)
- [ConnectionGroupContext](#)

ConnectionService

The Connection Service extends the Base Channel Service and is responsible for taking frames, information, or bytes, from registered sources and delivering them to registered [consumers](#). Sources and consumers register with the service, and are multiplexed or demultiplexed through a connection group. A [connection group](#) is the structure which encapsulates a group of inputs and a group of outputs, for data to be forwarded between.

Public Operations

```
addConnectionGroupConsumers(session : SessionTrack, connectionGroupId : String,  
consumerIds : List<String>) : void
```

Adds the specified consumer endpoints (referenced by consumerIds) to the list of outputs for the connection group associated with the connectionGroupId parameter.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The connection group identifier for the connection group to add a consumer to.
- consumerIds - The identifiers of the consumers to be added.

Raised Exceptions:

- [Exception](#)

```
addConnectionGroupSources(session : SessionTrack, connectionGroupId : String,  
sourceIds : List<String>) : void
```

Adds the specified source (referenced by sourceId) to the list of inputs for the connection group associated with the connectionGroupId parameter.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The connection group identifier for the connection group to add a source to.
- sourceIds - The identifiers of the sources to be added.

Raised Exceptions:

- [Exception](#)

```
deregisterConnectionGroups(session : SessionTrack, connectionGroupIds :  
List<String>) : void
```

Removes the specified connection groups from the list of registered connection groups.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupIds - The connection group identifiers to be deregistered.

Raised Exceptions:

- [Exception](#)

```
deregisterConsumers(session : SessionTrack, consumerIds : List<String>) : void
```

Removes the specified consumers from the existing registered consumers.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- consumerIds - The consumer identifiers for the consumers to be deregistered.

Raised Exceptions:

- [Exception](#)

```
deregisterSources(session : SessionTrack, sourceIds : List<String>) : void
```

Removes the specified sources from the existing registered sources.

Arguments:

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

- sourceIds - The source identifiers for the sources to be deregistered.

Raised Exceptions:

- [Exception](#)

`getAllConnectionGroups(session : SessionTrack) : List<ConnectionGroupContext>`

Retrieve a listing of all the connection groups currently registered with this service.

Arguments:

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

Raised Exceptions:

- [Exception](#)

`getAllConsumers(session : SessionTrack) : List<ChannelContext>`

Retrieve a listing of all of the consumers currently registered with this service.

Arguments:

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

Raised Exceptions:

- [Exception](#)

`getAllSources(session : SessionTrack) : List<ChannelContext>`

Retrieve a listing of all of the sources currently registered with this service.

Arguments:

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

Raised Exceptions:

- [Exception](#)

`getConnectionGroupConsumers(session : SessionTrack, connectionGroupId : String)
: List<ChannelContext>`

Obtains a channel list of all of the current consumers for a specific connection group.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The identifier of the connection group which the consumers are associated with.

Raised Exceptions:

- [Exception](#)

`getConnectionGroupContext(session : SessionTrack, connectionGroupId : String) : ConnectionGroupContextInterface`

Retrieve a listing of the identifiers for all of the connection groups currently registered with this service. This method returns a context that contains the requisite data.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The identifier of the connection group.

Raised Exceptions:

- [Exception](#)

`getConnectionGroupSources(session : SessionTrack, connectionGroupId : String) : List<ChannelContext>`

Obtains a channel list of all of the current sources for a specific connection group.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The identifier of the connection group which the sources are associated with.

Raised Exceptions:

- [Exception](#)

`isConnectionGroupRegistered(session : SessionTrack, connectionGroupId : String) : boolean`

Check if the identified connection group is registered with this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The identifier of the connection group.

Raised Exceptions:

- [Exception](#)

`isConsumerRegistered(session : SessionTrack, consumerId : String) : boolean`

Check if the identified consumer is registered with this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- consumerGroupId - The identifier of the consumer.

Raised Exceptions:

- [Exception](#)

```
isConsumerRegisteredWithConnectionGroup(session : SessionTrack,
connectionGroupId : String, consumerId : String) : boolean
```

Check if the identified consumer is registered with a specific connection group on this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The identifier of the connection group.
- consumerId - The identifier of the consumer.

Raised Exceptions:

- [Exception](#)

```
isSourceRegistered(session : SessionTrack, sourceId : String) : boolean
```

Check if the identified source is registered with this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- sourceId - The identifier of the source.

Raised Exceptions:

- [Exception](#)

```
isSourceRegisteredWithConnectionGroup(session : SessionTrack, connectionGroupId
: String, sourceId : String) : boolean
```

Check if the identified source is registered with a specific connection group on this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The identifier of the connection group.
- sourceId - The identifier of the source.

Raised Exceptions:

- [Exception](#)

```
registerConnectionGroups(session : SessionTrack, connectionGroupId : String,
groupContexts : List<ConnectionGroupContext>) : List<String>
```

Register the connection group on this service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- groupContexts - The connection group contexts

Raised Exceptions:

- [Exception](#)

```
registerConsumers(session : SessionTrack, connectionGroupId : String,
channelContexts : List<ChannelContext>) : List<String>
```

Registers consumers to receive data. This method returns the identifiers for the registered consumers.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelContexts - The list of ChannelContexts that describe the consumers to be registered.

Raised Exceptions:

- [Exception](#)

```
registerSources(session : SessionTrack, connectionGroupId : String,
channelContexts : List<ChannelContext>) : List<String>
```

Registers sources to stream data. This method returns the identifiers for the newly registered sources.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- channelContexts - The list of ChannelContexts that describe the sources to be registered.

Raised Exceptions:

- [Exception](#)

```
removeConnectionGroupConsumers(session : SessionTrack, connectionGroupId :
String, consumerIds : List<String>) : void
```

Drops the specified consumers referenced by consumerIds from the list of outputs for the connection group associated with the connectionGroupId parameter.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The identifier of the connection group.
- consumerIds - The list of consumers to be removed from membership in the connection group.

Raised Exceptions:

- [Exception](#)

```
removeConnectionGroupSources(session : SessionTrack, connectionGroupId : String,
sourceIds : List<String>) : void
```

Drops the specified sources referenced by sourceIds from the list of inputs for the connection group associated with the connectionGroupId parameter.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupId - The identifier of the connection group.
- sourceIds - The list of sources to be removed from membership in the connection group.

Raised Exceptions:

- [Exception](#)

```
resetConnectionGroups(session : SessionTrack, connectionGroupIds : List<String>)
: void
```

Drops all sources and consumers from the connection group, but keeps the connection group registered with the service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- connectionGroupIds - The identifiers of the connection groups to reset.

Raised Exceptions:

- [Exception](#)

```
resetConsumers(session : SessionTrack, consumerIds : List<String>) : void
```

Drops all connection group connections with the associated consumers, but keeps the consumer endpoints registered with the service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- consumerIds - The identifiers of the consumers to reset.

Raised Exceptions:

- [Exception](#)

```
resetSources(session : SessionTrack, sourceIds : List<String>) : void
```

Drops all connection group connections with the associated sources, but keeps the source endpoints registered with the service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- sourceIds - The identifiers of the sources to reset.

Raised Exceptions:

- [Exception](#)

Typical Use

This service can be used by itself to combine source feeds or to perform scatter/gather signaling to consumers. It can also be used as a disseminator for the StreamBrokering Service, which wraps much of the functionality in order to manage streams. The purpose of the service is to manage source and consumer connections into functional disseminators which are fixed until reorganized, rather than have connections dynamically managed at the time an information object is flowing through the services for processing. The connections are essentially 'pre-brokered', which is more optimal for streams, and removes unnecessary overhead.

Associated Diagrams

Class Diagrams

- [CD 0000 Phoenix IM Services](#)

ConnectionServiceConnector

This interface extends the Connection Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Connection Service)

ConnectionServiceContext

The Context for the Connection Service holds the list of registered consumers, sources (publishers), and connection groups. The list of registeredConsumers contains the Contexts that describe the end points for all consumers who have been registered with the associated Disseminator Service. The maxNumberOfConsumers is the maximum number of supported consumers. The maxNumberOfSources is the maximum number of supported sources. The maxNumberOfConnectionGroups is the maximum number of supported connection groups this service will support. The numberOfRegisteredConsumers is the current number of consumers who have registered with the associated Connection Service. The numberOfRegisteredSources is the current number of sources who have registered with the associated Connection Service. The numberOfRegisteredConnectionGroups is the current number of connection groups who have registered with the associated Connection Service.

Public Operations

`addConnectionGroup(connectionGroup : ConnectionGroup) : void`

Add a registered connection group.

Arguments:

- connectionGroup - The [ConnectionGroup](#) to add as a member of the service.

```
addConnectionGroupConsumer(connectionGroupId : String, consumerId : String) : void
```

Map a registered consumer connection to a registered connection group.

Arguments:

- connectionGroupId - The [ConnectionGroup](#) identifier to link the consumer to.
- consumerId - The consumer identifier for the consumer to route data from the connection group to.

```
addConnectionGroupSource(connectionGroupId : String, sourceId : String) : long
```

Map a registered source connection to a registered connection group.

Arguments:

- connectionGroupId - The [ConnectionGroup](#) identifier to link the source to.
- sourceId - The source identifier for the source from which to route data.

```
addRegisteredConsumer(consumerIdentifier : String, consumerChannelContext : ChannelContext) : void
```

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

Arguments:

- consumerIdentifier - The identifier for the consumer to be added.
- consumerChannelContext - The output channel context for the consumer.

Raised Exceptions:

- [Exception](#)

```
addRegisteredSource(sourceIdentifier : String, sourceChannelContext : ChannelContext) : void
```

Add a registered source's information to the service context.

Arguments:

- sourceIdentifier - The identifier for the source to be added.
- sourceChannelContext - The input channel context for the source.

Raised Exceptions:

- [Exception](#)

```
getConnectionGroup(connectionGroupId : String) : ConnectionGroup
```

Retrieve the connection group by id.

Arguments:

- connectionGroupId - The identifier for the connection group to be retrieved.

`getConnectionGroups() : Map <String, ConnectionGroup>`

Retrieve the set of currently registered connection groups.

`getConsumersConnectionGroups() : Map <String, List<ConnectionGroup>>`

Retrieve the set of currently registered connection groups according to their consumer.

`getMaxSupportedConnectionGroups() : long`

Retrieve the theoretical maximum number of concurrent connection groups that the associated parent service can support.

`getMaxSupportedConsumers() : long`

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

`getMaxSupportedSources() : long`

Retrieve the theoretical maximum number of concurrent sources that the associated parent service can support.

`getNumberOfRegisteredConnectionGroups() : long`

Retrieve the current number of registered connection groups.

`getNumberOfRegisteredConsumers() : long`

Retrieve the current number of registered consumers.

`getNumberOfRegisteredSources() : long`

Retrieve the current number of registered sources.

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

Retrieve the set of currently registered consumers.

`getRegisteredSources()() : Map <String, ChannelContext>`

Retrieve the set of currently registered sources.

`getSourcesConnectionGroups() : Map <String, List<ConnectionGroup>>`

Retrieve the set of currently registered connection groups according to their source.

`removeConnectionGroup(connectionGroupId : String) : ConnectionGroup`

Remove the identified connection group.

Arguments:

- connectionGroupId - The identifier for the connection group to be removed.

`removeConnectionGroupConsumer(connectionGroupId : String, consumerId : String) : boolean`

Remove the registered consumer from the connection group.

Arguments:

- connectionGroupId - The identifier for the connection group to remove a consumer from.

- consumerId - The identifier for the consumer to remove.

`removeConnectionGroupSource(connectionGroupId : String, sourceId : String) : boolean`

Remove the registered source from the connection group.

Arguments:

- connectionGroupId - The identifier for the connection group to remove a consumer from.
- sourceId - The identifier for the source to remove.

`removeRegisteredConsumer(consumerId : String) : ChannelContext`

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

Arguments:

- consumerId - The identifier for the consumer to remove.

`removeRegisteredSource(sourceId : String) : ChannelContext`

Remove the identified source's information from the list of registered sources.

Arguments:

- sourceId - The identifier for the source to remove.

`setMaxSupportedConnectionGroups(max : long) : void`

Set the maximum number of concurrently supported connection groups for this service.

Arguments:

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

`setMaxSupportedConsumers(max : long) : void`

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

Arguments:

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

`setMaxSupportedSources(max : long) : void`

Set the maximum number of concurrently supported sources for this service.

Arguments:

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

ConnectionServiceStub

This interface extends the Connection Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Connection Service)

ConnectionGroup

A Connection Group. A Connection Group contains a context and some type of structure for managing source and consumer memberships. A Connection Group, simplistically, is a set of source channels and a set of consumer channels, all tied together in membership for the purpose of multiplexing or de-multiplexing.

Connection Group allows for subclasses to type the input channels and output channels. So some extensions could define the channels as a channel-wrapped buffer, and others could use the transport itself. These are typed as 'InputChannel' and 'OutputChannel' 'InChannel' - The input channel format for the application level source members. 'OutChannel' - The output channel format for the application level consumer members.

Public Operations

`getConnectionGroupContext() : ConnectionGroupContext`

Retrieve the connection group context, which contains the id and attributes of the connection group.

`getConnectionGroupId() : String`

Retrieve the connection group unique identifier.

`reset() : void`

Reset the connection group. This will remove all of the sources and consumers for this connection group.

Raised Exceptions:

- ChannelException

`setConnectionGroupContext(connectionGroupCxt : ConnectionGroupContext) : void`

Set the ConnectionGroupContext.

Arguments:

- connectionGroupCxt - The connection group context, which contains metadata regarding the setup of the multiplexing/de-multiplexing

`setConnectionGroupId(connectionGroupId : String) : void`

Set the connection group unique identifier.

Arguments:

- connectionGroupId - The identifier for the connection group.

ConnectionGroupContext

A Connection Group Context. Is the container for configuration of a connection group.

Public Operations

`getConnectionGroupChannelType() : ChannelType`

Get the connection group channel type, which all of the sources and sinks for this connection group must comply with.

`getConnectionGroupId() : String`

Retrieve the connection group unique identifier.

`isManyConsumersAllowed() : String`

Checks whether the consumers are forced to be One or Many.

`isManySourcesAllowed() : String`

Checks whether the sources are forced to be one or many.

`setConnectionGroupChannelType(channelType : ChannelType) : void`

Set the connection group channel type.

Arguments:

- `channelType` - The channel type for all associated sources and consumers.

`setConnectionGroupId(id : String) : void`

Set the connection group unique identifier.

Arguments:

- `id` - The identifier for the connection group.

`setManyConsumersAllowed(flag : boolean) : void`

Determines whether the consumers are forced to be one or many.

Arguments:

- `id` - Whether or not more than one consumer is allowed for this connection group

`setManySourcesAllowed(flag : boolean) : void`

Determines whether the sources are forced to be one or many.

Arguments:

- `id` - Whether or not more than one source is allowed for this connection group

Stream Brokering

The streambrokering group contains the interfaces that define and support the stream brokering and registration/management capability of the Phoenix architecture. The streambrokering interfaces are:

- `AttributeType`
- `StreamBrokeringServiceConnector`

- [StreamBrokeringServiceContext](#)
- [StreamBrokeringService](#)
- [StreamBrokeringServiceStub](#)
- [StreamContext](#)
- [StreamHeaderAttribute](#)
- [StreamHeader](#)
- [StreamSubscriptionContext](#)

AttributeType

This enumeration contains the possible types of attributes for stream header attributes defined by the Phoenix architecture.

Public Fields

[INT](#)

Base Primitive, integer value type.

[SHORT](#)

Base Primitive, short value type.

[DOUBLE](#)

Base Primitive, double value type.

[FLOAT](#)

Base Primitive, float value type.

[LONG](#)

Base Primitive, long value type.

[STRING](#)

Base Primitive, string value type.

[BYTE](#)

Base Primitive, byte value type.

[BOOLEAN](#)

Base Primitive, boolean value type.

StreamBrokeringServiceConnector

This interface extends the Stream Brokering Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Stream Brokering Service)

StreamBrokeringServiceContext

The service context specific to the Stream Brokering Service. It is the manager of the stream, stream expression, and contributor membership.

Public Operations

```
addConnectionServiceAssociation(connectionService : ConnectionServiceStub) : void
```

Add a connection service association for the use of the stream brokering service.

Arguments:

- connectionService - The [ConnectionServiceStub](#) The connection service stub to add to the list of available connection service associations.

Raised Exceptions:

- [Exception](#)

```
addContributorContext(contributorId : String, channelContext : ChannelContext) : void
```

Add a contributor channel context to those held by this service.

Arguments:

- contributorId - The id of the contributor to which the context belongs.
- channelContext - The context to add to our listings.

```
addStreamContext(streamId : String, streamContext : StreamContext) : void
```

Add a stream context to those held by this service.

Arguments:

- streamId - The stream context to add to our listings.
- streamContext - The stream context to add to our listings.

```
addStreamSubscriptionContext(subContextId : String, subContext : StreamSubscriptionContext) : void
```

Add a stream subscription context to those held by this service.

Arguments:

- subContextId - The id of the stream subscription to which the context belongs.
- subContext - The context to add to our listings.

```
getAvailableConnectionServices() : List<ConnectionServiceStub>
```

Get the list of all available connection services which we can route streams through.

Raised Exceptions:

- `Exception`

`getContributorContexts() : List<ChannelContext>`

Get all of the contributor channel contexts held by this service.

`getStreamContext(streamContextId : String) : StreamContext`

Get the stream context referenced by the passed identifier.

Arguments:

- `streamContextId` - The stream context identifier..

`getStreamContexts() : List<StreamContext>`

Get the stream subscription context referenced by the passed identifier.

`getStreamSubscriptionContext(streamSubscriptionContextId : String) : StreamSubscriptionContext`

Get the stream subscription context referenced by the passed identifier.

Arguments:

- `streamSubscriptionContextId` - The stream subscription context identifier.

`getStreamSubscriptionContexts() : List<StreamSubscriptionContext>`

Get the stream subscription context referenced by the passed identifier.

`removeConnectionServiceAssociation(csStub : ConnectionServiceStub) : boolean`

Remove a connection service association from the list of those available for use by the stream brokering service.

Arguments:

- `csStub` - The Connection Service Stub to remove.

Raised Exceptions:

- `Exception`

`removeContributorChannelContext(contributorId : String) : ChannelContext`

Remove the channel context referenced by the passed contributor identifier.

Arguments:

- `contributorId` - The Channel Context to remove.

`removeStreamContext(streamId : String) : StreamContext`

Remove the stream context referenced by the passed parameter.

Arguments:

- `streamId` - The stream id referencing the stream context to be removed.

```
removeStreamSubscriptionContext(streamSubscriptionId : String) :  
StreamSubscriptionContext
```

Remove the stream subscription context referenced by the passed parameter.

Arguments:

- streamId - The stream subscription id referencing the stream subscription to be removed.

```
setAvailableConnectionServices(serviceList : List<ConnectionServiceStub>) : void
```

Set the list of all available connection services which we can route streams through.

Arguments:

- serviceList - The connection services available for use by this service.

Raised Exceptions:

- [Exception](#)

StreamBrokeringService

The Stream Brokering Service interface extends the Base Channel Service interface and offers the capability of registering a stream to stream producers, and subscribing to streams to consumers. The Stream Brokering Service wraps the functionality of Connection Service with Stream administration. Membership of streams, including their publication source's metadata and identity, consumers using stream-based expressions, and the registration of the stream definition itself, are all part of the Stream paradigm for this service. Data does not flow through this service (that is routed through a connection service) but the control and management operations for a stream are administered through this service.

Public Operations

```
deregisterStreamContributor(sessionTrack : SessionTrack, streamId : String,  
contributorId : String) : void
```

Deregister a stream contributor from a stream.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamId - The contributorId to deregister from a stream.
- contributorId - The stream which the contributor is deregistering from.

Raised Exceptions:

- [Exception](#)

```
deregisterStreams(sessionTrack : SessionTrack, streamIds : List<String>) : void
```

Deregister a list of specified streams from the service.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamIds - The stream ids of the Streams to deregister from this service.

Raised Exceptions:

- Exception

```
dropStreamSubscriptions(sessionTrack : SessionTrack, streamSubscriptionIds : List<String>) : void
```

Drop the specified list of stream expressions as specified in the list by their stream expression ids. Drop means to completely remove. Suspend is temporary, but drop is permanent, unless the expression is added again.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamSubscriptionIds - The Stream expression context identifiers for the expressions to drop.

Raised Exceptions:

- Exception

```
getAllStreamContexts(sessionTrack : SessionTrack) : List<StreamContext>
```

Get the list of streams registered with this service.

Arguments:

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

Raised Exceptions:

- Exception

```
getAllStreamSubscriptionContexts(sessionTrack : SessionTrack) : List<ExpressionContext>
```

Get the list of stream expressions registered with this service.

Arguments:

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

Raised Exceptions:

- Exception

```
getAllRegisteredStreamContributors(sessionTrack : SessionTrack) : List<ChannelContext>
```

Get the list of stream contributor channel contexts registered with this service.

Arguments:

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

Raised Exceptions:

- Exception

```
getStreamContext(sessionTrack : SessionTrack, streamId : String) : StreamContext
```

Get the stream context for a specific stream. This allows someone to check the attributes of a stream before deciding to subscribe to it.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamId - The stream id for the StreamContext to return

Raised Exceptions:

- Exception

```
getStreamSubscriptionContext(sessionTrack : SessionTrack, streamSubscriptionId : String) : ExpressionContext
```

Get the stream expression context associated with a subscription which was submitted previously.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamSubscriptionContextId - The stream expression context identifier to retrieve.

Raised Exceptions:

- Exception

```
registerStreams(sessionTrack : SessionTrack, streamContexts : List<StreamContext>) : List<String>
```

Register the stream with this service. The stream will be setup for stream expression matching from future consumer's subscriptions.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamContexts - The stream contexts which contains all necessary fields and attributes for defining the new streams.

Raised Exceptions:

- Exception

```
registerStreamContributor(sessionTrack : SessionTrack, streamId : String, streamContexts : ChannelContext) : ChannelContext
```

Register a stream producer with this service. The stream will be setup for stream expression matching from future consumer's subscriptions.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamId - The stream which the contributor is registering with.
- channelContext - The channel context which contains all necessary fields and attributes for defining a channel to publish to a stream.

Raised Exceptions:

- Exception

```
registerStreamSubscriptions(sessionTrack : SessionTrack, streamSubscriptions :  
List<ExpressionContext>) : List<String>
```

Register one or more stream expressions with the service, which will allow for the associated endpoints to be pushed the data at the time of publication.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamSubscriptions - The subscriptions to perform stream matching to.

Raised Exceptions:

- Exception

```
resumeStreamSubscriptions(sessionTrack : SessionTrack, streamSubscriptionIds :  
List<String>) : void
```

Resume certain stream subscriptions. Subscriptions must be in suspended state first, otherwise resuming has no effect. Expressions will have to be re-brokered at the time of being resumed.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamSubscriptionIds - The stream subscription identifiers for the stream subscriptions to resume distribution of their matching streams to.

Raised Exceptions:

- Exception

```
suspendStreamSubscriptions(sessionTrack : SessionTrack, streamSubscriptionIds :  
List<String>) : void
```

Suspend certain stream subscriptions, removing them from those being distributed the streaming data until they are resumed.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamSubscriptionIds - The stream subscription identifiers for the stream subscriptions to suspend.

Raised Exceptions:

- Exception

```
updateRegisteredStreams(sessionTrack : SessionTrack, streamContexts :
List<StreamContext>) : void
```

Update the registered stream with new context attributes and settings. Overrides any existing stream definition, and causes a re-mapping of the brokered consumer list for stream data dissemination. Stream's channel type and id must be the same, but all other criteria are changeable.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- expressionIds - The stream contexts to update to these new definitions.

Raised Exceptions:

- Exception

Typical Use

This service is used to simplify the operations of stream management, as well as the intricacies of the connection service, by providing a wrapper which does much of the work for the producer and consumer in terms of stream registration being converted to a connection group definition, a consumer and source being registered and added as members of that connection group based on the stream expression or their stream Id. The main purpose of this service however, and why it is typically used, is because it uses the ontological terms and language of streaming so that the operations of the connection service can be much easier related to, and also specialized for streaming functionality. The stream channel types allowed are information, frame, and byte.

As such, the Context containing this service's state may contain the Connection Service Bindings and the mappings of the stream contributors and expressions to their matching streams.

Associated Diagrams

Class Diagrams

- [CD 0000 Phoenix IM Services](#)

StreamBrokeringServiceStub

This interface extends the Stream Brokering Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Stream Brokering Service)

StreamContext

The context of a stream contains the attributes and stream associated data, so it can be registered, subscribed to, and matches for consumer membership determined based on its contents. A Stream has a set of fixed metadata which are not to change throughout the life of the stream. It has other attributes which are variable and allowed to change over its lifecycle; these are the individual payloads of the stream.

Public Operations

`getChannelType() : ChannelType`

Get the channel type associated with the stream.

`setChannelType(channelType : ChannelType) : void`

Set the channel type associated with the stream.

Arguments:

- `channelType` - The channel type which all data in the stream must be formatted into.

`getPublisherMetadata() : Map<String, String>`

Retrieve the metadata associated with all publishers tied to this Stream.

`getPublisherContexts() : Map<String, BaseContext>`

Retrieve the context associated with all publishers tied to this Stream.

`getStreamId() : String`

Get the stream id for the stream which is described by this context.

`getStreamName() : String`

Get the name of the stream for this stream context.

`getStreamMetadata() : String`

Get the stream metadata. This is what defines the stream, so that stream subscriptions can be matched against it to determine the feed consumers for the stream.

`getStreamHeaderDefinition() : String`

Get the stream header definition. This is the outline of the stream header, and can be used to define the bytes which are sent across as the header.

`addPublisherMetadata(publisherId : String, publisherMetadata : String) : void`

Add the publisher metadata to those associated with the stream.

Arguments:

- `publisherId` - The identifier for the publisher being added as a contributor to the stream.
- `publisherMetadata` - The metadata associated with the publisher being added.

`addPublisherContext(publisherId : String, publisherContext : BaseContext) : void`

Add the publisher context to those associated with the stream.

Arguments:

- publisherId - The identifier for the publisher.
- publisherContext - The context associated with the publisher being added.

`setStreamId(streamId : String,) : void`

Set the stream id for the stream associated with this context.

Arguments:

- streamId - The stream identifier.

`setStreamName(streamName : String,) : void`

Set the stream name for the stream associated with this context.

Arguments:

- streamName - The name of the stream associated with this context.

`setStreamMetadata(streamMetadata : String,) : void`

Set the stream metadata, which will be used to match stream subscriptions with stream contexts, administrating stream membership.

Arguments:

- streamMetadata - The stream metadata.

`setStreamHeaderDefinition(streamHeader : StreamHeader) : void`

Set the stream header definition for the stream associated with this context.

Arguments:

- streamHeader - The stream header definition.

`removePublisherMetadata(publisherId : String) : String`

Remove the publisher metadata from the list of publisher metadata for the stream.

Arguments:

- publisherId - The publisher id whose metadata will be removed from the stream.

`removePublisherContext(publisherId : String) : BaseContext`

Remove the publisher context from the list of publisher contexts for the stream.

Arguments:

- publisherId - The publisher id whose context will be removed from the stream.

StreamHeaderAttribute<T>

An individual attribute of a stream header, this interface defines what an attribute is, and outlines what it can consist of.

Public Operations

`getAttributeType() : <AttributeType>`

Get the attribute type for this attribute.

`setAttributeName(attributeValue : <T>) : void`

Set the attribute name itself.

Arguments:

- attributeValue - The attribute to set as the new value.

`getAttributeName() : String`

Get the attribute name.

`getAttributeDescription() : String`

Get the description which explains the relevance, purpose, and other details about the attribute.

`setAttributeDescription(description : String) : void`

Set the attribute description. For ease of use.

Arguments:

- description - The description of this attribute. Will replace any existing value.

`getAttributeLength() : String`

Get the length of this attribute. Assists in the definition of any protocol or header definition which relies on this attribute.

`setAttributeLength(length : int) : void`

Set the length of this attribute. Assists in the definition of any protocol or header definition which relies on this attribute.

Arguments:

- length - The length of the attribute. This is the length in regards to the number of bytes of the attribute value.

`setFixedLength(flag : boolean) : void`

Set the flag which describes whether the attribute is fixed length or variable. If variable, likely KLV, torrent, or other format should be used.

Arguments:

- flag - Whether the attribute is fixed length or not.

`isFixedLength() : boolean`

Get the flag which describes whether the attribute is fixed length or variable. If variable, likely KLV, torrent, or other format should be used.

StreamHeader

The header for a stream. This is custom-implemented to the stream. The stream may have just a base header, or may have a more extended, additional header for specific tags to be included in the header which will be informative to the end-consumer or to the broker.

Public Operations

`addHeaderAttribute(streamHeaderAttribute : StreamHeaderAttribute) : void`

Add a header attribute for the stream header. The header attributes are in order of their addition to the header, first to last.

Arguments:

- streamHeaderAttribute - The attribute to append to the header.

`removeHeaderAttribute(streamHeaderAttribute : StreamHeaderAttribute) : boolean`

Remove a specific header attribute from the header.

Arguments:

- streamHeaderAttribute - The header attribute to remove.

`getHeaderAttributes() : List<StreamHeaderAttribute>`

Get the complete list of attached stream header attributes. The order they are in the list is the order they will be encoded into the header.

`getHeaderFormatType() : HeaderFormatType>`

The header format type. This describes the encoding for how to interpret the bytes of the stream header.

StreamSubscriptionContext

The context for a stream subscription. Contains the expression associated with the subscription, a subscription identifier, and a list of the consumers currently registered as data receivers for this stream subscription. This is a server side structure meant to support easy cataloging of stream-based expressions and their matching consumers, or, at least all of the consumers which the Stream Brokering Service has ownership of. Stream expression matching is a little of a different beast than information brokering. Information is brokered on an object by object basis, however, a stream is brokered just once, and then all segments of the stream are assumed to match the consumers which are already members of the stream.

Public Operations

`addConsumerChannel(consumer : ChannelContext) : void`

Add a consumer channel to receive the results of this stream subscription.

Arguments:

- consumer - The context describing the consumer channel to receive the results of this stream subscription.

`getExpression() : ExpressionContext`

Get the context for the stream expression associated with this subscription.

`getSubscriptionId(streamHeaderAttribute : StreamHeaderAttribute) : String`

Get the subscription id for this stream subscription context.

`listConsumerChannels() : List<ChannelContext>`

Retrieve the entire list of consumer channels bound to this stream subscription.

`listRegisteredConsumers() : List<String>`

List the registered consumers who are applicable to this stream subscription in a list of strings.

`setExpression(expressionContext : ExpressionContext) : void`

Set the context for the stream expression associated with this stream subscription.

Arguments:

- `expressionContext` - The expression context.

`setRegisteredConsumersList(consumerList : List<String>) : void`

Set the list of consumers who are associated with this stream subscription.

Arguments:

- `consumerList` - The list of stream consumers.

`setSubscriptionId(subscriptionId : String) : void`

Set the subscription id for this context to reference.

Arguments:

- `subscriptionId` - The subscription id.

Stream Discovery

The streamdiscovery group contains the interfaces that define and support the stream discovery and registration/query capability of the Phoenix architecture. The streamdiscovery interfaces are:

- [StreamDiscoveryServiceConnector](#)
- [StreamDiscoveryServiceContext](#)
- [StreamDiscoveryService](#)
- [StreamDiscoveryServiceStub](#)
- [StreamDiscoveryContext](#)
- [StreamDiscoveryQueryContext](#)

StreamDiscoveryServiceConnector

This interface extends the Stream Discovery Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Stream Discovery Service)

StreamDiscoveryServiceContext

Context for the Stream Discovery Service. This has all necessary methods for the serialization, transport, and standup of the exact same Stream Discovery Service. It keeps a registry of registered streams, as well as a set of expression processors for brokering stream queries. Multiple expression processors are allowed and are chosen based on the query expression type.

Public Operations

```
addStreamDiscoveryContext(streamId : String, streamContext : StreamDiscoveryContext) : void
```

Add a stream context to those held by this service.

Arguments:

- streamId - The identifier of the stream discovery context to add to our listings.
- streamContext - The context to add to our listings.

```
getStreamDiscoveryContexts() : List<StreamDiscoveryContext>
```

Get all of the stream discovery contexts held by this service.

```
getStreamDiscoveryContext(streamId : String) : StreamDiscoveryContext
```

Get a single stream discovery context held by this service.

Arguments:

- streamId - The stream id referencing the context to be retrieved.

```
removeStreamDiscoveryContext(streamId : String) : StreamDiscoveryContext
```

Remove the stream discovery context referenced by the passed parameter.

Arguments:

- streamId - The stream id referencing the context to be removed.

StreamDiscoveryService

The Stream Discovery Service interface extends the Base Channel Service interface and is responsible for maintaining a registry of stream descriptions, along with relevant connection meta-data so someone can find out how and where to connect to them. Stream Contexts may include fields such as what specific data types a particular stream supports or where it is physically located, as well as a name, description, and header data for a defined stream.

Public Operations

```
brokerForStreams(sessionTrack : SessionTrack, query : StreamDiscoveryQueryContext) : List<StreamDiscoveryContext>
```

Broker for a set of streams that satisfy the specified constraints. This method returns a list of matching stream contexts which may be used to interact with their associated streams through references internal to the context.

Arguments:

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

Raised Exceptions:

- Exception - If the given query causes an error to occur.

```
getStreamDiscoveryContext(sessionTrack : SessionTrack, streamId : String) :  
StreamDiscoveryContext
```

Retrieve the context being used by the Stream to describe itself by requesting it using its associated stream identifier.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamId - The identifier of the stream context to retrieve.

Raised Exceptions:

- Exception - If the values passed or current service state causes an error to occur.

```
deregisterStream(sessionTrack : SessionTrack, streamId : String) :  
StreamDiscoveryContext
```

Deregister the stream from discovery operations.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamId - The identifier of the stream context to deregister.

Raised Exceptions:

- Exception - If the values passed or current service state causes an error to occur.

```
registerStream(sessionTrack : SessionTrack, streamContext :  
StreamDiscoveryContext) : void
```

Register the stream from discovery operations.

Arguments:

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

Raised Exceptions:

- Exception - If the values passed or current service state causes an error to occur.

```
updateStream(sessionTrack : SessionTrack, streamContext : StreamDiscoveryContext) : void
```

Update the given stream discovery context by cross-referencing its Id with the existing registry.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamContext - The Context object containing the description of the stream to be updated.

Raised Exceptions:

- Exception - If the values passed or current service state causes an error to occur.

StreamDiscoveryServiceStub

This interface extends the Stream Discovery Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Stream Discovery Service)

StreamDiscoveryContext

Stream Discovery Context represents a Stream Definition (As a Stream Context), but is extended to include service stubs related to the stream so that the stream can be joined at the point of its entry into the SOA by any discovering it.

Public Operations

```
getStreamBrokeringServiceStub() : StreamBrokeringServiceStub
```

Get the stream brokering service stub associated with this stream.

```
setStreamBrokeringServiceStub(serviceStub : StreamBrokeringServiceStub) : void
```

Set the stream brokering service stub associated with this stream.

Arguments:

- serviceStub - The stream brokering service to whom is delegated the stream for registration.

StreamDiscoveryQueryContext

A QueryContext used specifically for Stream Discovery operations. An extension of QueryContext, and so is interchangeable in query methods, but is created here for extensions later on.

Public Operations

Stream Repository

The stream repository group contains the interfaces that provide the stream repository capability for the Phoenix IM Services. The stream repository interfaces are:

- [StreamQueryContext](#)
- [StreamQueryService](#)
- [StreamRepository](#)
- [StreamRepositoryService](#)
- [StreamRepositoryServiceConnector](#)
- [StreamRepositoryServiceContext](#)
- [StreamRepositoryServiceStub](#)
- [StreamSequenceContext](#)
- [StreamSequenceRange](#)

StreamQueryContext

A context defining a stream query.

Public Operations

`addStreamForQuery(streamId : String) : void`

Add a stream for querying over to the list of those this query applies to.

Arguments:

- `streamId` - The stream identifier.

`getStreamSequenceContext() : StreamSequenceContext`

Get the stream sequence context of this stream query.

`getStreamsForQuery() : List<String>`

Get the streams for querying over. Returns the list of streams that this query applies to.

`removeStreamForQuery(streamId : String) : void`

Remove a stream for querying over from the list of those this query applies to.

Arguments:

- `streamId` - The stream identifier.

`setStreamSequenceContext(context : StreamSequenceContext) : void`

Set the stream sequence context of the stream query.

Arguments:

- context - The associated stream sequence context for this stream.

```
setStreamsForQuery(streamIds : List<String>) : void
```

Set the streams for querying over.

Arguments:

- streamIds - The stream identifiers.

StreamQueryService

This service extends the Base Channel Service interface and provides a retrieval capability for persisted streams. This service permits actors to retrieve records from the underlying data store. Using a Stream Query Context construct to describe the actual query to be executed allows the architecture to mandate a small set of required query attributes while leaving the door wide open for individual implementations of the IM Services to include additional attributes to tune the query processing of each stream 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 channels.

Typical Use:

This service serves as a query interface for the Stream Repository Service.

Public Operations

```
cancelQuery(sessionTrack : SessionTrack, queryId : String) : boolean
```

Cancel a currently executing query. Executing queries are defined as queries that have any processors cycles associated with them, i.e. a query is not done executing until all results (if any) are delivered to the Dissemination Service for delivery. Returns True if the query was canceled, False otherwise.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- queryId - The unique identifier for the query to be canceled.

Raised Exceptions:

- [Exception](#)

```
executeQuery(sessionTrack : SessionTrack, queryCtx : StreamQueryContext) : int
```

This method processes the specified query to satisfy some inquirers request for data. Actual result sets are delivered via a channel set up between the consumer and the query service. When this method is invoked synchronously, the return value signals 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. Returns a flag that signals the inquisitor the number of matching records that were found or that an error occurred while processing their query.

Arguments:

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

Raised Exceptions:

- [Exception](#)

```
getCounts(sessionTrack : SessionTrack, streamIds : List<String>) : Map<String, Integer>
```

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

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamIds - The listing of stream identifiers to retrieve the count(s) for.

Raised Exceptions:

- [Exception](#)

```
getMatchingSequences(sessionTrack : SessionTrack, streamId : String, sequenceContext : StreamSequenceContext) : List<Long>
```

Retrieves the sequence ids of the matching items in the repository for the specified streams. This method returns a list of longs which are a chronologically ordered list of the sequence identifiers matching the passed query. Returns the sequence identifiers of the records in the repository matching the passed query.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamId - The stream identifier to narrow the focus of the query.
- sequenceContext - The query to perform over the requested stream.

Raised Exceptions:

- [Exception](#)

```
listActiveQueryIds(sessionTrack : SessionTrack) : List<String>
```

List the unique identifiers for the currently executing queries. Executing queries are defined as queries that have any processors cycles associated with them, i.e. a query is not done executing until all results (if any) are delivered to the consumer or a proxy service for delivery. Returns the list of unique identifiers for the currently executing queries.

Arguments:

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

Raised Exceptions:

- [Exception](#)

StreamRepository

An interface for stream repositories that will allow a single RepositoryService to support multiple disparate stream repository implementation technologies.

Public Operations

`addStreamSupport(streamContext : StreamContext) : void`

Add the given stream to the list of supported streams for this repository.

Arguments:

- streamContext - The context describing the stream to be supported.

Raised Exceptions:

- [Exception](#)

`cancelQuery(queryId : String) : boolean`

Cancel the identified query. Returns True if canceled, False if not.

Arguments:

- queryId - The identifier for the query to be canceled.

Raised Exceptions:

- [Exception](#)

`closeRepository() : void`

Close the repository or its connection.

Raised Exceptions:

- [Exception](#)

`delete(query : StreamQueryContext) : int`

Delete all records that match the predicate provided in the given query context object. Returns the number of information instances deleted from the repository.

Arguments:

- query - The context that contains the query to be executed and whose matching records will be deleted.

Raised Exceptions:

- [Exception](#)

`deleteStream(streamId : String) : void`

Delete a stream from the repository.

Arguments:

- streamId - The stream identifier.

Raised Exceptions:

- [Exception](#)

`executeQuery(query : StreamQueryContext) : int`

Execute a query against the stream repository. Returns the number of results found that match the query, or -1 to indicate asynchronous query mode.

Arguments:

- query - The context that contains the query to be executed.

Raised Exceptions:

- [Exception](#)

`getCount(streamId : String) : int`

Retrieve the number of instances stored in the stream repository for the identified stream. Returns a Map with stream id as keys and counts as values.

Arguments:

- streamId - The stream id to retrieve the count for.

Raised Exceptions:

- [Exception](#)

`getMatchingSequences(streamId : String, sequenceContext : StreamSequenceContext) : List<Long>`

Retrieve the sequences for the given parameters. Returns the list of matching sequence identifiers.

Arguments:

- streamId - The stream identifier.

- sequenceContext - The context that defines what sequences to search for within the stream.

Raised Exceptions:

- [Exception](#)

`getUID() : String`

Retrieve the unique identifier for the Stream Repository implementation.

Raised Exceptions:

- [Exception](#)

`insert(insertionList : List<T>) : Object`

Insert a set of streaming data into the stream repository. Returns an Object that describes some result of the insert operation or null if nothing is returned.

Arguments:

- insertionList - The array of data to be inserted.

Raised Exceptions:

- [Exception](#)

`isStreamSupported(streamId : String) : boolean`

Check if the identified stream is supported by this repository. Returns True if this stream is being supported by this repository, False otherwise.

Arguments:

- streamId - The stream identifier.

Raised Exceptions:

- [Exception](#)

`listActiveQueryIds() : List<String>`

List the identifiers for the set of currently active queries. An active query is defined as any query that has processor cycles associated with it, i.e. a query is not finished 'executing' until all results have been delivered from the repository implementation to the output buffer to the dissemination service. Returns the list of active query identifiers.

Raised Exceptions:

- [Exception](#)

`openRepository() : void`

Initialize the connection to the repository and make the class instance ready for use in all respects.

Raised Exceptions:

- [Exception](#)

```
removeStreamSupport(streamId : String) : void
```

Remove the identified stream from the list of streams supported by this repository.

Arguments:

- streamId - The stream identifier.

Raised Exceptions:

- [Exception](#)

StreamRepositoryService

The Stream Repository Service extends the Stream Query Service interface and inserts frames into its associated data store(s). Although these interfaces are consistent with the general repository service, some methods are not functional, as the stream repository service is based on frames rather than information, which means that a) there is no type associated with the data, although it is correlated with the type associated with the stream context, b) there are no contexts for the frames, and c) frames are based around streams, which means that the repository should have streams as representative of the types of data, encapsulating methods which can retrieve the schema definitions for the stream, metadata content, and payload content. There is no actual insert frames method defined as part of the service API. Instead, the Repository Service receives frames via channels, which it reads internally, making insertion an internal process. This decision was made to ensure the physical separation of control versus data interactions. The frame storage interface is an extension of the frame 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 data in a repository while aging data would be moved to an archive. This service may be implemented in such a way that it can be used as a wrapper for existing legacy data stores.

Public Operations

```
deleteRecords(sessionTrack : SessionTrack, queryStreamQueryContext :  
StreamQueryContext) : int
```

Delete records that match the provided query. Any consumer channels defined for the provided query are ignored. There are several possibilities that arise from using a QueryContext for this operation: If a predicate and streams are both specified the predicate is applied to only the specified streams and the matching records are deleted, If a predicate is specified but streams are not the predicate is applied to all supported streams and the matching records are deleted, If no predicate is provided but a set of streams are specified all records of the specified streams are deleted, If neither a predicate not a set of streams are specified nothing happens and an exception is thrown. This is done because we specifically want to lock out the possibility of the default case being to delete all records for all supported types.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- queryStreamQueryContext - The context that defines the subset of records to be deleted.

Raised Exceptions:

- [Exception](#)

```
deleteStream(sessionTrack : SessionTrack, streamIdentifier : String) : void
```

This method will tell the service to permanently remove the data store for the identified stream. This method should fail if the repository is currently storing data for the specified stream (i.e. “end” method must be called first, before a “remove” call is executed).

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamIdentifier - The id of the stream to remove the resident data store for.

Raised Exceptions:

- [Exception](#)

```
isStreamSupported(sessionTrack : SessionTrack, streamIdentifier : String) :  
boolean
```

This method will check if the service supports storing the identified stream. This is a check to see if the stream has been registered with the repository via the startStoringDataForStream method. This method returns true if the stream is supported by this service, false if not.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamIdentifier - The identifier of the stream to determine support for.

Raised Exceptions:

- [Exception](#)

```
startStreamSupport(sessionTrack : SessionTrack, streamContext : StreamContext) :  
void
```

This method will register tell the service to begin storing streams of the identified type. If the stream does not have a location (XML container, database table, etc) to store the data in, one will be created. If a location already exists, that existing location will be appended to. If the desired functionality is to create a new store for an already registered stream, an actor should call the archive method, which will move the existing data store contents to another location.

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamContext - The context that describes the stream to begin storing.

Raised Exceptions:

- [Exception](#)

`stopStreamSupport(sessionTrack : SessionTrack, streamIdentifier : String) : void`

This method will tell the service to stop storing streams of the identified type. Any further data for this stream that are received will be ignored (dropped out of memory at processing time).

Arguments:

- sessionTrack - The pedigree of the invokers for this method.
- streamIdentifier - The identifier of the stream to stop storing data for.

Raised Exceptions:

- [Exception](#)

StreamRepositoryServiceConnector

This interface extends the Stream Repository Service interface, thereby exposing all of its methods on the connector side of the Phoenix control channel.

Public Operations

(Inherited from the Stream Repository Service)

StreamRepositoryServiceContext

A context for the stream repository service.

Public Operations

`getDefaultTableType() : mil.af.rl.phoenix.repository.TableType`

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

`getMaxRepositorySize() : long`

Retrieve 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`

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

`getSupportingStreamRepositories(streamId : String) : List<StreamRepository>`

Retrieve the repositories supporting the identified stream. Returns the list of stream repositories.

Arguments:

- streamId - The stream identifier.

Raised Exceptions:

- [Exception](#)

`isStreamSupported(streamId : String) : boolean`

Check if the identified stream is currently being supported. Returns True if the stream is being supported, False otherwise.

Arguments:

- streamId - The stream identifier.

Raised Exceptions:

- [Exception](#)

`startStreamSupport(streamContext : StreamContext) : StreamRepository`

Start supporting the identified stream. Returns the stream repository that will be used to support the stream.

Arguments:

- streamContext - The context describing the stream to begin supporting.

Raised Exceptions:

- [Exception](#)

`stopStreamSupport(streamId : String) : void`

Stop supporting the identified stream.

Arguments:

- streamId - The stream identifier.

Raised Exceptions:

- [Exception](#)

[StreamRepositoryServiceStub](#)

This interface extends the Stream Repository Service, thereby inheriting and exposing all of its methods on the stub side of the Phoenix control channel.

Public Operations

(Inherited from the Stream Repository Service)

StreamSequenceContext

Class description.

Public Operations

`getStreamSegmentQueryRange() : StreamSequenceRange`

Get the stream segment query range for the query to search by using the required sequence id component. Returns the stream segment query range.

`isSequenceRangeDefined() : boolean`

Check if the sequence range is defined or not. Returns True if the sequence range is constrained.

`setFrameQueryRange(streamQueryRange : StreamSequenceRange) : void`

Set the stream segment query range for the query to search. Constrains the range for bounding.

Arguments:

- streamQueryRange - The stream query range.

StreamSequenceRange

Defines the sequenceId range for a stream segment query.

Public Operations

`getEndRangeSequenceIdentifier() : long`

Get the sequence id to end the stream query over. Returns the sequence id to stop processing query.

`getStartRangeSequenceIdentifier() : long`

Get the sequence id to begin the stream query over. Returns the sequence id to begin query.

`isRangeEndDefined() : boolean`

The end range may be undefined, in which case the query should be over all specified streams beginning at the start range, if defined. Returns True or false, is the query constrained by a sequence id end to the range.

`isRangeStartDefined() : boolean`

The start range may be undefined, in which case the query should be over all specified streams until the end range, if defined. Returns True or false, is the query constrained by a sequence id start of the range.

`setEndRangeSequenceIdentifier(sequenceId : long) : void`

Set the sequence id to end the stream query over.

Arguments:

- sequenceId - Sequence id to stop processing query.

```
setStartRangeSequenceIdentifier(sequenceId : long) : void
```

Set the sequence id to begin the stream query over.

Arguments:

- sequenceId - Sequence id to begin query.

Reference

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. There are several identified types of actors: consumers, producers, and inquisitors. An actor may represent either a service or a user of a service.
Archive	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.
Client	An actor that is an application, system, or service that accesses another service.
Consumer	An actor that receives information.
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.
Endpoint	In Service-oriented architecture, an endpoint is the entry point to a service, a process, or a queue or topic destination. The ability to detect one or more Point of Presence/Entry/Connection to the information management system.
Information	The basic building block of data containing a minimum set of data including 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	An established category of information, the descriptor for which contains,

Type	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 an instance of information. 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 usage of data.
Payload	The actual unmodified data of interest.
Expression	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.
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 consistent with constraints and policies as specified by the service description (Organization for the Advancement of Structured Information Standards).

Acronyms

Acronym Definition

AFRL	Air Force Research Laboratory
ATO	Air Tasking Order
C2	Command and Control
COI	Community of Interest
CoT	Cursor-on-Target
DoD	Department of Defense
FORCH	Filter Orchestration
FOS	Filter Orchestration Service

GIG	Global Information Grid
IM	Information Management
IMS	Information Management Service(s)
IP	Internet Protocol
JB1	Joint Battlespace Infosphere
QoS	Quality of Service
RMI	Remote Method Invocation
SAB	Scientific Advisory Board
SOA	Service-Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language
XPath	XML Path Language
XSD	XML Schema Document

Interface Hierarchies

This section shows the parent and child relationships for each of the components of the Phoenix architecture that are specified in Section 3. For example, all of the service interfaces defined in Section 3.2 are extensions (children) of the BaseService interface, some directly, and some as children of children. In Section 3 all interfaces within the Phoenix architecture are presented in alphabetical order while in this section only those interfaces that are part of a parent-child hierarchy are shown.

BaseContext

- ActionContext
- ActorContext
- AuthorizationContext
- ChannelContext
- InformationChannelContext
- InputChannelContext

- ConnectionContext
- ConnectorContext
- StubContext
- EndPointContext
- EventContext
- EventDescriptorContext
- EventNotificationRequestContext
- ExpressionContext
- FilterContext
- FilterChainContext
- InformationContext
- InformationQueryContext
- InformationTypeContext
- ProtocolContext
- TransportProtocolContext
- ServiceBrokeringQueryContext
- ServiceContext
- ClientRuntimeServiceContext
- ConnectionServiceContext
- InformationServiceContext
- DisseminationServiceContext
- EventNotificationServiceContext
- InformationBrokeringServiceContext
- QueryServiceContext
- RepositoryServiceContext
- SubmissionServiceContext
- ServiceBrokeringServiceContext
- SessionManagementServiceContext
- StreamBrokeringServiceContext
- StreamDiscoveryServiceContext
- SubscriptionServiceContext
- SessionContext
- SubscriptionContext

ContextContainer

- BaseChannel
- BaseService
- BaseServiceConnector
- BaseServiceStub
- Event
- Filter
- FilterChain
- Information

BaseChannel

- InputChannel
- ByteInputChannel
- EventInputChannel
- FrameInputChannel
- InformationInputChannel*
- OutputChannel
- ByteOutputChannel
- EventOutputChannel
- FrameOutputChannel
- InformationOutputChannel*

** Dual Inheritance*

InformationChannel

----InformationInputChannel*
----InformationOutputChannel*

** Dual Inheritance*

BaseService

----BasePersistentService
----BaseChannelService
-----AuthorizationService
-----ClientRuntimeService
-----ConnectionService
-----DisseminationService
-----EventNotificationService
-----FilterManagementService
-----InformationBrokeringService
-----InformationDiscoveryService
-----InformationTypeManagementService
-----QueryService
-----RepositoryService
-----ServiceBrokeringService
-----SessionManagementService
-----StreamBrokeringService
-----StreamDiscoveryService
-----SubmissionService
-----SubscriptionService

BaseServiceConnector

----BaseChannelServiceConnector
-----AuthorizationServiceConnector
-----ClientRuntimeServiceConnector
-----ConnectionServiceConnector
-----DisseminationServiceConnector
-----EventNotificationServiceConnector
-----FilterManagementServiceConnector
-----InformationBrokeringServiceConnector
-----InformationDiscoveryServiceConnector
-----InformationTypeManagementServiceConnector
-----QueryServiceConnector
-----RepositoryServiceConnector
-----ServiceBrokeringServiceConnector
-----SessionManagementServiceConnector
-----StreamBrokeringServiceConnector
-----StreamDiscoveryServiceConnector
-----SubmissionServiceConnector
-----SubscriptionServiceConnector

BaseServiceStub

----BaseChannelServiceStub
-----AuthorizationServiceStub
-----ClientRuntimeServiceStub
-----ConnectionServiceStub

- DisseminationServiceStub
- EventNotificationServiceStub
- FilterManagementServiceStub
- InformationBrokeringServiceStub
- InformationDiscoveryServiceStub
- InformationTypeManagementServiceStub
- QueryServiceStub
- RepositoryServiceStub
- ServiceBrokeringServiceStub
- SessionManagementServiceStub
- StreamBrokeringServiceStub
- StreamDiscoveryServiceStub
- SubmissionServiceStub
- SubscriptionServiceStub

Event

- ExceptionEvent
- InformationEvent
- InformationDeliveryEvent
- InformationTypeEvent

Package Structure

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
-----	client
-----	connection
-----	core
-----	dissemination
-----	event
-----	eventnotification
-----	expression
-----	filter
-----	frame
-----	information
-----	informationbrokering
-----	informationdiscovery
-----	informationtype
-----	query
-----	repository
-----	security
-----	servicebrokering
-----	session
-----	sessionmanagement
-----	stream
-----	streambrokering

----- subscription

streamdiscovery
submission

How To...

How To...Submit Information

Required Phoenix IM Services :

- SubmissionService

Optional Phoenix IM Services :

- *InformationTypeManagementService* - may be used by the producer or SubmissionService to verify that an information type exists or to validate instances of information.
- *ServiceBrokeringService* - may be used by the producer and/or the SubmissionService to obtain control stubs for other Phoenix IM services.

Workflow

1. *The producer obtains a control stub instance for the SubmissionService.*
- This stub can be provided by a runtime library or brokered for via the ServiceBrokeringService.

```
/* This example assumes an implementation-specific control stub is provided  
by a runtime library. */  
SubmissionServiceStub serviceStub = new MySubmissionServiceStubImpl();
```

2. *The consumer must first activate and connect the stub to its associated service before it can be utilized.*
- This is done by calling the "activate()" and "connect()" methods on the stub. Activating a stub initializes its state within the local address space of the consumer. Connecting the stub to the service establishes the control channel between the consumer and the InformationBrokeringService.

```
/* activate the control stub. */  
serviceStub.activate(false);
```

```
/* Connect the stub to its associated service. This example assumes the  
absence of any SessionManagementService from the collection of Phoenix IM  
services being utilized. */  
serviceStub.connect(null);
```

3. *Create the Channel Context that describes the channel being requisitioned by the producer.*

- This context may contain specific configuration settings or a subset. If a subset then the rest of the channel settings will be determined by the service.

```
/* Create a Channel Context that will describe the Output Channel to be created. */
```

```
ChannelContext channelContext = new MyChannelContext();
```

```
// Set the location of the Submission Service the channel will connect to.
```

```
EndPointContext endPointContext = new MyEndPointContext();
```

```
endPointContext.setHostAddress(155.244.60.54);
```

```
endPointContext.setHostPort(1234);
```

```
channelContext.setEndPointContext(endPointContext);
```

```
TransportProtocolContext transportContext = new  
MyTransportProtocolContext();  
transportContext.setProtocolId("tcp");
```

```
ProtocolContext applicationContext = new MyProtocolContext();  
applicationContext.setProtocolId("information");
```

```
channelContext.setApplicationProtocolContext(applicationContext);  
channelContext.setTransportProtocolContext(transportContext);
```

4. *Create an instance of an OutputChannel for the selected SubmissionService.*

- The OutputChannel created can be a ByteOutputChannel, InformationOutputChannel, or an implementation-specific OutputChannel. OutputChannels are created by the actor connecting to the SubmissionService.

```
/* Request that the Submission Service double check the output channel  
context to insure that the created channel will match one of the service's  
available input channels. */
```

```
channelContext =  
serviceStub.configureActorOutputChannelContext(channelContext);
```

```
/* End of Channel Context configuration. */
```

```
/* Some implementation-specific mechanism then creates the channel from the  
channel context. In the case of this example a channel factory exists for  
this purpose. */
```

```
InformationOutputChannel ioc = (InformationOutputChannel)  
ChannelFactoryImpl.createOutputChannel(null, channelContext);
```

5. *Connect the OutputChannel to the SubmissionService.*
- This is done by calling the "connect()" method on the created OutputChannel. This operation will connect the OutputChannel to a corresponding SubmissionService InputChannel. the connect method takes a Session Track object as a parameter. This example assumes we have such an object already created and populated.

```
/* Connect the Output Channel to the Submission Service's Input Channel. */  
ioc.connect(mySessionTrack);
```

6. *Write information to the OutputChannel.*
- This is done by calling one of the "write()" methods provided by the OutputChannel. This method invocation realizes the concept of submitting information. If using a ByteOutputChannel or an implementation-specific channel, either the SubmissionService or a channel filter is responsible for converting the received data to Phoenix's defined notion of information, if conversion is required.

```
/* This example assumes you have one or more instances of Information ready  
to be submitted. */
```

```
Information information = new MyInformation();
```

```
List<Information> informationList= new ArrayList<Information>();  
informationList.add(information);
```

```
/* Submit information by invoking one of the write methods on the Output  
Channel. */
```

```
/* Option 1: Synchronous singular write. */
```

```
ioc.write(information);
```

```
/* Option 2: Synchronous multiple write. */
```

```
ioc.write(informationList);
```



```
/* Option 1: Asynchronous singular write. */  
ioc.writeAsync(information);
```

How To...Subscribe to Information

Required Phoenix IM Services :

- DisseminationService
- InformationBrokeringService
- SubscriptionService
- SubmissionService

Optional Phoenix IM Services :

- *InformationTypeManagementService* - may be used by the consumer or the required services to verify that an information type exists or to validate instances of information.
- *ServiceBrokeringService* - may be used by the consumer and/or the required services to obtain control stubs for other Phoenix IM services.
- *SessionManagementService* - can be used to create actor sessions for use when invoking methods on the Phoenix IM service interfaces. Alternately, a service's implementation could accept a null value for actor sessions.

Workflow

1. *The consumer obtains a control stub instance for the Subscription Service.*
- This binding can be provided by a runtime library or brokered for via the Service Brokering Service.

```
/* This example assumes an implementation-specific control stub is provided  
by a runtime library using static configuration. */  
SubscriptionServiceStub serviceStub = new MySubscriptionServiceStub();
```

2. *The consumer must first activate and connect the stub to its associated service before it can be utilized.*
- This is done by calling the "activate()" and "connect()" methods on the stub. Activating a stub initializes its state within the local address space of the consumer. Connecting the stub to the service establishes the control channel between the consumer and the Subscription Service.

```
/* activate the control stub. */  
serviceStub.activate(false);
```

```

/* Connect the stub to its associated service. This example assumes the
absence of any Session Management Service from the collection of Phoenix IM
services being utilized, hence the 'null' session track parameter. */
serviceStub.connect(null);

```

3. *Create the Channel Context that describes the channel to be used for delivering matching information.*

- This context may be contain specific configuration settings or a subset. If a subset then the rest of the channel settings will be determined by the service.

```

/* Create a Channel Context that will describe the Input Channel to be
created. */
ChannelContext channelContext = new MyChannelContext();

```

```

// Set the location of the channel that the Dissemination Service will
connect to.

```

```

EndPointContext endPointContext = new MyEndPointContext();

```

```

endPointContext.setHostAddress(155.244.60.69);

```

```

endPointContext.setHostPort(9786);

```

```

channelContext.setEndPointContext(endPointContext);

```

```

TransportProtocolContext      transportContext      =      new
MyTransportProtocolContext();
transportContext.setProtocolId("tcp");

```

```

ProtocolContext applicationContext = new MyProtocolContext();
applicationContext.setProtocolId("information");

```

```

channelContext.setApplicationProtocolContext(applicationContext);
channelContext.setTransportProtocolContext(transportContext);

```

4. *Setup the InformationInputChannel for receiving results.*
- This context may contain specific configuration settings or a subset. If a subset then the rest of the channel settings will be determined by the service.

```
/* Some implementation-specific mechanism creates the channel from the
channel context. In this example we use an implementation-specific channel
factory. */
```

```
InformationInputChannel ioc = (InformationInputChannel)
ChannelFactoryImpl.createInputChannel(null, channelContext);
```

```
/* Instruct the Input Channel to listen for connection attempts from Output
Channels. This example assumes no Session Management Service, hence the
'null' session track parameter. */
```

```
iic.open(null);
```

5. *Construct and populate the Context describing the subscription to be registered.*
- Subscriptions are registered through the use of SubscriptionContext instances.

```
/* Create a new SubscriptionContext and populate it with the required and
any optional attributes. */
```

```
SubscriptionContext subscriptionToBeRegistered = new
MySubscriptionContext();
```

```
/* (REQUIRED) Create and set the expression for the subscription. */
```

```
ExpressionContext expressionContext = new ExpressionContext();
```

```
expressionContext.setExpression("/us/af/aircraft[@tailNo='VIXEN03']");
expressionContext.setExpressionType("XPath");
```

```
subscriptionToBeRegistered.setExpression(expressionContext);
```

```
/* (REQUIRED) Set the EndPointContext for the consumer (at least one) of
the brokered information that matches this subscription. In this example we
add the context that describes the previously created input channel. */
```

```
subscriptionToBeRegistered.addConsumerChannel(channelContext);
```

```
/* (REQUIRED) Set the brokering result type. Either the registrant for this
subscription wishes to have information that matches the subscription test
forwarded to the associated consumers via some Phoenix Dissemination
```

```

Service or the registrant wishes to have consumer hit lists for brokered
information forwarded to the associated consumers via some Phoenix Event
Notification Service. */
subscriptionToBeRegistered.setBrokeringResultType(BrokeringResultType.INFOR
MATION);

```

```

/* (OPTIONAL) Set the time, in ms, for that the consumer(s) are willing to
wait for the first result that matches this subscription. If this time
expires without a match being found, the subscription will automatically be
unregistered. */
subscriptionToBeRegistered.setFirstMatchFoundTime(60000);

```

```

/* (OPTIONAL) Set one or more information type identifiers for the
information types that this subscription should be applied to. If not set,
the subscription will be applied to all types known to the registering
InformationBrokeringService(s). */
subscriptionToBeRegistered.addInformationTypeName("mil.af.aircraft");

```

6. *Register the subscription with the Subscription Service.*
 - The consumer then registers the subscription by submitting it to the Subscription Service.

```

/* Submit the subscription for registration by providing its Context object
to the Subscription Service. This example assumes the absence of any
Session Management Service from the collection of Phoenix IM services being
utilized. */
List<SubscriptionContext> subscriptionList = new
LinkedList<SubscriptionContext>();

```

```

subscriptionList.add(subscriptionToBeRegistered);

```

```

serviceStub.registerExpressions(null, subscriptionList);

```

7. *Use the previously created input channel to read information that matched the registered subscription.*
 - This may be done using any of the synchronous or asynchronous read methods from the input channel interface.

```

/* Invoke one of the read methods from the Input Channel interface or the
specific channel interface. */
/* Option 1: The asynchronous read from the input channel interface. */

```

```

iic.readAsync(new InputHandler<Information>());

```

```

/* Option 2: The singular, synchronous blocking read from the information
input channel interface. */

```

```
Information information = iic.read();
```

```
/* Option 3: The multiple, synchronous blocking read from the information
input channel interface. This example will read 5 information instances are
return them all at once in a List construct. */
List<Information> informationList = iic.read(5);
```

How To...Store Information

Required Phoenix IM Services :

- Repository Service
- Submission Service

Optional Phoenix IM Services :

- *InformationTypeManagementService* - may be used by the producer or required services to verify that an information type exists or to validate instances of information.
- *ServiceBrokeringService* - may be used by the producer and/or the required services to obtain bindings for other Phoenix IM services.

Workflow 1 : Submission Service is Configured to Store all Submitted Information

1. *The information producer submits information.*
- This is accomplished per the instructions given in the [How To...Submit Information](#) section.
2. *The Submission Service forwards the Information to the Repository Service.*
- This is accomplished via the internal information channels the services established between themselves. For more details, please see [Sequence Diagram 0008 Information Storage \(Persistence\)](#).

Workflow 2 : Producer Flags Individual Information Instances for Storage

1. *The producer flags an information instance for storage.*
- This is done by setting the corresponding flag within the InformationContext that describes the instance of information. This is not required and, based on the implementation, may or may not impact whether or not this information instance is stored in a repository.

```
/* This example assumes you have one or more instances of Information ready
to be submitted. */
Information information = new MyInformation();
```

```
/* Retrieve the Information Context for the information instance to be
stored. */
```

```

InformationContext      informationContext      =
information.getContext();

/* Set the persistence flag on the Context. This example assumes that the
value '1' tells the Submission Service to forward this Information instance
for storage. */
informationContext.setPersistenceFlag(1);

/* Set the Context for the instance of information. */
information.setContext(informationContext);

```

2. *The information producer submits information.*
- This is accomplished per the instructions given in the [How To...Submit Information](#) section. For this workflow the Information instance referenced in the information submission workflow and the information instance constructed above and flagged for storage are one and the same.
3. *The Submission Service forwards the information instance to the Repository Service.*
- This is accomplished via the internal information channels the services established between themselves. For more details, please see [Sequence Diagram 0008 Information Storage \(Persistence\)](#). The logic to determine whether or not to store an information instance may or may not utilize the aforementioned persistence flag from the Information Context.

How To...Query for Information

Required Phoenix IM Services :

- Query Service

Optional Phoenix IM Services :

- *InformationTypeManagementService* - may be used by the inquisitor or the Query Service to verify that an information type exists or to validate retrieved instances of information.
- *ServiceBrokeringService* - may be used by the inquisitor and/or the Query Service to obtain control stubs for other Phoenix IM services.
- *SessionManagementService* - can be used to create actor sessions for use when invoking methods on the Phoenix IM service interfaces. Alternately, a service's implementation could accept a null value for actor sessions.

Workflow

1. *The inquisitor obtains a control stub instance for the Query Service.*
 - This binding can be provided by a runtime library or brokered for via the ServiceBrokerService.

```
/* This example assumes an implementation-specific control stub is provided
by a runtime library. */
QueryServiceStub controlStub = (QueryServiceStub)
serviceBinding.getControlStub();
```

2. *The inquisitor must first activate and connect the stub to its associated service before it can be utilized.*
 - This is done by calling the "activate()" and "connect()" methods on the stub. Activating a stub initializes its state within the local address space of the inquisitor. Connecting the stub to the service establishes the control channel between the inquisitor and the Query Service.

```
/* activate the control stub. */
controlStub.activate(false);
```

```
/* Connect the stub to its associated service. This example assumes the
absence of any SessionManagementService from the collection of Phoenix IM
services being utilized. */
controlStub.connect(null);
```

3. *Create the Channel Context that describes the channel being requisitioned by the inquisitor.*
 - This context may contain specific configuration settings or a subset. If a subset then the rest of the channel settings will be determined by the service.

```
/* Create a Channel Context that will describe the Input Channel to be
created. */
ChannelContext channelContext = new MyChannelContext();
```

```
// Set the location of the actor channel that the Dissemination Service
will connect to.
```

```
EndPointContext endPointContext = new MyEndPointContext();
```

```
endPointContext.setHostAddress(155.244.60.54);
```

```
endPointContext.setHostPort(4567);
```

```
channelContext.setEndPointContext(endPointContext);
```

```

TransportProtocolContext      transportContext      =      new
MyTransportProtocolContext();
transportContext.setProtocolId("tcp");

```

```

ProtocolContext applicationContext = new MyProtocolContext();
applicationContext.setProtocolId("information");

```

```

channelContext.setApplicationProtocolContext(applicationContext);
channelContext.setTransportProtocolContext(transportContext);

```

4. *Setup the Information Input Channel for receiving results.*
 - The Input Channel will be used to receive the results of the submitted query. The Query Service is responsible for interfacing with one or more Dissemination Services to create the associated Output Channel.

```

/* End of Channel Context configuration. */

```

```

/* Some implementation-specific mechanism then creates the channel from the
channel context. */

```

```

InformationInputChannel      ioc      =      (InformationInputChannel)
ChannelFactoryImpl.createInputChannel(null, channelContext);

```

```

/* Instruct the Input Channel to listen for connection attempts from Output
Channels. */

```

```

iic.accept();

```

5. *Construct the query to be submitted.*
 - This is done via the InformationQueryContext interface.

```

/* Create and populate a new InformationQueryContext instance. */
InformationQueryContext queryToSubmit = new InformationQueryContext();

```

```

/* (REQUIRED) Create and set the expression for the query. */
ExpressionContext expressionContext = new ExpressionContext();

```



```

expressionContext.setExpression("/us/af/aircraft[@tailNo='VIXEN03']");
expressionContext.setExpressionType("XPath");

/* (REQUIRED) Add the expression test to the query. */
queryToSubmit.addExpression(expressionContext);

/* (REQUIRED) Add a ChannelContext that describes the consumer channel. */
queryToSubmit.addConsumerChannel(channelContext);

/* (OPTIONAL) Set the time, in ms, the inquisitor is willing to wait for
the submitted query to returns its first result. */
queryToSubmit.setFirstResultReturnedTime(60000);

/* (OPTIONAL) Set the time, in ms, the inquisitor is willing to wait for
the entire result set for the submitted query. */
queryToSubmit.setAllResultsReturnedTime(300000);

/* (OPTIONAL) Set one or more information types, by identifier, for this
query to be executed against */
queryToSubmit.addInformationTypeName("mil.af.aircraft");

/* (OPTIONAL) Set the execution mode flag to notify the Query Service of
whether or not to return the result set size before sending any results
over the information channel. This example assumes that the value '1'
requires that the Query Service return the result set size. */
queryToSubmit.setExecutionModeFlag(1);

```

6. *Submit the query for execution.*
- This is done by invoking the "executeQuery()" method on the Query Service. The Query Service will setup the Output Channel for delivering results, execute the query against the data store(s), and return the result set size, if required.

```

/* Submit the query for execution. This example assumes the absence of any
SessionManagementService from the collection of Phoenix IM services being
utilized. */
int resultSetSize = controlStub.executeQuery(null, queryToSubmit);

```

7. *The Query Service will connect to the Inquisitor's Input Channel.*
- The Query Service creates an Output Channel and connects it to the Inquisitor's listening Input Channel. Then the Inquisitor will listen for information being written to the channel. This information is the submitted query's result set.

```
/* Read the result set through the input channel. */  
iic.readAsync(new InputHandler<Information>());
```

8. *The Query Service delivers the result set, then the inquisitor and the Query Service disconnect and destroy the Information Channel between themselves.*

```
/* The Inquisitor should destroy its Input Channel once the result set has  
been received. */  
iic.close(null);
```