# PHASE ONE: MSIX-REPOSITORY *FUNCTIONAL* SPECIFICATIONS v1.0

DEFINITION
The MSIX Repository is a secure data repository that takes requests from the client, processes it, stores the data and then sends the response back. Requests and responses conform to the MSIX v1.2 protocol.

Note: While one Repository can service many clients, the component will be referred to in the singular.
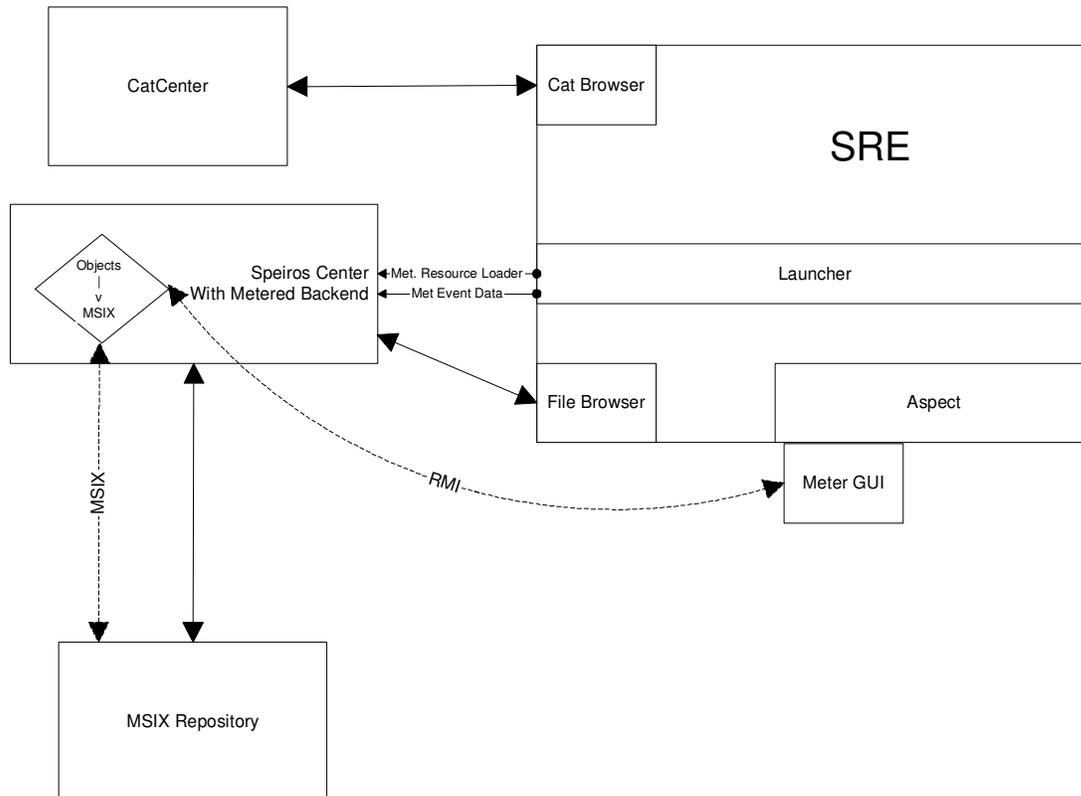
TERMINOLOGY
Client- The entity interfacing with the Repository.
MBES- Metering Back End Service. It acts as the middleman between the SRE (Client) and the MSIX (server).
MSIX- Metering Service Information eXchange. It is an XTP request-response based protocol used to transfer metering information between entities and was written by Alan Blount and Derek Young.
MSIX Repository- An application which handles MSIX requests, validates and stores the information, then sends back a response.
Service- A type of task that is performed by an application server for a client that can be measured by the MSIX.
Session- One instance of a particular service usage by an end user.
SRE- The Speiros Run-time Environment.

FEATURES
One of the major features for the Repository is it can service more than one client.

Other features include:

- δ It implements the MSIX v1.2 protocol.
- δ Cyrus Intersoft has extended the MSIX Protocol to include Query Session, which returns information about sessions or services.
- δ It can communicate with different types of JDBC compliant databases.
- δ Session data is not lost by service interruptions.
- δ Data is transmitted securely using SSL protocol.
- δ Access is controlled by user authentication.

FUNCTION
The Repository implements the **MSIX Transaction Protocol** for storing and retrieving metered event data. Also included in this section is an **overview** of how the Repository functions within the metering service as a whole.

1. MSIX Transaction Protocol
MSIX protocol involves a Service and a Session. The service of any particular type must first be defined in order for a session to start. Version 1.2 of the MSIX Protocol is implemented.

Note: To fully understand the MSIX Protocol, please refer to the MSIX Protocol Specifications v1.2.

2. Functional Overview
In order to offer a metered service, it must first be defined by the administrator of the MBES using the Service Tool. The MBES validates the service, sends an MSIX request to the **MSIX Repository** (which does its own validation), and if successful, records the service information. The Repository then sends back a response to the MBES. If the response is successful, the service is recorded on the MBES.

When the user launches a metered application, the Metered Resource Loader initiates the GUI as part of the client side support. The GUI contacts the MBES to get the cost information and displays it to the user. The user then can chose to continue and accept the charges, or cancel. If the user accepts it, the GUI contacts the Speiros Server to add them to the user group, giving it the permissions to load the Java jar files. The Resource Loader downloads the jar files, contacts the MBES to record the usage, and then removes them from the user group.

The MBES stores all the session information until the session is closed, at which point the session data is transferred to the Repository and removed from the MBES. For transactions with a per-use measurement, the session is immediately closed and the information is transferred immediately to the Repository since no more data will be needed.

Note: When the per-time measurement transaction is implemented in Phase Three development, the session will be opened when the user launches the metered application, updated at regular increments, and closed when the user is done. Then the session information is transferred to the Repository.

REQUIREMENTS
The Repository requirements consist of the following: Repository Startup, Processing Requests Cycle, Cyrus Enhancement to MSIX v1.2, Database Connectivity, and Data Reliability.

1. Repository Startup
Starting the Repository consists of the following requirements: the Certificate, Server Home Directory, Port, and the Default Properties Values.

1.1 Certificate
A digital certificate must first be set up between the client and the Repository.

1.2 Server Home Directory
The Repository starts with specifying the home directory on the listener.

1.3 Port
The Repository uses the default port number if the number is not supplied.

1.4 Default Properties Values
The Repository then reads the default properties values from the properties file or overrides the values with arguments supplied at the command line.

1.5 Command Recovery
The Repository checks the database for any unfinished commands (see section 5.2, **Database Recovery** for more information).

2. Processing Requests Cycle
Once the Repository is up and running, it can process requests from the client. The process follows the same **beginning steps** up to the point where the command is created and consists of the following requests: Define Service, Relate Service, Begin Session, Update Session, Commit Session, Abort Session, and Get Version.

Note: Validating Permissions is not part of the scope for Phase One.

2.1 Beginning Steps
The beginning steps consist of the following: Connection, Authentication, Parses Request, and Check Hash Table.

2.1.1 Connection- The Repository listens for a TCP/IP connection from the client and creates a socket. The connection is then either accepted or rejected.

2.1.2 Authentication- The Repository checks the client for valid or invalid SSL credentials for authentication.

2.1.3 Parses Request- The Repository parses the request and checks to see if it is in a valid or invalid MSIX format.

2.1.4 Check Hash Table-The Repository checks to see if the recovery hash table is empty. If it is not empty, it goes through the recovery process (see section 5, **Data Reliability**, for more information on recovery).

If the hash table is empty, the process continues with creating a command for the request.

2.2 Define Service Command
The Define Service Request allows repository clients to create new services that can be metered. The process continues through the final steps after the Repository creates the command for the request and consists of **Command Validation** and **Command Execution**.

2.2.1 Command Validation- Consists of the following requirements:

- δ Checking if the service distinguished name-version combination is unique.
- δ Checking for valid Ptypes.

2.2.2 Command Execution- Consists of the following requirements:

- δ Creating the Define Service object.
- δ Creating the Define Service recovery object.
- δ Storing the service data in the database.
- δ Writing to the Transaction log.
- δ Sending a response back to the client.

2.3 Relate Service Command

The Relate Service Request allows administrators to establish a relationship between two existing services. The process continues through the final steps after the Repository creates the command for the request and consists of **Command Validation** and **Command Execution**.

2.3.1 Command Validation- Consists of the following requirements:

- δ Checking the database for the existence of the Parent Service distinguished name.
- δ Checking the database for the existence of the Child Service distinguished name.
- δ Verifying the Child Service does not have multiple Parents (Note: a Parent can have multiple Children).
- δ Verifying the Parent-Child Service Relationship does not generate a loop (i.e. A is a Parent of B, B is the Parent of C, C is the Parent of A).

2.3.2 Command Execution- Consists of the following requirements:

- δ Linking the Child to the Parent in the database
- δ Storing the Service data in the database.
- δ Writing to the Transaction log.
- δ Sending a response back to the client.

2.4 Begin Session Command

The Begin Session Request creates an instance of a pre-existing service. The process continues through the final steps after the Repository creates the command for the request and consists of **Command Validation** and **Command Execution**.

2.4.1 Command Validation- Consists of the following requirements:

- δ Checking if the service distinguished name-version combination is in the database.
- δ Checking the database for the existence of the Parent ID if one is specified.
- δ Checking the database to validate the Parent service.
- δ Checking the database to see if the Session ID is unique.
- δ Checking the request properties to make sure they match the existing Ptypes of the service.

2.4.2 Command Execution- Consists of the following requirements:

- δ Creating the Begin session object.
- δ Creating the Begin session recovery object.
- δ Storing the session data in the database.
- δ Writing to the Transaction log.
- δ Sending a response back to the client.

2.5 Update Session Command
The Update Session Request updates an open session. The process continues through the final steps after the Repository creates the command for the request and consists of **Command Validation** and **Command Execution**.

2.5.1 Command Validation- Consists of the following requirements:

- δ Checking the database for the existence of an open session.
- δ Checking the request properties to make sure they match the existing Ptypes of the service.

2.5.2 Command Execution- Consists of the following requirements:

- δ Creating the Update session object.
- δ Creating the Update session recovery object.
- δ Storing the session data in the database.
- δ Writing to the Transaction log.
- δ Sending a response back to the client.

2.6 Commit Session Command
The Commit Session Request changes the session status from open to closed. The process continues through the final steps after the Repository creates the command for the request and consists of **Command Validation** and **Command Execution**.

2.6.1 Command Validation- Consists of checking the database for the existence of an open session.

2.6.2 Command Execution- Consists of the following requirements:

- δ Creating the Commit Session command session object.
- δ Creating the Commit Session command recovery object.
- δ Finding all the Children sessions and closing them as well as any open Descendents.
- δ Writing to the Transaction log.
- δ Sending a response back to the client.

2.7 Abort Session Command
The Abort Session Request aborts an existing open session, removing the information from the database. The process continues through the final steps after the Repository creates the command for the request and consists of **Command Validation** and **Command Execution**.

2.7.1 Command Validation- Consists of checking the database for the existence of an open session.

2.7.2 Command Execution- Consists of the following requirements:

- δ Creating the Abort session object.
- δ Creating the Abort session recovery object.
- δ Finding all the Children sessions and deleting them as well as any open Descendents.
- δ Writing to the Transaction log.
- δ Sending a response back to the client.

2.8. Get Version Command
The Get Version Request returns the current MSIX Protocol version to the Repository it is supporting. The process continues through the final steps after the Repository creates the command for the request and consists of sending a response back to the client indicating the MSIX protocol version it supports.

Note: As of this specification, the current version is v1.2.

3. Cyrus Enhancement to MSIX v1.2
Cyrus Intersoft has extended the MSIX Protocol to include Query, which returns information about sessions or services. The process continues through the final steps after the Repository creates the command for the request and consists of the following: Define Query, Query Service, Query Session and Send Response.

3.1 Define Query
Once the command is created, the next step is to determine whether the Query is for a session or service.

3.2 Query Service
The Repository checks the Service database for services matching the filter.

3.3 Query Session
The Repository checks the Closed Session database for sessions matching the filter.

3.4 Send Response
The Repository sends a response back to the client.

4. Database Connectivity
The Repository communicates with **JDBC** compliant databases to store and retrieve metering information. **Hypersonic SQL** is the default.

4.1 JDBC
JDBC driver information consists of the following:

	ORACLE	MS SQLServer	HypersonicSQL
Driver	Oracle JDBC Driver	Sun JDBC-ODBC Bridge and MS SQLServer ODBC Driver	HypersonicSQL Driver
TRANSACTION SUPPORT	YES	YES	NO
MULTITHREADING	YES	YES	NO

JDBC CALLS
java.sql.Connection.close()
java.sql.Connection.commit()
java.sql.Connection.getAutoCommit()
java.sql.Connection.prepareStatement(String)
java.sql.Connection.rollback()
java.sql.Connection.setAutoCommit(boolean)
java.sql.Connection.setTransactionIsolation(int)
java.sql.DriverManager.getConnection(String, String, String)
java.sql.PreparedStatement.clearParameters()
java.sql.PreparedStatement.close()
java.sql.PreparedStatement.execute()
java.sql.PreparedStatement.executeQuery()
java.sql.PreparedStatement.executeUpdate()
java.sql.PreparedStatement.setBoolean(int, boolean)
java.sql.PreparedStatement.setFloat(int, float)
java.sql.PreparedStatement.setInt(int, int)
java.sql.PreparedStatement.setMaxRows(int)
java.sql.PreparedStatement.setObject(int, Object)
java.sql.PreparedStatement.setString(int, String)

java.sql.PreparedStatement.setTimestamp(int, Timestamp)
java.sql.ResultSet.close()
java.sql.ResultSet.getBoolean(int)
java.sql.ResultSet.getBoolean(String)
java.sql.ResultSet.getInt(int)
java.sql.ResultSet.getInt(String)
java.sql.ResultSet.getObject(int)
java.sql.ResultSet.getObject(String)
java.sql.ResultSet.getString(int)
java.sql.ResultSet.getString(String)
java.sql.ResultSet.getTimestamp(int)
java.sql.ResultSet.getTimestamp(String)
java.sql.ResultSet.getStatement()
java.sql.ResultSet.next()

4.2 Hypersonic SQL

The Repository default database is Hypersonic SQL but can be reconfigured to utilize other databases locally or remotely using the appropriate JDBC drivers.

5. Data Reliability

The Repository can recover session and service data lost due to hardware or communication failures. Once the request has passed validation, the recovery object for the request is created and stored in the Recovery Table of the database.

The purpose of the recovery object is to remind the Repository of the request it was processing in case of failure. After the request has been successfully processed and a response is sent to the client, the recovery object is no longer needed and is deleted.

Data reliability consists of the following requirements: Process Recovery Commands and Database Recovery. The Repository then resumes normal processing.

5.1 Process Recovery Commands

Processing each command for Recovery consists of the following: Check Hash Table, Send Response, Write to Log, Delete Record, and Delete Recovery Object. The Repository then resumes the process for requests (see section 2, **Processing Request Cycle**, for more information the process).

5.1.1 Check Hash Table- Checking the hash table consists of the following:

- δ The Repository checks to see if the hash table is empty.
- δ If it is not, the Repository checks if the request ID matches the ID in the hash table.

5.1.2 Send Response- If the request ID is in the hash table, that means the client did not receive the original response. The Repository then sends the corresponding response from the hash table to the client and deletes the record from the hash table.

5.1.3 Write to Log- The Repository writes to the Transaction log.

5.1.4 Delete Record- The Repository deletes the record in the hash table.

5.1.5 Delete Recovery Object- The Repository deletes the service data in the database.

5.2. Database Recovery

Once service resumes after the interruption, the Repository checks the Recovery Table for recovery commands and retrieves them from the database. Each Recovery command is specific but finishes the process starting from the first state change.

COMPONENTS

The Repository consists of the following components: the Connection Handler, MSIX Library, MSIX Database, and the Activity Log.

1. Connection Handler

Receives Incoming requests and sends outgoing responses.

2. MSIX Library

These are the MSIX commands used by Repository.

3. MSIX Database

This database stores all the current request and session information.

4. Activity Log

This log keeps track of all the Repository activity. The file is stored under the home directory.