SUN SEEBEYOND

COMMON SERVICES FRAMEWORK USER'S GUIDE

Release 3.0.9

Sun microsystems

Version 20070619160535

Contents

Chapter 9

Chapter 10

Chapter 11

Chapter 12

List of Figures

Chapter 1

Common Services Framework (CSF)

Building common services, such as error handling, alerting and logging that can be used by all components has proven itself to be a best practice for many EAI projects. This reduces time to develop and deploy the solutions and standardizes support and maintenance. The Common Services Framework (CSF) provides a foundation for development of such common services. Although it is primarily intended for development of *technical* services, it could also serve as a foundation for implementing *business* services. The CSF fits seamlessly in the CAPS architecture leveraging the Service Oriented Architecture, Web Services and J2EE design patterns. CSF is an open platform and allows adding, removing and replacing services to meet specific client and project requirements. It is based on best practices and development standards and can meet most performance and scalability demands. A high level overview of the CSF and its core components is presented in Figure 1.

What's in this Chapter

1.1 About this Document

This document provides instructions on how to use the Common Services Framework. It covers the use of CSF APIs by the client applications, structure of the Common Message Envelope, used to carry service requests and framework GUIs.

1.1.1 Scope

This guide introduces you to CSF and gets you started using the product.

1.1.2 Intended Audience

This guide is intended for experienced computer users who are also knowledgable about Enterprise Application Integration (EAI). This includes the following:

- Project Managers
- System Architects

- Developers
- Administrators

1.1.3 Text Conventions

The following conventions are observed throughout this document.

Table 1 Text Conventions

Text Convention	Used For	Examples
Bold	Names of buttons, files, icons, parameters, variables, methods, menus, and objects	- Click **OK**. - On the **File** menu, click **Exit**. - Select the **eGate.sar** file.
`Monospaced`	Command line arguments, code samples; variables are shown in ***`bold italic`***	`java -jar filename.jar`
Blue bold	Hypertext links within document	See **Text Conventions** on page 9
Blue underlined	Hypertext links for Web addresses (URLs) or email addresses	http://www.sun.com

1.1.4 Screenshots Used in this Document

Depending on what products you have installed, and how they are configured, the screenshots in this document may differ from what you see on your system.

1.2 Related Documents

CSF Installation Guide

CSF Development Guide

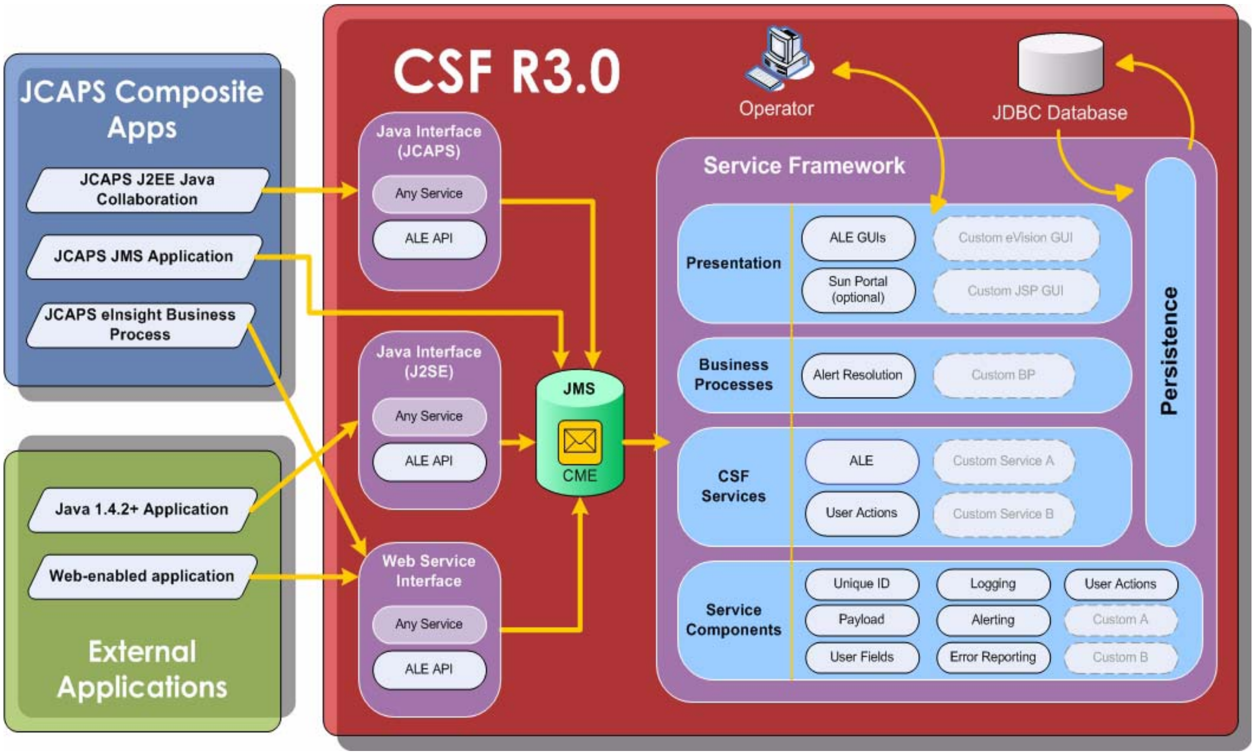
eGate Integrator User's Guide

eGate Integrator Tutorial

eGate Integrator Systems Administration Guide

ePortal Composer User's Guide

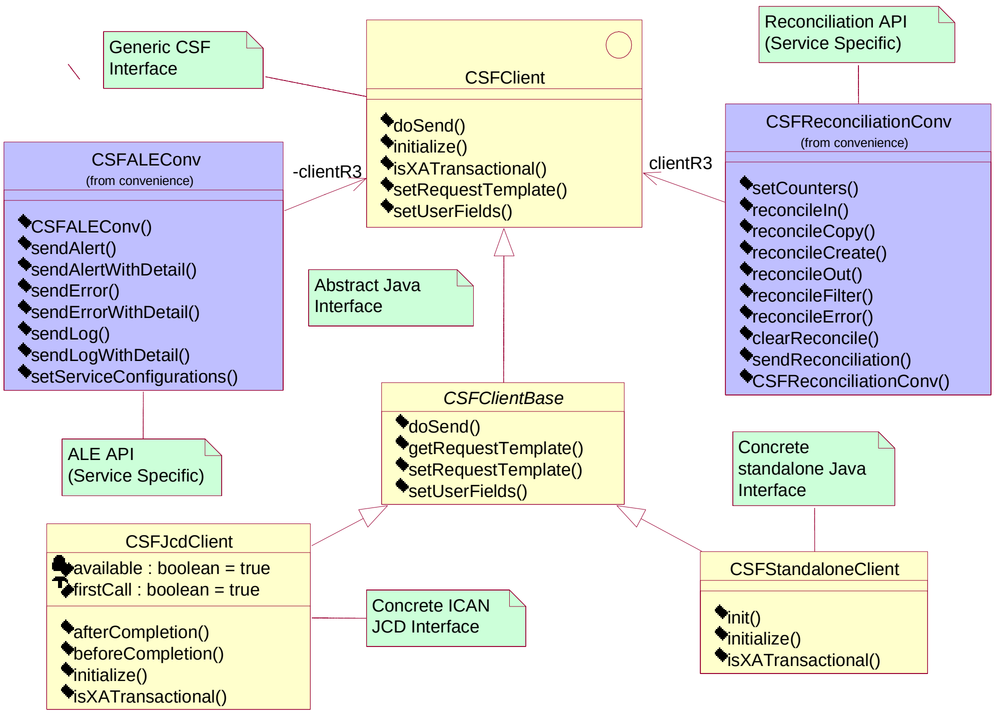
Figure 1 CSF Diagram



CSF includes a common platform and a number of *CSF Services* that can be invoked from client applications through one of the client interfaces. All client requests are passed in the form of the universal Common Message Envelope (CME) which is routed to the appropriate service. This decouples services from each other and CSF interfaces, allowing independent development of new services, and new types of interfaces, such as .NET or SRE. All CSF interfaces are generic and can be used to invoke any CSF Service. To facilitate ease of use, individual services can include their own *convenience* APIs, which act as a facade to one or more generic interfaces.

The primary way to invoke CSF services is through one of the Client Java Interfaces. The JCD interface should be used from Java Collaborations running inside CAPS, while Standalone interface is for external Java applications, servlets, etc. Both clients implement the same Java interface `CSFClient`, but have differences in the implementation, for example JCD client can participate in XA transactions. Figure 2 illustrates the relationships between two service-specific convenience APIs and two CSF Java Interfaces.

Figure 2 Java APIs to CSF



Similarly, generic Web Service CSF interface has a single operation `executeService` that takes a complete CME as an input parameter. ALE, Reconciliation, and User Actions Audit services each come with a convenience WS API that fronts the generic one and provides service-specific methods such as `sendAlert` and `sendLogwithPayload`.

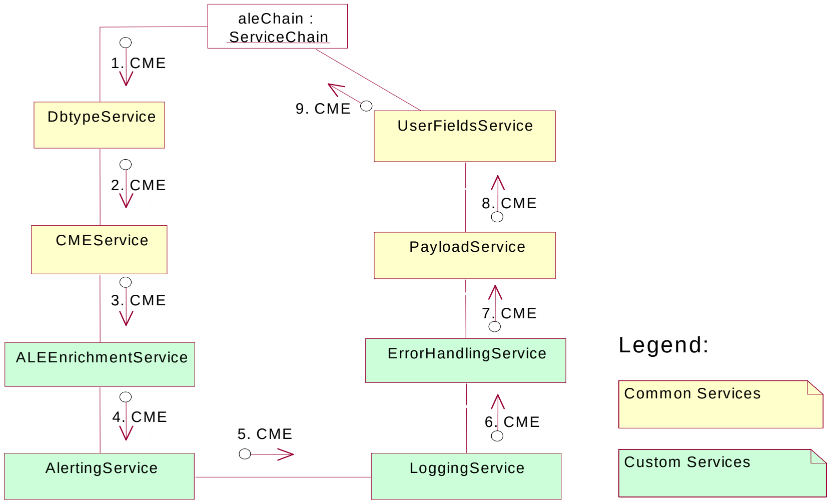
On the server side each SOA Service is realized by one or more *Implementation Services*. CSF comes with a number of reusable Implementation Services, for processing CME, Payloads, User Fields, etc. The framework facilitates the development of new SOA Services by allowing combining multiple Implementation Services using the Chain of Responsibility pattern as shown in the Figure 3 below.

CSF configuration information and run-time logs are stored in the Repository, which can be deployed on one of the supported database platforms. Currently, the standard release version supports Oracle 8 - 10g, DB2* and MS SQL Server, but it can be ported on any fully featured JDBC-accessible relational database. Access to the repository is abstracted through the persistence layer, which includes DAO and Value Objects, and additional helper classes and services.

CSF comes with a comprehensive set of GUIs, which allow framework users and administrators to access and manipulate configuration and run-time data for the framework and individual services. These GUIs are consolidated into a single personalized dashboard through ePortal, which allows adding, removing and configuring individual GUI components based on the deployed SOA Services.

Note: *Due to a product limitation of the JDBC eWay, the CSF GUI projects may not function properly when connected to DB2. To circumvent this issue, the GUI Projects can be re-implemented using the DB2 eWay*

Figure 3 Chaining Implementation Services to Create an SOA Service



1.2.1 Common Messaging Envelope

The use of CME is the key to CSF flexibility and extendibility. It contains the common information which is used by the framework and multiple services, while all service-specific details are encapsulated in a nested XML document. CME structure is broken into six sections, or nodes, that contain related fields. Complete information about each field is provided in **"CME Structure" on page 81**.

MessageHeader

`MessageHeader` is a required node, which contains globally unique `MessageID` that is used to identify all CSF requests and origination timestamp. Optional priority field allows to expedite message delivery and, depending on the destination service, processing.

SourceInfo

`SourceInfo` is a required node, which contains detailed information about the application that issued the request. It also contains optional Application message ID and timestamp. These fields are used to tie CSF requests back to the application transaction that triggered them, and allow correlate messages which belong to the same transaction. The detailed description of `SourceInfo` fields and how they are populated is provided **"CME Structure" on page 81**.

ServiceRequest

`ServiceRequest` is a required node, which specifies CSF service invocation. `RequestName` determines routing of the request to the target service(s). `ServiceLevel` provide a direct way to control message priority, by separating *priority* and *batch* requests. Specific differences between the two depend on the message delivery mechanism used in CSF deployment, as described **Message delivery** on page 13.

Optional *ParameterPayload* field is used to carry service-specific request parameters in the form of base64 encoded XML document. This mechanism allows strongly typed service invocation via service-independent generic interfaces. Each CSF Service may define the parameter format by providing an XSD and XML template associated with its request type. Optional *ServiceUserFields* node allows passing untyped, irregular or repeating parameters in the form of name-value pairs. It can be used instead of or in conjunction with *ParameterPayload*.

With the exception of *ServiceRequest* and *ServiceLevel* fields, is opaque to the framework and can be interpreted only by the target service(s).

Payload

Payload is an optional node, which allows CSF clients to attach own application data to their service requests. This data can be resubmitted back to the client application for error recovery or stored by the framework for audit. *PersistFlag* and *PersistMode* determine whether and how the payload is to be stored. *EncodeFlag* and *EncodeMode* control payload encoding. *OriginalMessage* and *TransformedMessage* fields are used to carry the application data before and after processing by the application as defined in the *PayloadType* field.

HeaderUserFields

HeaderUserFields is an optional node, which allows adding untyped data in the form of name-value pairs to the service requests. The key difference form *ServiceUserFields* node is that the former is visible to the framework and can be processed and persisted independently from the target services. This gives CSF users a quick way to extend the functionality of the framework as a whole, without changing the CME schema. For example these fields can be used to authenticate service requests.

VersionInfo

VersionInfo is a required node, which carries the information used to validate the compatibility between CSF clients and services. Field *CMEVersion* contains the version of CME schema and protocol, which is strictly enforced by the framework. Optional field *ClientVersion* contains the version of client interface that was used to create and dispatch the request; this information is not used by the framework, but can be useful in resolving the compatibility issues between CSF services, interfaces and convenience APIs.

Message delivery

CSF utilizes pluggable message delivery mechanisms to transport CME requests to the designated Services. The default delivery mechanism is a *two lane service bus* that consists of two JMS topics designated for *priority* and batch *service* level respectively. All CSF service requests are tagged with a specified *RequestName* and published to the topic corresponding to their *ServiceLevel*. CSF services read from one or both topics, filtering by the request names they are interested in. The distinction between two service levels is purely nominal and depends on the usage pattern, so designating request as *priority* does not guarantee faster delivery and processing. It is recommended that *priority* service level is only used for high priority, small and low volume messages. For example, all Reconciliation messages are sent as batch by default, and the ALE service sends alerts as priority and all other requests as batch.

The bus architecture and user fields in the CME allow easy enhancements to the CSF by piggybacking new services on the existing request types. For example, logger messages can also carry usage, billing and audit trail information.

The default delivery mechanism offers a good balance between performance and flexibility and should satisfy most end user requirements. When this is not the case, it can be easily replaced with a custom mechanism without affecting CSF clients or Service implementations. For example a *queue-per-service* delivery mechanism, why lacking the bus architecture flexibility, would allow multiple instances for the same service to process requests in parallel, thus providing improved scalability for high load environments.

Services Isolation and transparency

One of the key features of CSF is complete isolation of SOA Services from each other and the common platform. This allows development, modification, deployment and retirement of CSF services without affecting other services and, most importantly, clients that do not plan to use these services. This is achieved by maintaining careful separation between common and service-specific elements in CSF requests, interfaces, CAPS projects, and CSF repository.

Chapter 2

Integrating with the CSF

Since CSF is an open Services Platform it comes with a number of generic *Interfaces* for different kinds of clients (see Figure 1 above). These interfaces can be used to invoke any service; however the trade of for this universal access is that these interfaces are very fine-grained, untyped, and at times can be cumbersome to use. Service developers can define *Convenience APIs*, which facilitate service invocation, by providing coarse-grained access to service functions and strongly typed parameter passing.

What's in this Chapter

2.1 Interfaces and APIs

Convenience APIs serve as faÁades for the corresponding Interface. In CSF Interfaces are designed to expose the entire functionality of the framework and individual services, while APIs hide most of it, only exposing several commonly used operations for convenient single-call access.

CSF users are free to mix the two approaches using service APIs to access frequently used functionality and reverting back to the generic interface for more esoteric needs.

2.2 Java Clients

CSF provides two Java clients for use from CAPS JCDs and standalone Java applications. These clients implement a common interface *com.stc.csf.client.api.CSFClient* and from the client's point of view are almost indistinguishable. Once the appropriate client class has been instantiated, the same code would usually work against either of them. For example all convenience service APIs have a single implementation that works with both clients. Unless noted otherwise, the remainder of this section applies to both clients.

Table 2: CSF Client JARs

JAR file name	Purpose	Used in
`cme_otd.jar`	Contains classes generated from CME OTD.	CSF Server code
stccf_CAPSclient.jar	Contains generic Java Interface	CAPS Client.
stccf_j2seclient.jar	Contains generic Java Interface	Standalone Client.
stccf_properties.jar	Contains service-generic client properties for all Java clients.	All Java clients.
stccf_ale_res.jar	Contains property files, XML schema and template required by Alerting, Logging and Error Handling Service.	All Java clients, which use ALE Service.
stccf_ale_ez.jar	Contains ALE Service convenience API.	All Java clients, which access ALE Service through the convenience API.
stccf_recon_res.jar	Contains property files, XML schema and template required by Reconciliation Service.	All Java clients, which use Reconciliation Service.
stccf_recon_ez.jar	Contains Reconciliation Service convenience API.	All Java clients, which access Reconciliation Service through the convenience API.
stccf_UserActionsAudit_res.jar	Contains property files, XML schema and template required by User Actions Audit Service.	All Java clients, which use User Actions Audit Service.
stccf_UserActionsAudit_ez.jar	Contains User Actions AuditService convenience API.	All Java clients, which access User Actions Audit Service through the convenience API.
stccf_compat.jar	Contains CSF 2.0 compatibility API and all Java classes needed by the legacy client code. NOTE: legacy clients that use this library do not need to include any of the above JAR files.	CAPS Clients that were written to CSF 2.0.
stccf_compat_prop.jar	Contains all property files required by CSF 2.0 compatibility API.	CAPS Clients that were written to CSF 2.0.
stccsf_UserActionsAudit_res.jar	Contains property files, XML schema and template required by User Actions Audit Service.	All Java clients, which use User Actions Audit Service.

Table 2: CSF Client JARs (Continued)

JAR file name	Purpose	Used in
`stccsf_UserActionsAudit_ez.jar`	Contains User Actions Audit Service convenience API.	All Java clients, which access User Actions Audit Service through the convenience API.

All of these files, except the service-specific ones can be found in the project *SunCAPSCSFCore/SOA_Common/lib*. Service-specific files *stccf_<service>_ez.jar* and *stccf_<service>_res.jar* can be found in the project *SunCAPSCSFCore/SOA_Services/<servicename>/lib*.

In order to make CSF Interfaces and APIs to a Java client the JAR files that contain them must be added to the CLASSPATH of the JVM. When using the standalone client how to do this depends on the way client application is executed. Typically it can be done by adding them to the CLASSPATH environment variable or to the script used to launch the JVM. When standalone client is run from within a Java Application Server, please refer to the server documentation for information how to register external libraries.

In case of CAPS client, follow these steps to include each required CSF jar file when creating a new JCD that will call a CSF service:

- In the Java Collaboration Editor, click on the *Import Jar* button.
- Click *Add*.
- Navigate to the project, which contains the JAR file and select it.
- Click *Import*.

Refer to section “JAR File Consolidation” on page 73 for suggestions how to streamline library management by creating consolidated JAR files.

In order to use CSF Java interface, clients need to import *com.stc.csf.client.api.**, if they want to make use of convenience APIs or API consolidator they also need to import *com.stc.csf.client.convenience.**. CSF 2.0 legacy clients only need to import *com.stc.csf.CSFClient*.

The table in Figure 4 provides information about the property and resource files used by CSF clients. Please refer to section 12 for detailed information about CSF client configuration.

Note: *Please see the CAPS CSF Installation Guide for information on how to incorporate modified properties files into the CSF CAPS projects.*

Table 3: CSF Client Property Files

File name	Purpose	Packaged in
csfPriorityService.properties	Contains JMS parameters for delivering priority level service requests.	*stccf_properties.jar*

Table 3: CSF Client Property Files (Continued)

File name	Purpose	Packaged in
csfBatchService.properties	Contains JMS parameters for delivering batch level service requests.	*stccf_properties.jar*
csfj2seclient.properties	Contains additional JMS parameters, for standalone clients.	*stccf_properties.jar*
log4j.properties	Log4j configuration for the standalone client. If the client application is using log4j, this file is not required and the CSF client logging should be configured in the main log4j property file.	*stccf_properties.jar*
csf<RequestName>.properties	Contains default service level, transactionality, and priority for each request type.	*stccf_<service>_res.jar*
<RequestName>.xsd	Contains XML schema definition for each request type. **Users should never change this file.**	*stccf_<service>_res.jar*
<RequestName>.xml	Contains XML template for each request type. Users can modify the default values in this file.	*stccf_<service>_res.jar*

CSF Client Interface

Most client interactions with CSF occur through the interface *CSFClient* from package *com.stc.csf.client.api*. Typically client would obtain an instance of *CSFClient* from *CSFClientFactory*, initialize and configure it once and keep using it during its entire lifespan.

Client Initialization

Fists step is to obtain a copy of *CSFClientFactory* using static method *getInstance()*. The factory then creates the appropriate concrete client object in method:

CSFClient getClient(Object sourceApplication, String clientAppType)

Parameter *sourceApplication* is the object that will be using CSF. In case of the CAPS client it is the invoking JCD, which will be used to populate the *SourceInfo* fields, and provide access to the context objects, such as logger, alerter, etc. For standalone client it can be any object, and its type will be used as *SourceInfo.UnitName*. The second parameter determines the type of client to be created. At the moment only two client types defined in *CSFConstants*, are supported: *CSF_CAPS_JCD* for CAPS clients and *CSF_J2SE_APP* for standalone clients.

Newly created client then has to be initialized using one of the methods:

```
public void initialize() throws CSFException;
public void initialize(String messageFlowID) throws CSFException;
public void initialize(String messageFlowID, String appName)
throwsCSFException;
```

The difference between them is that the second method allows the application to define explicit transaction ID, which is then passed with each CME as *SourceInfo.MessageID* and can be used to correlate CSF requests caused by the same transaction. Similarly the third method allows the client to define both explicit transaction ID and application name. If a single *CSFClient* instance spans multiple transactions, this ID has to be reset after each one. The first method generates a unique ID value and should be used when client transaction ID is not known.

CSF Client utilizes *CSFRequest* object to accumulate data used populate the CME. These objects are created from a template to improve the performance and free clients from re-setting the values, such as *SourceInfo* fields, which do not change between requests. A default request template is created in the *initialize()* method and can be customized through the use of *getRequestTemplate()* and *setRequestTemplate()*.

Finally, if the user wants to use any convenience APIs, they need to be bound to the properly initialized client:

CSFALEConv aleAPI = new CSFALEConv(r3Client);

Atomic requests

CSFClient contains a number of convenient *send()* methods that create, populate, and dispatch CME in one step:

```
send(String serviceName, Map serviceParameters, String serviceLevel)
send(String serviceName, Map serviceParameters, String serviceLevel,
intmsgPriority)
send(String serviceName, Map serviceParameters, String strPayload,
StringserviceLevel)
send(String serviceName, Map serviceParameters, String strPayload,
StringserviceLevel, intmsgPriority)
```

Here *serviceName* is the Request name, such as ALE or Reconciliation; *serviceParameters* is a Map of parameter values used to populate Parameter Payload; *serviceLevel* is either *CSF_SERVICE_BATCH* or *CSF_SERVICE_PRIORITY*, as defined in *CSFConstants*; *msgPriority* can be used to override the default priority; and *strPayload* is used for attaching the application message to the CME. All *send()* methods return the CME message ID that was generated when sending the request.

Multi-step requests

CME has a very rich and diverse information model, where most values are optional, so providing single-step methods for all possible parameter combination would be impractical. When the information, that should be passed in a request does not match one of the signatures of the provided *send()* methods, the users can use more flexible multi-step protocol. They start with calling one of the *prepareRequest()* methods, that have similar signatures, parameters and semantics to the *send()* methods, with a difference, that they do not dispatch the request, but return it to the user in a form of a *CSFRequest* object. The user can then manipulate this object by setting the less frequently used parameters, such as transformed payloads, transactionality, etc. when all required information is configured, the populated request can be dispatched by calling method *doSend(CSFRequest rq)*.

Handling CSF Client Exceptions

All CSF client methods that can fail, throw `CSFException`. When a failure is considered recoverable (i.e. it might not reoccur if the operation is retried, for example JMS failure) `RecoverableException`, which is a sub-type of `CSFException`, is thrown. The suggested error handling logic is presented in the following code fragment:

```
try {
  csfClient.initialize();
  HashMap params = new HashMap();
  params.put( "LogUseFlag", "Y" );
  params.put( "LogCode", "200" );
  String msgId = csfClient.send("ALE", params, payload, ìBATCHî );
} catch (RecoverableException ex) {
  logger.warn("Attempting to recover CSF Client from problem: ", ex);
  // perform error recovery logic here
} catch (CSFException ex) {
  logger.error("CSF Client failed due to: ", ex);
  // perform error handling and cleanup here
}
logger.debug("CSF opration successfullî);
```

Please see section 3.2.4.4 on the differences in error handling between CAPS and standalone clients.

CSF Client utilizes exception chaining feature, so if more sophisticated error handling is required, root cause of the exception can be determined by calling method `getCause()`.

Differences between Clients

Since both `CSFJcdClient` and `CSFStandaloneClient` implement `CSFClient` interface and share a common base class, they behave almost identically when accessed through that interface, so in most cases the same client code would work both inside and outside of CAPS. This section highlights the differences between the two client implementations.

Transactionality

CAPS JCDs are transactional by default. This means that any JCD operations, including the sending of CSF messages via JMS from the CSF java client, can be rolled back in the event of an unhandled exception. This behavior differs significantly from that of ICAN 5.0.x, and should be noted.

If you wish to ensure that a CSF message is delivered from a JCD, even if an exception is encountered later in the transaction, it is reccommended that the CSF client call be made as part of a new transaction. To do this the JCD will need to implement the Runnable interface and a run() method will need to be created. This run() method will contain the instructions to be executed in the new transaction. An example of how to accomplish this is shown below:

```
public class jcdTestALEConv implements Runnable
...
// Send message in separate transaction: obtain a new instance of the
// same EJB and pass it a runnable object
javax.naming.Context ctx = new javax.naming.InitialContext();
 = null;
com.stc.codegen.framework.runtime.AssistantServiceHome h =
(com.stc.codegen.framework.runtime.AssistantServiceHome) ctx.lookup(
"java:comp/env/ejb/stc/deployedServices/AssistantServiceBean" );
com.stc.codegen.framework.runtime.AssistantServiceLocal ejb =
h.create();
```

```
ejb.assist( this );
...
public void run()
{
 try {
  String mesgID = "";
  if (!ALERequest_Request_1.hasPayload()) {
  mesgID = aleConv.sendErrorWithDetail(
  ALERequest_Request_1.getCode(), ALERequest_Request_1.getDetails(),
  ALERequest_Request_1.getDisplayMessage() );
 } else if

(ALERequest_Request_1.getPayload().getX_sequence_C().getOriginalMessa
ge() != null) {
  mesgID = aleConv.sendError( ALERequest_Request_1.getCode(),
  ALERequest_Request_1.getDisplayMessage(),

ALERequest_Request_1.getPayload().getX_sequence_C().getOriginalMessag
e() );
 }
 FileClient_1.setText( "Successfully executed ALE(" +
 ALERequest_Request_1.getName() + ") Service (Message ID: " +
 mesgID + ")" );
 FileClient_1.write();
} catch ( Exception e ) {
 logger.error( "Failed to execute ALE Service " + e.getMessage() );
 FileClient_1.setText( "Failed to execute ALE Service " +
 e.getMessage() );
 try {
  FileClient_1.write();
 } catch ( Exception fce ) {
 logger.error( "unable to update external system
 output file with previous log error message" );
 }
}
}
```

CAPS client can participate in XA transactions, so if the client fails, some requests can be rolled back and vice versa. Standalone client does not have such capability. In future other types of concrete clients can be developed for third-party application servers, which also support XA transactions.

CSFClient contains method *isXATransactional()* which returns *true* if the client has XA capability. However when this is the case, not all requests are sent as a part of the client transaction. Default transactional behavior is defined on the service level. The default behavior can be re-defined for each request, by calling method *setRequiresXA(boolean requiresXA)* on *CSFRequest*. The request is sent as a part of XA transaction only when:

```
CSFClient.isXATransactional() && CSFRequest.getRequiresXA() == true
```

Application Context

During the initialization, the CAPS client can populate most of *SourceInfo* fields from the *CollaborationContext*. Standalone client can only use application class name, so the user is responsible for providing the appropriate information for the *SourceInfo* fields (See “Client Initialization” on page 18.) This is best done through the request template mechanism described in “SourceInfo Values” on page 87.

Lifecycle

JCD client is implemented as a stateless session bean, and is re-initialized for each transaction. Consequently CSF client is re-initialized for every new message or batch. Some types of standalone clients, such as servlets, can be more durable, so the user has to take this into consideration, and reset some parameters, including transaction IDs, etc., before starting each new transaction.

Exception handling and recovery

CAPS automatically implements retry logic for failed JCDs. Consequently, CAPS client should let recoverable exceptions to fly out of the JCD, while unrecoverable exceptions should be caught, logged and swallowed in order to prevent infinite loops.

Standalone client might have a different retry mechanism or none at all. So in general, the reverse behavior is recommended: catch recoverable exceptions and retry the operation, but let unrecoverable exceptions to fly out and be handled by the system.

Logging

CAPS provides advanced logging and alerting services that are not available for standalone clients. In order to provide uniform functionality, all logging and alerting functions are delegated to the Messenger interface. CAPS client implements it with *`JcdMessenger`* that uses JCD Logger and Alerter from the *`CollaborationContext`*. Standalone client implements it with *`StandaloneMessenger`* that utilized log4j to emulate both logging and alerting. Its behavior can be configured through the log4j properties file.

2.3 Web Service Interface

Web Services interface to CSF specified in CSFService.wsdl contains port type CSFServicePort, which consists of a single operation executeService. This operation takes a single input message containing a CME document. It validates the request, fills in the fields that should be set on the server, and sends the request to the designated topic. Upon successful completion, the operation returns CSFResponse message that contains assigned Message ID and timestamp. If an error occurs, *`executeService`* returns a fault, containing description of the problem.

Chapter 3

CSF Repository

CSF uses the repository to persist its configuration information, run-time logs and resolution status information. CSF Repository can be deployed on any fully JDBC compliant database platform. Testing has been done with Oracle 9i and 10g so those databases are guaranteed compatible.

Note: *Recent testing has revealed a product limitation in the JDBC eWay when connected to DB2 via IBM drivers. The limitation manifests as DB connection errors when using the CSF eVision GUI projects. The Center of Excellence is pursuing a fix with SeeBeyond Development, but at the time of this release there is no resolution available. The GUI projects could, however, be reconfigured to use the DB2Connect eWay, which would permit a reliable connection.*

The CSF Repository resides in schema SBYN_CSF_R3 by default (please refer to the CAPS CSF Installation Guide for the details on creating, and populating the schema). The schema includes configuration tables (shown in green in the E/R diagrams below) and run-time tables (shown in beige). Configuration tables contain the information used by the framework and individual services in order to process the requests. This information is maintained by the framework administrators through CSF GUI or DBA tools. These tables should be populated prior to processing CSF requests. All Configuration tables have four Audit Trail Fields, specifying the user ID and timestamp of the original creator and the last updater. Most of them have *ACTIVE_FLAG*, which allows logical deletion of entries without affecting referential integrity.

What's in this Chapter

3.1 Common Tables

In order to support service isolation, CSF Repository schema is logically divided into the Common section, containing tables used by multiple services and the framework itself and service-specific sections containing tables used only by those services. To facilitate deployment/un-deployment of individual services, all database scripts are partitioned along these sections. This section covers the common tables (please see Figure 6 below), while service-specific tables are described in the corresponding sub-sections of chapter 7.

3.1.1 Configuration Tables

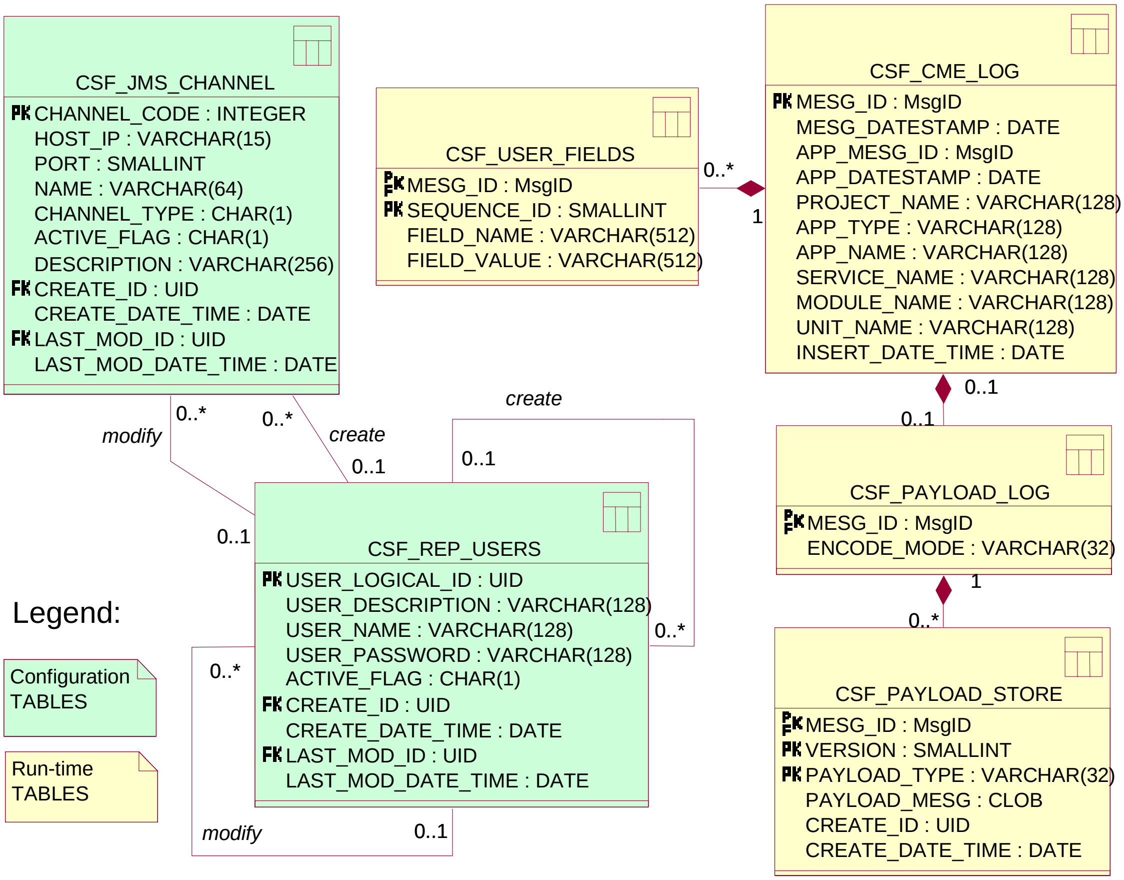
Table *CSF_REP_USERS* contains the information about framework users. It is used for user management, authentication and auditing purposes. When a Repository configuration table is modified, the *USER_LOGICAL_ID* is used for auditing purposes.

Table 4: CSF Repository Users

Column Name	Data Type	Description
USER_LOGICAL_ID	*VARCHAR(128)*	User ID used for logging into the GUI and audit purposes.
USER_DESCRIPTION	*VARCHAR(128)*	User role(s) or other administrative notes. May also contain contact information.
USER_NAME	*VARCHAR(128)*	Real name of the user.
USER_PASSWORD	*VARCHAR(128)*	Login password for the user (unencrypted)
ACTIVE_FLAG	*CHAR(1)*	Y or N. Flag to set this row active or not. This allows for easy activation/deactivation of users authorized to make table edits.
Audit trail fields		

Table *CSF_JMS_CHANNEL* contains the information about JMS endpoints. It is not currently used anywhere in CSF but will be used in a number of future services.

Figure 4 Common CSF Repository Schema



Run-time tables

Table *CSF_CME_LOG* contains the common information from CME that is repeated in all the logs. It is the root of the log hierarchy.

Table 5: CSF CME Log

Column Name	Data Type	Description	CME Field
MESG_ID	*VARCHAR(128)*	Unique message identifier generated by the CSF.	*MessageHeader.MessageID*
MESG_DATESTAMP	*DATE*	Date Time Stamp of Event from CSF.	*MessageHeader.DateTimeStamp*
APP_MESG_ID	*VARCHAR(128)*	ID of the message given by the source.	*SourceInfo.MessageID*

Table 5: CSF CME Log (Continued)

Column Name	Data Type	Description	CME Field
APP_DATESTAMP	*DATE*	Date Time Stamp of Event from source application.	*SourceInfo. DateTimeStamp*
PROJECT_NAME	*VARCHAR(128)*	Business area, e.g. EDI, Procurement, Eligibility	*SourceInfo. ProjectName*
APP_TYPE	*VARCHAR(128)*	Type of the client application, e.g. CAPS, J2SE, .Net, SRE.	*SourceInfo. ApplicationType*
APP_NAME	*VARCHAR(128)*	The name of a business application. This field will be used to reference application-specific configuration tables.	*SourceInfo. ApplicationName*
SERVICE_NAME	*VARCHAR(128)*	Represents a deployable set of functionality.	*SourceInfo. ServiceName*
MODULE_NAME	*VARCHAR(128)*	Represents a runnable unit, such as instantiation of a Service. In component-based architectures it could also be a Component.	*SourceInfo. ModuleName*
UNIT_NAME	*VARCHAR(128)*	Represents the lowest level identifiable logical component.	*SourceInfo. UnitName*
INSERT_DATE_TIME	*DATE*	Date Time Stamp of when the row was inserted. Also serves as a timestamp for all the linked tables.	

Tables *CSF_PAYLOAD_LOG* and *CSF_PAYLOAD_STORE* are used for persisting client message payloads that are submitted with CSF requests. The latter contains the actual payloads, while the former consolidates different types and versions of the payloads associated with the same CME request.

Table 6: CSF Payload Log

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	Foreign Key to *CSF_CME_LOG*.
ENCODE_MODE	*VARCHAR(32)*	NO ENCODING, ASCII/Text or BASE64. Defines encoding of associated payload CLOBs in *CSF_PAYLOAD_STORE*.

When message resubmission service is implemented, *CSF_PAYLOAD_STORE* table will also contain user-modified versions of payloads that were resubmitted into client applications. It only has half of the audit trail, since *LAST_MODOFIED_* fields are not applicable here - every time payload is changed for resubmission it is stored as another version.

Table 7: CSF Paylod Store

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	Foreign Key to *CSF_PAYLOAD_LOG*.
VERSION	*SMALLINT*	Used to distinguish original payloads from CME (VERSION= 0) from subsequent modifications for resubmission.
PAYLOAD_TYPE	*VARCHAR(32)*	ORIGINAL_MSG or TRANSFORMED_MSG. The type of payload from CME.
PAYLOAD_MESG	*CLOB*	The actual message payload. Optionally encoded as per *ENCODE_MODE* column in *CSF_PAYLOAD_LOG*.
CREATE_ID	*VARCHAR(128)*	Audit trail field.
CREATE_DATE_TIME	*DATE*	Audit trail field.

Finally table *CSF_USER_FIELDS* is used for persisting the HeaderUserFields from CME. There is no need to store ServiceUserFileds since they will be interpreted and stored if necessary by the services themselves.

Table 8: CSF User Fields

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	Foreign Key to *CSF_CME_LOG*.

Table 8: CSF User Fields (Continued)

Column Name	Data Type	Description
SEQUENCE_ID	*SMALLINT*	Field's index in CME.
FIELD_NAME	*VARCHAR(512)*	The name of the field.
FIELD_VALUE	*VARCHAR(512)*	The value of the field.

Chapter 4

Repository Sizing and Tuning

CSF repository has been designed with careful consideration regarding separation of services and balance between flexibility, performance and space considerations.

The default scripts that come with CSF construct the necessary schema, but administrators may wish to make optimizations to the database based on their unique usage of CSF.

4.1 Configuration tables

The configuration tables generally are small in size and fairly static in nature once they are defined. There are possible exceptions depending on the environment:

- *CSF_ERROR_CODES, CSF_ALERTER_CODES* and *CSF_LOGGER_CODES* – these tables define the behavior of the ALE services. Clients who have a complex system of codes representing their various errors, alert conditions, etc. may find that these tables can grow extremely large compared to the handful of rows in the sample data set. This should be taken into account when sizing the initial DB instance.

Depending on the services being deployed, the configuration tables are actively accessed during runtime for reading and referential integrity. These tables contain the processing information for handling the various CSF service types. A configuration table lookup occurs at least once during the processing of every CSF message. CSF is designed to permit changes to these processing instructions while system is running (*on the fly* configuration). In general, however, regardless of size, the code tables tend to have far more read operations than insert/updates and should be optimized for read performance.

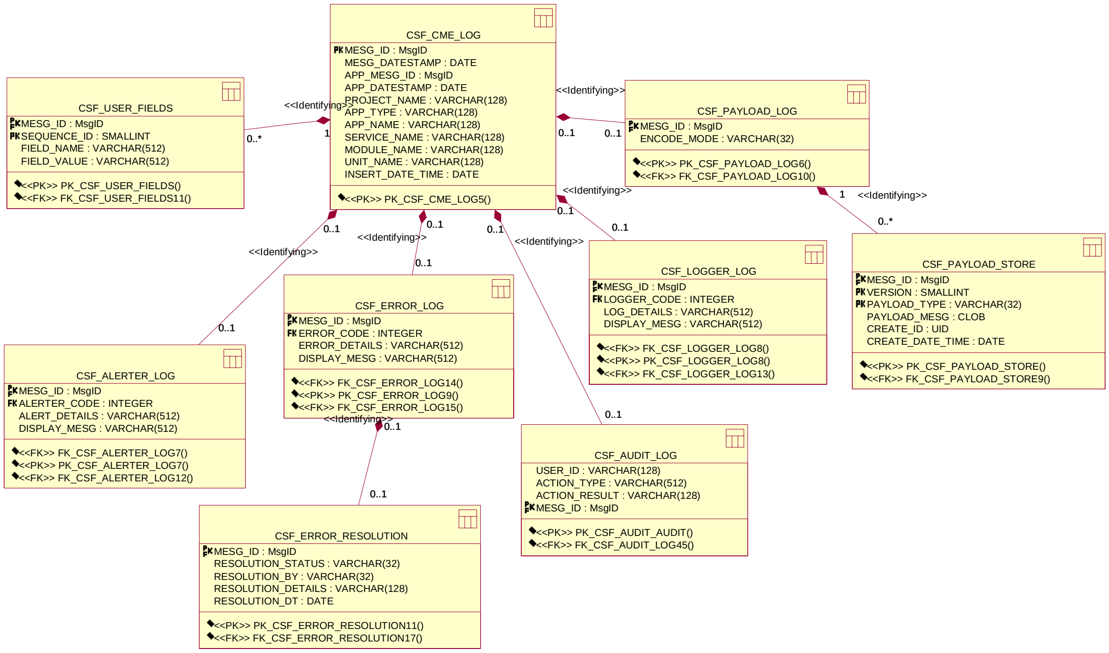
In addition, the codes defined in configuration tables are used for referential integrity with every log, error, alert, and audit message written to the run time tables.

Thus, DB Administrators may want to consider DB specific configuration techniques for improving performance for accessing the configurations tables. This may include placing configuration tables into faster storage area or specifying DB specific caching. As was stated before, in most environments, these tables are generally small in size and fairly static and do not require special optimizations.

4.2 Runtime messaging tables

The following diagram is a representation of the messaging run time tables and their relationships:

Figure 5 Messaging Runtime Tables


CSF_CME_LOG
PK MESG_ID : MsgID
MESG_DATESTAMP : DATE
APP_MESG_ID : MsgID
APP_DATESTAMP : DATE
PROJECT_NAME : VARCHAR(128)
APP_TYPE : VARCHAR(128)
APP_NAME : VARCHAR(128)
SERVICE_NAME : VARCHAR(128)
MODULE_NAME : VARCHAR(128)
UNIT_NAME : VARCHAR(128)
INSERT_DATE_TIME : DATE
<<PK>> PK_CSF_CME_LOG5()
CSF_USER_FIELDS
PFK MESG_ID : MsgID
PK SEQUENCE_ID : SMALLINT
FIELD_NAME : VARCHAR(512)
FIELD_VALUE : VARCHAR(512)
<<PK>> PK_CSF_USER_FIELDS()
<<FK>> FK_CSF_USER_FIELDS11()
CSF_PAYLOAD_LOG
PFK MESG_ID : MsgID
ENCODE_MODE : VARCHAR(32)
<<PK>> PK_CSF_PAYLOAD_LOG6()
<<FK>> FK_CSF_PAYLOAD_LOG10()
CSF_PAYLOAD_STORE
PFK MESG_ID : MsgID
PK VERSION : SMALLINT
PK PAYLOAD_TYPE : VARCHAR(32)
PAYLOAD_MESG : CLOB
CREATE_ID : UID
CREATE_DATE_TIME : DATE
<<PK>> PK_CSF_PAYLOAD_STORE()
<<FK>> FK_CSF_PAYLOAD_STORE9()
CSF_LOGGER_LOG
PFK MESG_ID : MsgID
FK LOGGER_CODE : INTEGER
LOG_DETAILS : VARCHAR(512)
DISPLAY_MESG : VARCHAR(512)
<<FK>> FK_CSF_LOGGER_LOG8()
<<PK>> PK_CSF_LOGGER_LOG8()
<<FK>> FK_CSF_LOGGER_LOG13()
CSF_ERROR_LOG
PFK MESG_ID : MsgID
FK ERROR_CODE : INTEGER
ERROR_DETAILS : VARCHAR(512)
DISPLAY_MESG : VARCHAR(512)
<<FK>> FK_CSF_ERROR_LOG14()
<<PK>> PK_CSF_ERROR_LOG9()
<<FK>> FK_CSF_ERROR_LOG15()
CSF_ALERTER_LOG
PFK MESG_ID : MsgID
FK ALERTER_CODE : INTEGER
ALERT_DETAILS : VARCHAR(512)
DISPLAY_MESG : VARCHAR(512)
<<FK>> FK_CSF_ALERTER_LOG7()
<<PK>> PK_CSF_ALERTER_LOG7()
<<FK>> FK_CSF_ALERTER_LOG12()
CSF_AUDIT_LOG
USER_ID : VARCHAR(128)
ACTION_TYPE : VARCHAR(512)
ACTION_RESULT : VARCHAR(128)
PFK MESG_ID : MsgID
<<PK>> PK_CSF_AUDIT_AUDIT()
<<FK>> FK_CSF_AUDIT_LOG45()
CSF_ERROR_RESOLUTION
PFK MESG_ID : MsgID
RESOLUTION_STATUS : VARCHAR(32)
RESOLUTION_BY : VARCHAR(32)
RESOLUTION_DETAILS : VARCHAR(128)
RESOLUTION_DT : DATE
<<PK>> PK_CSF_ERROR_RESOLUTION11()
<<FK>> FK_CSF_ERROR_RESOLUTION17()
<<Identifying>>
0..*
1
0..1
0..*

The sizing and performance considerations for run time tables are highly dependent on the environment, applications using CSF, and throughput placed on CSF by those applications. For the services shipped with CSF the major considerations are the number and rate of the log, error, alert and audit messages submitted by the applications. A special consideration should also be given to the storage of the payload and user fields that can be attached to every message.

In general the load on the CSF repository can be estimated ahead of time if applications can provide information on the CSF usage, the rate of incoming messages and the usage of the payload.

At a minimum, every message will create an entry in *CSF_CME_LOG* and one in *CSF_<SERVICE>_LOG*. *CSF_CME_LOG* is the root of the runtime tables. It maintains a MSGID that is referenced by other tables. So for the environments that do not create user fields and payloads these are the main tables that need to be taken into the consideration.

For the environments that take advantage of user fields, one *CSF_USER_FIELDS* record will be added for every user field in the message. So the DB administrator may want to take this into consideration when deciding on the size/percentage of journal and table file extensions.

Storage of the payload is one of the more serious considerations. The payload included with the message is stored only once in the *CSF_PAYLOAD_STORE* table. It is stored as CLOB and by default shares the same tablespace/storage file with all tables. *CSF_PAYLOAD_LOG* table is the reference to the actual payload and is used as an indication that payload exists. Additional entries in the *CSF_PAYLOAD_STORE* table will be created for every payload associated with the message. For example if there is original payload and transformed payload, it will be stored as separate entries.

Once the payload is written, it will be only accessed for viewing or replaying purposes by the user. So while it is important to provide users with adequate performance, these functions most likely will not place high throughput requirements on the system. The primary optimization goal for the Payload tables should be insert performance.

Administrators should discuss the payload requirements with the developers and pay close attention to the payload size, number of payload records and the rate with which these records are added. Additionally, administrators may want to consider how long the payload records should be maintained for. Offloading the payload records that are no longer used will not sacrifice the integrity of the repository, but will limit the functionality associated with these records.

Environments where large payload records are being saved, need to pay close attention to the storage requirements imposed by it. The storage should be sufficiently allocated, the expansion sizes for the payloads and for the journal should be properly defined or it may negatively affect overall CSF throughput. In some cases it may be prudent to place the payload in the separate storage space. Different DBs have specific ways of dealing with CLOB. For example they may allow storing an entry with up to 4000 bytes in row but anything larger than that will be moved to the separate space. Thus the size of the expected payload is an important consideration.

While the simple default configuration may be sufficient for number of environments, the requirements and loads places on CSF repository will be different for every implementation. It is important to pay close attention to these requirements, monitor the usage, tune the DB properly and take advantage of the DB utilities on perioic bases.

4.3 Scripts

In most environments administrators will want to create purge and backup utilities/scripts that will keep the DB in check. With large payloads and a high volume of records added, the size of the DB can grow very quickly.

Chapter 5

CSF GUI

CSF GUIs are Web based applications that provide a user friendly interface to configure CSF services and to examine data in the CSF Runtime Repository. As explained in chapter 4, CSF stores configuration settings in multiple database tables. As an alternative to updating the configuration tables using SQL or a database tool such as TOAD, CSF now provides a Configuration Management GUI.

CSF Configuration GUI is easy to use thanks to menu based navigation, drop-down lists with sets of possible, valid entries, data validation, and detailed field labels. Since the Configuration GUI is Web based, it can be accessed from any computer inside the corporate firewall using a standard browser. It can also be accessed from Internet if Web administrator has configured a proper URL mapping in the corporate firewall. User access is controlled by user name/password setup that is done in the ICAN deployment environment User Management screen.

What's in this Chapter

5.1 Overview

Each CSF service is represented by independent, modular, eVision based mini-applications. As will be explained further in this section, these multiple mini-applications can be aggregated by a portal and collectively presented to the users.

CSF GUIs are developed using **eInsight**, **eVision**, and **JDBC eWay**. All GUIs are currently located under `SeeBeyondCSFGUI` project hierarchy. Please refer to section 11.2 for details about the project structure for GUI applications. CSF GUIs are an optional part of CSF framework that can be installed independently from `SeeBeyondCSFCore` project. CSF also supports alternative means of configuration through either direct database updates or through custom developed applications that use JSPs. If you do choose to install CSF GUIs, each service specific GUI has a separate deployment profile that needs to be activated. You will typically find the deployment profile in top level `deploy` directory for each service specific GUI. For more details on installation please refer to CSF Installation and Configuration Guide. You can also check section 8.2 for suggestions on deployment consolidation.

Each service can have a number of mini-applications or portlets associated with it:

- Runtime GUI allows users to examine service data and service performance in CSF Runtime repository.
- Configuration GUI provides a user friendly way to configure a service in CSF repository.
- Additional optional GUIs can provide users additional access to service operations. For example, Alert Resolution GUI allows users to use Worklist Manager Interface to track resolution of alerts created by ALE service.

As new services are added to CSF, there could be a need to develop additional service specific GUIs for each service.

5.2 GUI Organization

Each CSF service *portlet* has a separate URL that can be used to access it as a self-contained eVision based mini-application. User access can be granted or revoked independently for each service. The table in Figure 7 presents sample URLs for GUIs that are provided as part of CSF release 3. Note that your specific URLs will vary based on logical host port number and host name selected for CSF.

Table 9: Sample CSF GUI URLs

Service	GUI Name	URL
ALE	ALE Configuration	*http://localhost:18001/CSFAleConfig*
ALE	ALE Runtime	*http://localhost:18001/CSFAleRuntime*
ALE	Alert Resolution	*http://localhost:18001/wlm*

You can access each service using above URLs. It is possible to create an Index page that lists all available services in an enterprise. However, the best approach is to use a portal to create an aggregate, personalized view of all services. Any portal software can be used with CSF GUIs. Refer to your portal vendor’s documentation for instructions on installing and configuring the portal. Using the URLs listed above you should be able to integrate the CSF GUIs into the portal easily.

Chapter 6

CSF Services

This section describes the services shipped with CSF 3.0.1. As new CSF services are released, they will come with their own set of documentation.

What's in this Chapter

6.1 ALE Service

- ALE is a combined service that provides Alerting, Logging, and Error Reporting functionality.
- Logging allows client applications to log events, messages, and milestones. It can be also be used for audit purposes. When used via ALE convenience API through an XA-capable interface logging is transactional, i.e. logged messages will not be delivered if the client fails after calling the log methods. This is different from the default behavior of the other ALE functions and this feature is especially useful for auditing CAPS applications.
- Alerting allows client applications to generate email notifications for critical events or system errors.
- Error Reporting allows clients to record exceptional conditions and error events

ALE functions use codes to identify their messages. These codes are defined in the CSF Repository. Each ALE event must be assigned a code which corresponds to an entry in its respective code table. The code tables define the behavior of all messages that are sent with a given code.

6.1.1 Java Convenience API

The ALE convenience API reduces the amount of code needed to invoke ALE services by adding service-specific API calls to the generic service-independent java API. The ALE convenience API is implemented by the class `CSFALEConv` from package `com.stc.csf.client.convenience`. A typical ALE convenience API example is shown here:

```
csfClient.initialize();
CSFALEConv aleAPI = new CSFALEConv(csfClient);
```

```
aleAPI.sendLog (‘200’, ‘Process A Completed Successfully’)
```

The example above creates an instance of the ALE convenience API class `CSFALEConv` and then uses the object to call ALE methods such as `sendLog`. Note that when the log message is sent, a log code of 200 is provided, which the ALE service will compare to the log code table to determine how this particular log message should be processed.

The following public methods are available in the `CSFALEConv` class:

```
String sendLog(String logCode)
String sendLog(String logCode, String logMesg)
String sendLogWithDetail(String logCode, String logDetail,
StringlogMesg)
String sendLog(String logCode, String logMesg, String logPayload)
String sendLog(String logCode, String logMesg, String
logPayloadOriginal, StringlogPayloadTransformed)
String sendLog(String logCode, String logMesg, Exception e)
String sendLog(String logCode, String logMesg, Exception e,
StringlogPayload)
String sendLog(String logCode, String logMesg, Exception e,
StringlogPayload)
String sendError(String errorCode, String errorMesg)
String sendErrorWithDetail(String errorCode, String errorDetail,
StringerrorMesg)
String sendError(String errorCode, String errorMesg, Exception e)
String sendError(String errorCode, String errorMesg,
StringerrorPayload)
String sendError(String errorCode, Exception e)
String sendError(String errorCode, Exception e, String errorPayload)
String sendAlert(String alertCode)
String sendAlert(String alertCode, String alertMesg)
String sendAlertWithDetail(String alertCode, String alertDetail,
StringalertMesg)
String sendAlert(String alertCode, String alertMesg,
StringalertPayload)
String sendAlert(String alertCode, Exception e)
String sendAlert(String alertCode, Exception e, String alertPayload)
void setServiceConfigurations(String serviceName, CSFRequest req)
```

To avoid *double-fault* problems Alert and Error methods do not throw any exceptions, but return `null` instead of message ID to indicate failure.

6.1.2 Web Service Convenience API

Web Service ALE Convenience API specified in *`ALEService.wsdl`* (which can be found in the `OTDFiles/Source` folder of the installation package) contains port type `ALEServicePort`, which consists of six operations:

Table 10: Web Services Convenience API Operations

Basic Operations	Operations with payload
sendLog	*sendLogwithPayload*
sendError	*sendErrorwithPayload*
sendAlert	*sendAlertwithPayload*

The standard operations take a two-part input message defined as *`ALERequestMessage`*:

```
<message name="ALERequestMessage">
  <part name="SourceInfo" element="csf:SourceInfo"/>
  <part name="aleRequest" element="req:ALERequest"/>
</message>
```

The *'withPayload'* operations take a three-part input message defined as *ALEwithPayloadMessage*:

```
<message name="ALEwithPayloadMessage">
  <part name="SourceInfo" element="csf:SourceInfo"/>
  <part name="aleRequest" element="req:ALERequest"/>
  <part name="alePayload" element="csf:Payload"/>
</message>
```

All of the operations return the same one-part message:

```
<message name="CSFResponseMessage">
  <part name="resposne" element="csf:CSFResponse"/>
</message>
```

This is a standard CSF Web Services response, that contains the CSF assigned Message ID and timestamp.

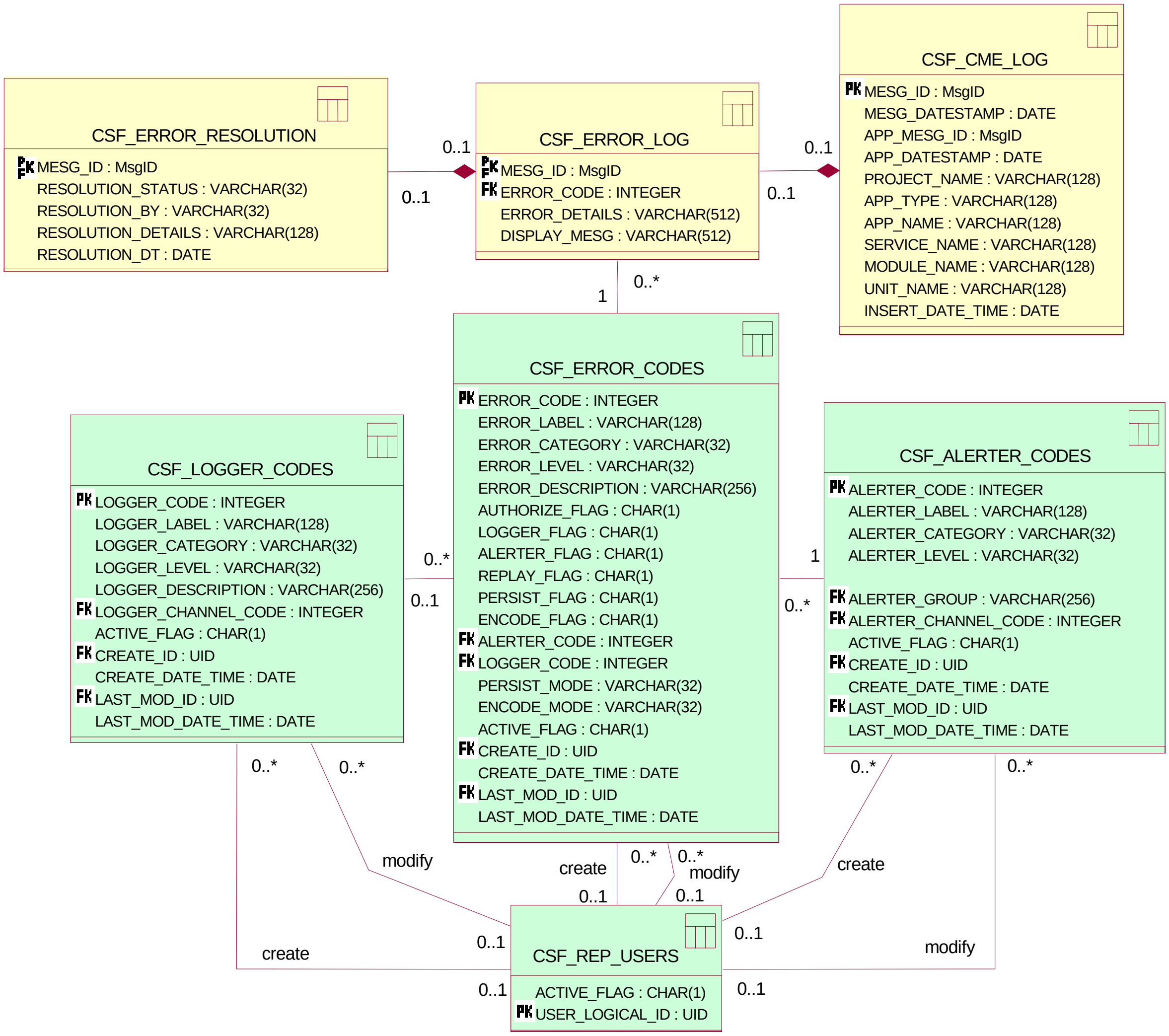
6.1.3 ALE Service Repository

The ALE Service Repository schema consists of 10 tables. The schema models and tables details are presented below. For simplicity, the schema is broken down into three sections along the three ALE functions.

Error Handling

The diagram in the following figure shows the Error Handling part of the ALE Service Schema.

Figure 6 ALE Service Repository Schema - Error Handling



The following *CSF_ERROR_CODES* table lists unique codes for each type of error that may be sent. Each error code can have its own unique behavior.

Table 11: CSF Error Codes

Column Name	Data Type	Description
ERROR_CODE	*INTEGER*	Unique numeric code identifying the error code.
ERROR_LABEL	*VARCHAR(128)*	Name describing the code.
ERROR_CATEGORY	*VARCHAR(32)*	SYSTEM or APPLICATION.

Table 11: CSF Error Codes (Continued)

Column Name	Data Type	Description
ERROR_LEVEL	*VARCHAR(32)*	One of the CSF standard severity levels: FATAL, CRITICAL, ERROR, INFO, WARNING, DEBUG.
ERROR_DESCRIPTION	*VARCHAR(256)*	Description of what type of error this code represents.
AUTHORIZE_FLAG	*CHAR(1)*	Y if authorization is required to process this error code N otherwise. Currently NOT USED.
LOGGER_FLAG	*CHAR(1)*	Y or N. Determines whether this message should automatically trigger a related Log message.
ALERTER_FLAG	*CHAR(1)*	Y or N. Determines whether this message should automatically trigger a related Alert message.
REPLAY_FLAG	*CHAR(1)*	Y if the original message should be re-sent when this error is encountered N otherwise. Reserved for future use.
PERSIST_FLAG	*CHAR(1)*	Y or N. Determines whether the payload associated with this type of error should be persisted. Only used if the corresponding value is not set in the CME.
ENCODE_FLAG	*CHAR(1)*	Y or N: Determines whether the payload associated with this type of error should be encoded. Used in conjunction with the corresponding value in the CME – the payload is encoded to the highest of the values.
ALERTER_CODE	*INTEGER*	Alert code to use when generating an automatic alert. Foreign Key to CSF_ALERTER_CODES.

Table 11: CSF Error Codes (Continued)

Column Name	Data Type	Description
LOGGER_CODE	*INTEGER*	Log code to use when generating an automatic log. Foreign Key to CSF_LOGGER_CODES.
PERSIST_MODE	*VARCHAR(32)*	FILE or DATABASE. Currently only DATABASE' is supported. Only used if the corresponding value is not set in the CME.
ENCODE_MODE	*VARCHAR(32)*	Types of encoding to be used on the payload if ENCODE_FLAG is set to Y. Valid values are: NOENCODING, ASCII/TEXT, or BASE64. Only used if the corresponding value is not set in the CME.
ACTIVE_FLAG	*CHAR(1)*	Y or N – designates whether this code can be used.
Audit trail fields		

The following *CSF_ERROR_LOG* is a runtime table which records all ALE Error events. It’s a child table of *CSF_CME_LOG*.

Table 12: CSF Error Log

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	The unique message ID assigned by CSF to this CSF message. Foreign Key to *CSF_CME_LOG*.
ERROR_CODE	*INTEGER*	Error code associated with this error. Foreign Key to *CSF_ERROR_CODES*.
ERROR_DETAILS	*VARCHAR(512)*	Descriptive text explaining the error.
DISPLAY_MESG	*VARCHAR(512)*	Short text describing the error.

The following *CSF_ERROR_RESOLUTION* is a runtime table which records resolution status of error messages captured in *CSF_ERROR_LOG*.

Table 13: CSF Error Resolution

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	The unique message ID assigned by CSF to this CSF message. Foreign Key to CSF_ERROR_LOG.
RESOLUTION_STATUS	*VARCHAR(32)*	Current status.
RESOLUTION_BY	*VARCHAR(32)*	This field contains self-description of the last person to resolve this error. It is populated by the ERROR Resolution GUI.
RESOLUTION_DETAILS	*VARCHAR(128)*	Text describing the resolution state.
RESOLUTION_DT	*DATE*	Timestamp of last resolution update.

Alerting

The diagram in Figure 7 shows the Alerting part of the ALE service schema.

The following *CSF_ALERTER_CHANNELS* table defines the channels by which alert messages can be delivered. Currently CSF delivers only via EMAIL channels.

Table 14: Alerter Channels

Column Name	Data Type	Description
ALERTER_CHANNEL_CODE	*INTEGER*	Unique numeric code identifying the channel.
CHANNEL_TYPE	*VARCHAR(128)*	EMAIL for email alerts. Currently only email alerts are supported.
HOST_NAME	*VARCHAR(256)*	Fully-qualified network name of the host machine.
COMMUNITY	*VARCHAR(32)*	SNMP Community.
TRAP_PORT	*VARCHAR(32)*	SNMP Trap port.
LISTENER_PORT	*VARCHAR(32)*	SNMP Listener port.
ACTIVE_FLAG	*CHAR(1)*	Y or N – designates whether this channel can be used.
***Audit trail field**s*		

The following *CSF_ALERTER_CODES* table lists unique codes for each type of alert that may be sent to CSF. Each alert code can have its own unique behavior.

Table 15: CSF Alerter Codes

Column Name	Data Type	Description
ALERTER_CODE	*INTEGER*	Unique numeric code identifying the alert code.
ALERTER_LABEL	*VARCHAR(128)*	Name describing the code.
ALERTER_CATEGORY	*VARCHAR(32)*	SYSTEM or APPLICATION.
ALERTER_LEVEL	*VARCHAR(32)*	One of the CSF standard severity levels: FATAL, CRITICAL, ERROR, INFO, WARNING, DEBUG.
ALERTER_DESCRIPTION	*VARCHAR(256)*	Description of what type of alert this code represents.
ALERTER_GROUP	*VARCHAR(256)*	The alerter group associated with this code. This determines who gets notified when an alert with this code is processed
ALERTER_CHANNEL_CODE	*INTEGER*	The alerter channel code for the channel through which notifications should be sent
ACTIVE_FLAG	*CHAR(1)*	Y or N – designates whether this code can be used.
Audit trail fields		

CSF_ALERTER_GROUPS table defines user groups that can receive alert notifications.

Table 16: CSF Alerter Groups

Column Name	Data Type	Description
ALERTER_GROUP	*VARCHAR(256)*	Name of the group
ALERTER_FROM	*VARCHAR(256)*	Email address to appear in the "From" address in alert emails
ALERTER_TO_PRIMARY	*VARCHAR(256)*	Email address of the primary recipient (use an email group on your email server to send to multiple primaries)

Table 16: CSF Alerter Groups (Continued)

Column Name	Data Type	Description
ALERTER_TO_SECONDARY	*VARCHAR(256)*	Email address of the secondary recipient (use an email group on your email server to send to multiple secondary recipients).
ACTIVE_FLAG	*CHAR(1)*	Y or N – designates whether this channel can be used.
Audit trail fields		

CSF_ALERTER_LOG is a runtime table which records all ALE Alert events. It's a child table of *CSF_CME_LOG*.

Table 17: CSF Alerter Log

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	The unique message ID assigned by CSF to this CSF message. Foreign Key to *CSF_CME_LOG*.
ALERTER_CODE	*INTEGER*	Alert code associated with this alert. Foreign Key to *CSF_ALERTER_CODES*.
ALERT_DETAILS	*VARCHAR(512)*	Descriptive text explaining the alert.
DISPLAY_MESG	*VARCHAR(512)*	Short text describing the alert.

The following *CSF_ALERT_RESOLUTION* is a runtime table which records resolution status of alert messages captured in the *CSF_ALERTER_LOG*.

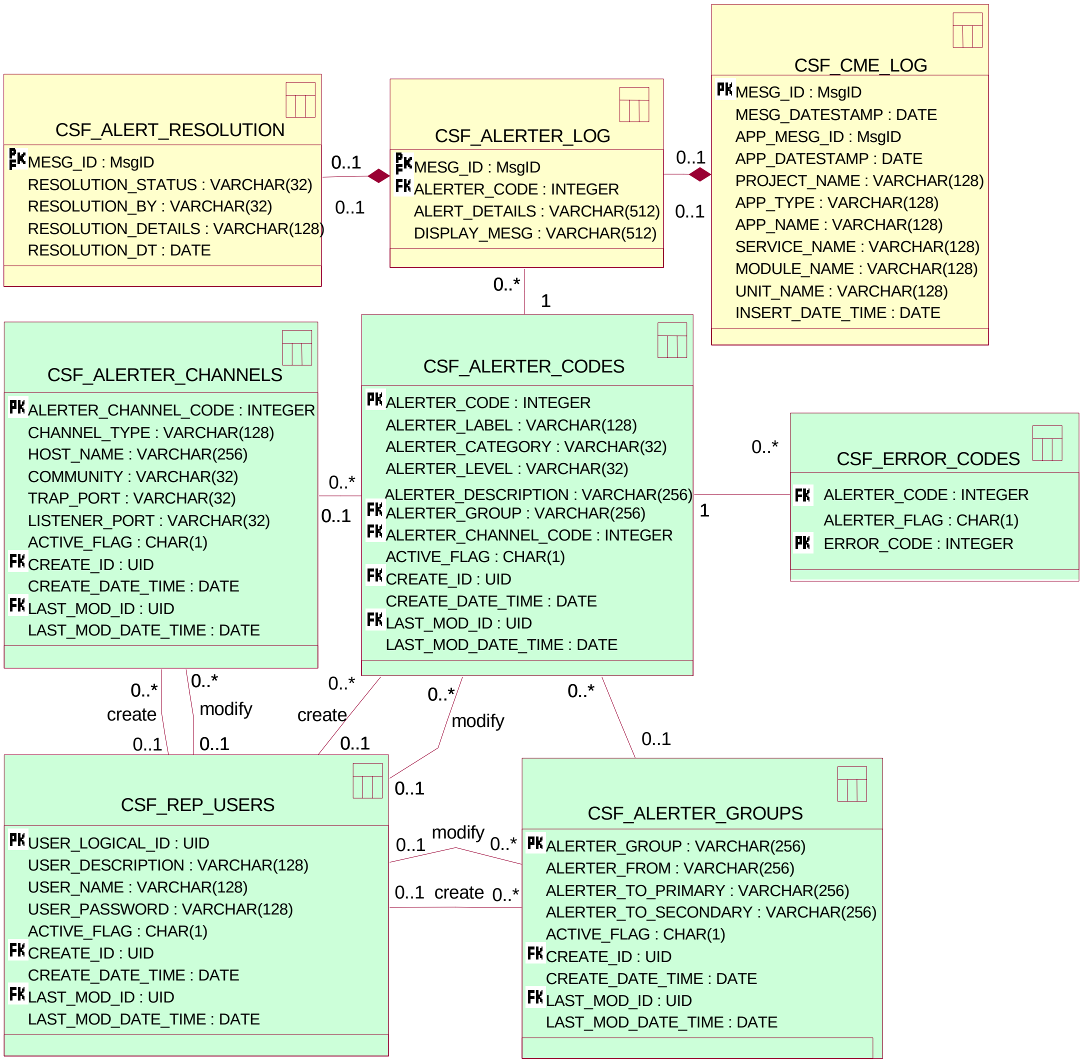
Table 18: CSF Alert Resolution

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	The unique message ID assigned by CSF to this CSF message. Foreign Key to *CSF_ALERTER_LOG*.
RESOLUTION_STATUS	*VARCHAR(32)*	Current status.

Table 18: CSF Alert Resolution (Continued)

Column Name	Data Type	Description
RESOLUTION_BY	*VARCHAR(32)*	This field contains self-description of the last person to resolve this error. It is populated by the Alert Resolution GUI.
RESOLUTION_DETAILS	*VARCHAR(128)*	Text describing the resolution state.
RESOLUTION_DT	*DATE*	Timestamp of last resolution update.

Figure 7 ALE Service Repository Schema - Alerting



Logging

The diagram in Figure 9 shows the Logging part of the ALE service schema.

The following *CSF_LOGGER_CHANNELS* table defines the channels by which log messages can be stored. Currently CSF delivers only allows two types of LOGGER channels, DB and FILE.

Table 19: CSF Logger Channels

Column Name	Data Type	Description
LOGGER_CHANNEL_CODE	*INTEGER*	Unique numeric code identifying the channel.
CHANNEL_TYPE	*VARCHAR(128)*	The target format of the logger service. FILE for a text file; DB for database (note: this replaced ORADB value from CSF 2.0).
FILE_NAME	*VARCHAR(256)*	Path and file name of the log file for storing log messages. Only used when CHANNEL_TYPE is FILE.
ACTIVE_FLAG	*CHAR(1)*	Y or N – designates whether this channel can be used.
Audit trail fields		

The following *CSF_LOGGER_CODES* table lists unique codes for each type of log message that may be sent to CSF. Each logger code can have its own unique behavior.

Table 20: CSF Logger Codes

Column Name	Data Type	Description
LOGGER_CODE	*INTEGER*	Unique numeric code identifying the logger code.
LOGGER_LABEL	*VARCHAR(128)*	Name describing the code.
LOGGER_CATEGORY	*VARCHAR(32)*	SYSTEM or APPLICATION
LOGGER_LEVEL	*VARCHAR(32)*	One of the CSF standard severity levels: FATAL, CRITICAL, ERROR, INFO, WARNING, DEBUG.
LOGGER_DESCRIPTION	*VARCHAR(256)*	Description of what type of event this code represents.

Table 20: CSF Logger Codes (Continued)

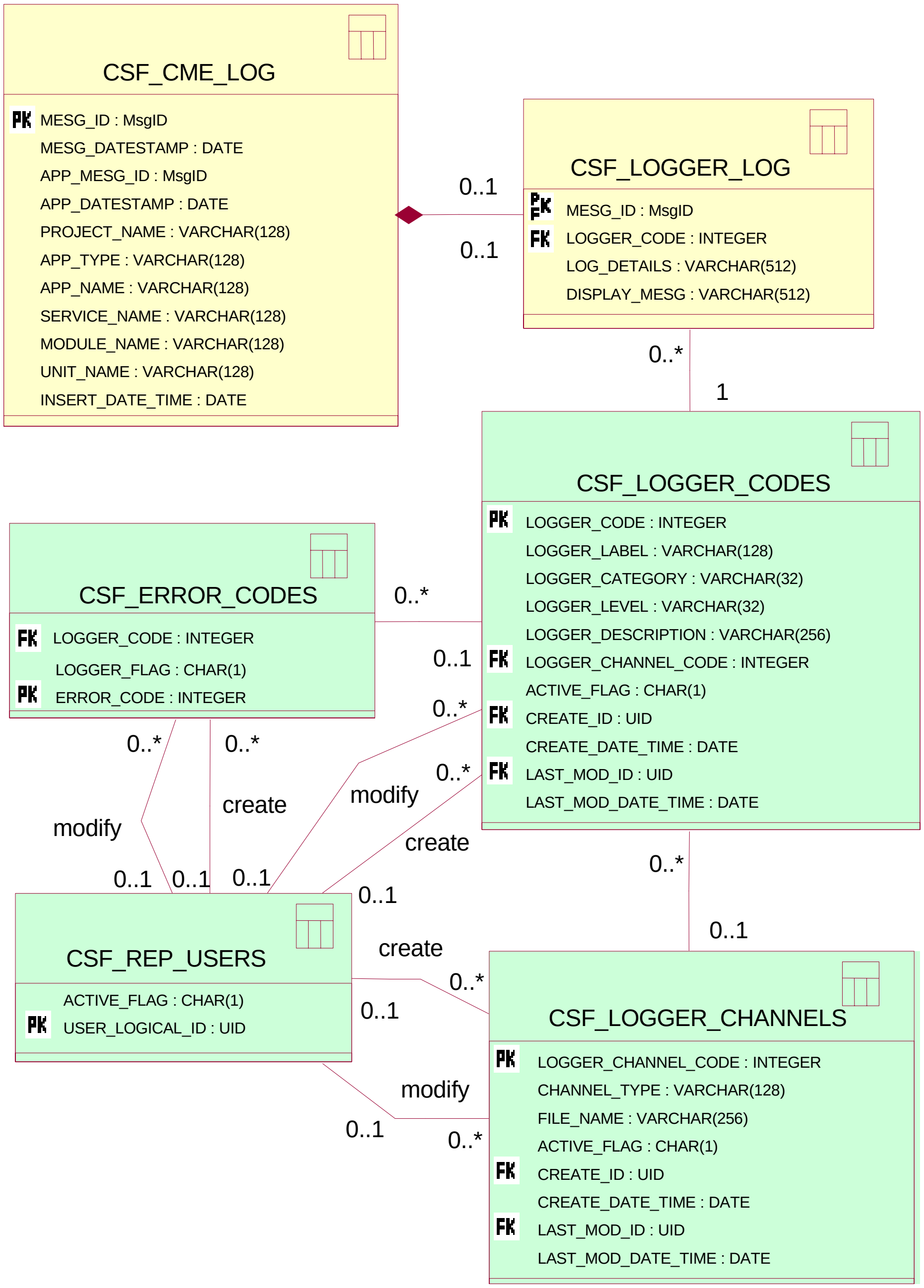
Column Name	Data Type	Description
LOGGER_CHANNEL_CODE	*INTEGER*	The log channel code for the channel by which logs will be stored. Foreign Key to *CSF_LOGGER_CHANNELS*.
ACTIVE_FLAG	*CHAR(1)*	Y or N – designates whether this code can be used.
Audit trail fields		

The following *CSF_ALERTER_LOG* is a runtime table which records all ALE Alert events. It is a child table of *CSF_CME_LOG*.

Table 21: CSF Alerter Log

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	The unique message ID assigned by CSF to this CSF message. Foreign Key to *CSF_CME_LOG*.
LOGGER_CODE	*INTEGER*	Logger code associated with this log record. Foreign Key to *CSF_LOGGER_CODES*.
LOG_DETAILS	*VARCHAR(512)*	Descriptive text explaining the log.
DISPLAY_MESG	*VARCHAR(512)*	Short text describing the log.

Figure 8 ALE Service Repository Schema - Logging



6.1.4 ALE GUI

The ALE GUI consists of two major components:

- Configuration management
- Runtime viewing and resolution

Configuration management part of the GUI is designed to simplify and provide administrative access to the configuration of the ALE component. It provides simple to use set of screen flows for populating the information in the configuration tables described in the section 7.1.3 including: *CSF_LOGGER_CODES*, *CSF_ERROR_CODES*, *CSF_ALERTER_CODES, CSF_LOGGER_CHANNELS, CSF_ALERTER_CHANNELS, CSF_ALERTER_GROUPS*.

Runtime portion of the GUI assists in selecting and viewing runtime messages and information. In essence it provides reporting capabilities.

Access to ALE functionality is protected through the use of user IDs and passwords.

Pre-Requisites

Installation of CAPS Suite including eVision and eInsight is required to deploy the ALE Service GUIs.

The GUIs are using JDBC and should, in theory, support any JDBC compliant DB compatible with JDBC eWay.

Before the GUIs can be started, the project needs to be deployed (see the *CSF Installation Guide* for deployment instructions).

ALE Configuration Management GUI

The Configuration Management GUI is a Web based application that provides a user friendly interface to configure the CSF Runtime Repository. It provides an alternative to updating the configuration tables using SQL directly. Users can still create the scripts for populating these tables in cases where it may be more appropriate, like part of the large production deployment.

CSF Configuration GUI is easy to use thanks to menu based navigation, drop-down lists with lists of possible, valid entries, data validation, and detailed field labels. It is based on eVision and designed to demonstrate the use of the best practices.

The Configuration GUI is Web based application and can be accessed from any computer inside the corporate firewall using a standard browser. It can also be accessed from Internet if Web administrator has configured a proper URL mapping in the corporate firewall. User access is controlled by user name and password which are configured in CAPS deployment environment through the User Management screen. The configuration can be accessed by using a URL similar to the following: **http://myHost:18001/CSFAleConfig/** where *myHost* and port will be based on your CSF deployment. MyHost should be substituted with the name of the host machine on which your logical host domain is running. The port number is usually the domain base port + 1

ALE Configuration GUI Navigation

ALE GUI configuration components provide well separated access to every configuration area. Access to the ALE configuration is protected via user ID and password.

To start the GUI click on the Configuration tab to navigate to Configuration Management screens. The screens provide access to the following areas that can be mapped to the configuration tables used by ALE.

Table 22: ALE Configuration Navigation

Button	Description
Configuration	Navigate to the main CSF Configuration menu.
Error Codes	View and edit error codes.
Logger Codes	View and edit logger codes.
Alerter Codes	View and edit alerter code.
Logger Channels	View and edit logger channels.
Alerter Channels	View and edit alerter channels.
Alerter Groups	View and edit alerter groups.
User Management	View and edit CSF user definitions.

The ALE configuration for each ALE area follows the same pattern. The top level screen provides a list of all items. Clicking on the link in the selected row opens a detailed view screen allowing to perform modifications. The new button on the top of the list screen opens up a detailed screen and provides ability to create new entry.

Error Codes Management

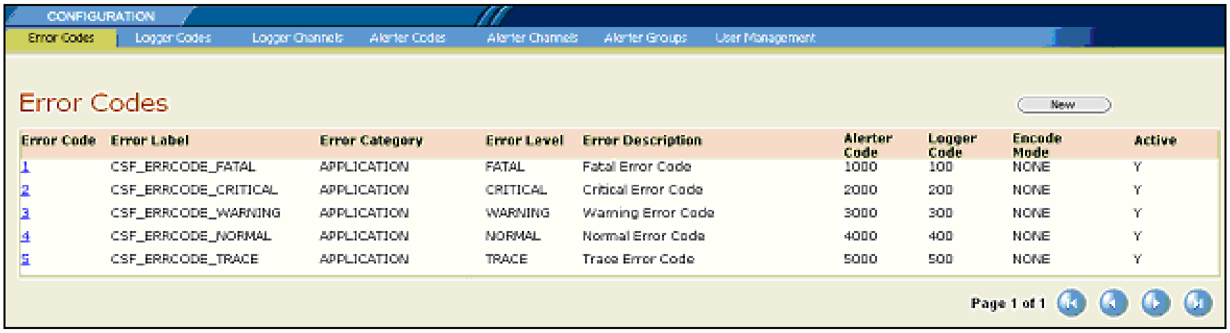
Error Code Management provides ability to view, change, activate, deactivate and create error codes. The error codes are related to the alerts and logs via alerter code and logger code respectively. Therefore, you may want to make sure that all the codes required by error codes are defined ahead of time.

1. Click on the Error Codes tab to view the list of all Error Codes from the *CSF_ERROR_CODES* table. The screen shows summary information for each error code defined in the CSF. This screen is read-only.
2. Clicking on error code link will drill down to the Error Code Detail screen. The detail screen displays detailed information about the selected error code and allows to modify any field. To save changes click Save and to revert changes to previous values click Reset. You may use the browser's *Back* button to return to the summary screen or you may click on Error Codes to display all error codes again. The list of values in Alerter Code and Logger Code dropdown lists is based on values defined for Alerter Codes and Logger Codes. These codes can be managed in the appropriate configuration management sections.
3. To define a new error code click the New button located on top of the Error Codes summary screen.

The error detail screen provides capability to specify all required information for the given error code. The capabilities of error code definition include: ability to specify if error code is active or deactivated, if it results in the alert, if it is persisted, logged and so on. These capabilities and definitions can be changed at runtime without affecting the rest of the environment.

The following two screens are sample illustrations of the error code view and definition screens:

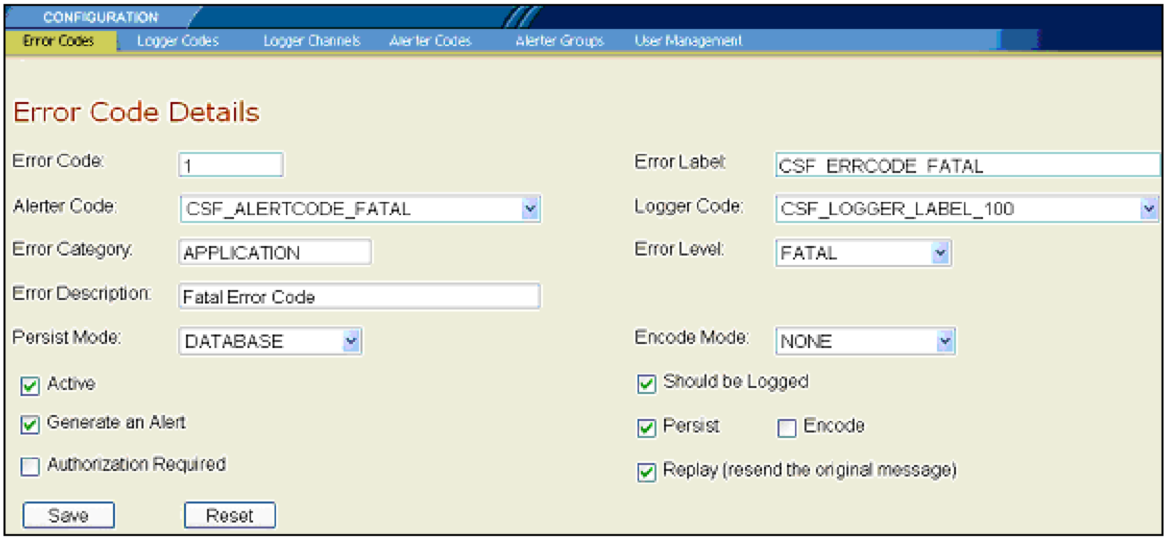
Figure 9 Error Codes Summary



Error Codes

Error Code	Error Label	Error Category	Error Level	Error Description	Alerter Code	Logger Code	Encode Mode	Active
1	CSF_ERRCODE_FATAL	APPLICATION	FATAL	Fatal Error Code	1000	100	NONE	Y
2	CSF_ERRCODE_CRITICAL	APPLICATION	CRITICAL	Critical Error Code	2000	200	NONE	Y
3	CSF_ERRCODE_WARNING	APPLICATION	WARNING	Warning Error Code	3000	300	NONE	Y
4	CSF_ERRCODE_NORMAL	APPLICATION	NORMAL	Normal Error Code	4000	400	NONE	Y
5	CSF_ERRCODE_TRACE	APPLICATION	TRACE	Trace Error Code	5000	500	NONE	Y

Figure 10 Error Code Details



Error Code Details

Error Code: 1

Error Label: CSF ERRCODE FATAL

Alerter Code: CSF_ALERTCODE_FATAL

Logger Code: CSF_LOGGER_LABEL_100

Error Category: APPLICATION

Error Level: FATAL

Error Description: Fatal Error Code

Persist Mode: DATABASE

Encode Mode: NONE

☑ Active

☑ Should be Logged

☑ Generate an Alert

☑ Persist □ Encode

□ Authorization Required

☑ Replay (resend the original message)

Logger Codes Management

Logger Code Management provides ability to view, change, activate, deactivate and create logger codes. The logger codes are used by errors as well. Log messages are stored according to the specified logger channel definition determined through the mapping for the log message at runtime.

The logger can also be used for auditing purposes. One can define a code and a channel for audit messages. The audit messages can then be filtered for separate processing.

1 Click the Logger Codes tab to view the list of all Logger Codes from the *CSF_LOGGER_CODES* table. The screen shows summary information for each logger code defined in the CSF. This screen is read-only.

2 Clicking the Logger Codes link will drill down to the Logger Code Detail screen. The detail screen displays detailed information about the selected logger code and allows to modify any field. To save changes click *Save* and to revert changes to previous values click Reset. You may use the browser's *Back* button to return to the summary screen or you may click the Logger Codes to display all logger codes again. The list of values in Logger Channel

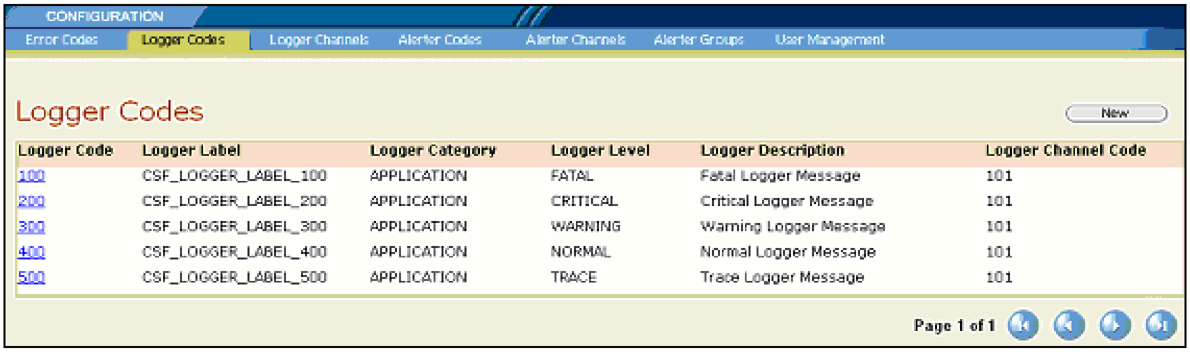
dropdown list is based on values defined for the Logger Channels. Thus it is suggested to define the channels before the logger codes; otherwise you will need to update the channel mapping at a later time.

3 To define a new logger code click the New button located on top of the Logger Codes summary screen.

The logger detail screen provides capability to specify all required information for the given logger code. Logger code definition may be active or deactivated.

The following two screens are sample illustrations of the logger code view and definition screens:

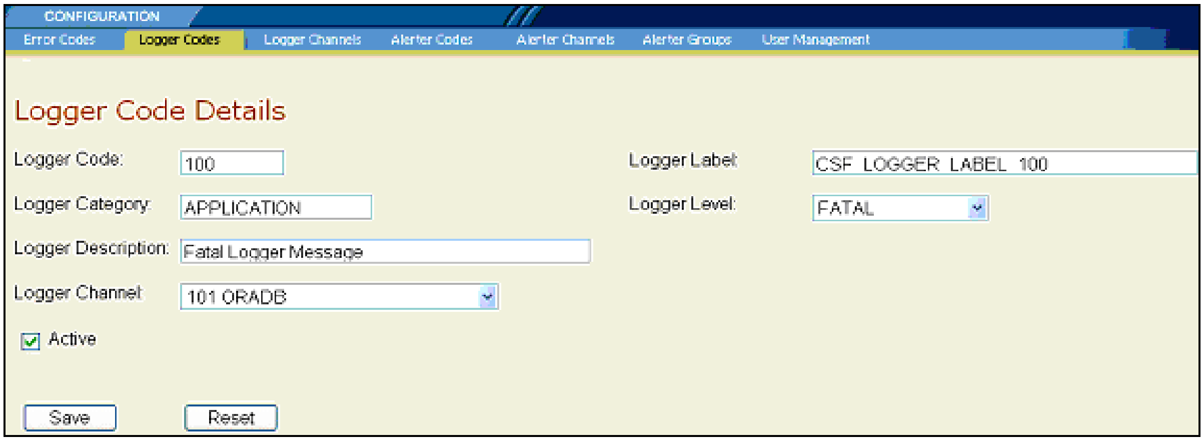
Figure 11 Logger Codes Summary Screen



Logger Codes

Logger Code	Logger Label	Logger Category	Logger Level	Logger Description	Logger Channel Code
100	CSF_LOGGER_LABEL_100	APPLICATION	FATAL	Fatal Logger Message	101
200	CSF_LOGGER_LABEL_200	APPLICATION	CRITICAL	Critical Logger Message	101
300	CSF_LOGGER_LABEL_300	APPLICATION	WARNING	Warning Logger Message	101
400	CSF_LOGGER_LABEL_400	APPLICATION	NORMAL	Normal Logger Message	101
500	CSF_LOGGER_LABEL_500	APPLICATION	TRACE	Trace Logger Message	101

Figure 12 Logger Code Details



Logger Code Details

Logger Code: 100

Logger Label: CSF LOGGER LABEL 100

Logger Category: APPLICATION

Logger Level: FATAL

Logger Description: Fatal Logger Message

Logger Channel: 101 ORADB

☑ Active

Logger Channels Management

Logger Channel Management provides ability to view, change, activate, deactivate and create logger channels. The logger channels are used for specifying the data store used by the log messages mapped to that channel.

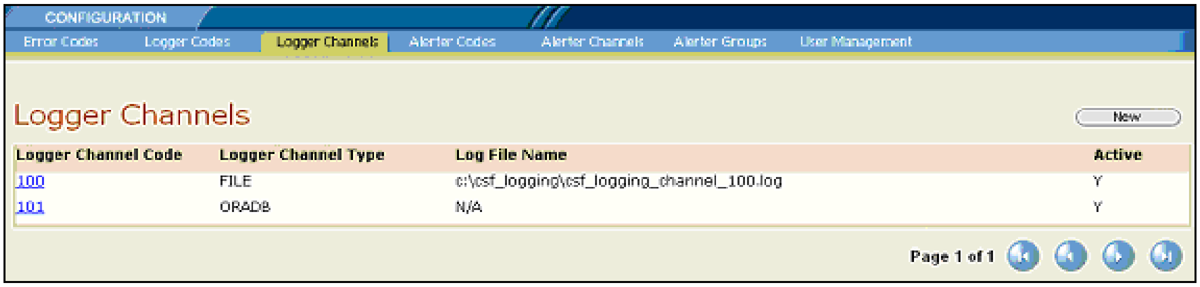
At this time logger channels support two types of data store: a database channel and a File system based channel.

1. Click the Logger Channels tab to view the list of all Logger Channels from the `CSF_LOGGER_CHANNELS` table. The screen shows summary information for each logger channel defined in the CSF. This screen is read-only.
2. Clicking the *logger channel code* link will drill down to the Logger Channel Detail screen. The detail screen displays detailed information about the selected logger channel and allows to modify any field. To save changes click Save and to revert changes to previous values click Reset. You may use the browser's *Back* button to return to the summary screen or you may click Logger Channels to display all logger channels again.
3. To define a new logger channel click the New button located on top of the Logger channels summary screen.

The logger channel detail screen provides capability to define a channel and allows specifying if it is active or deactivated.

The following two screens are sample illustrations of the logger channel view and definition screens:

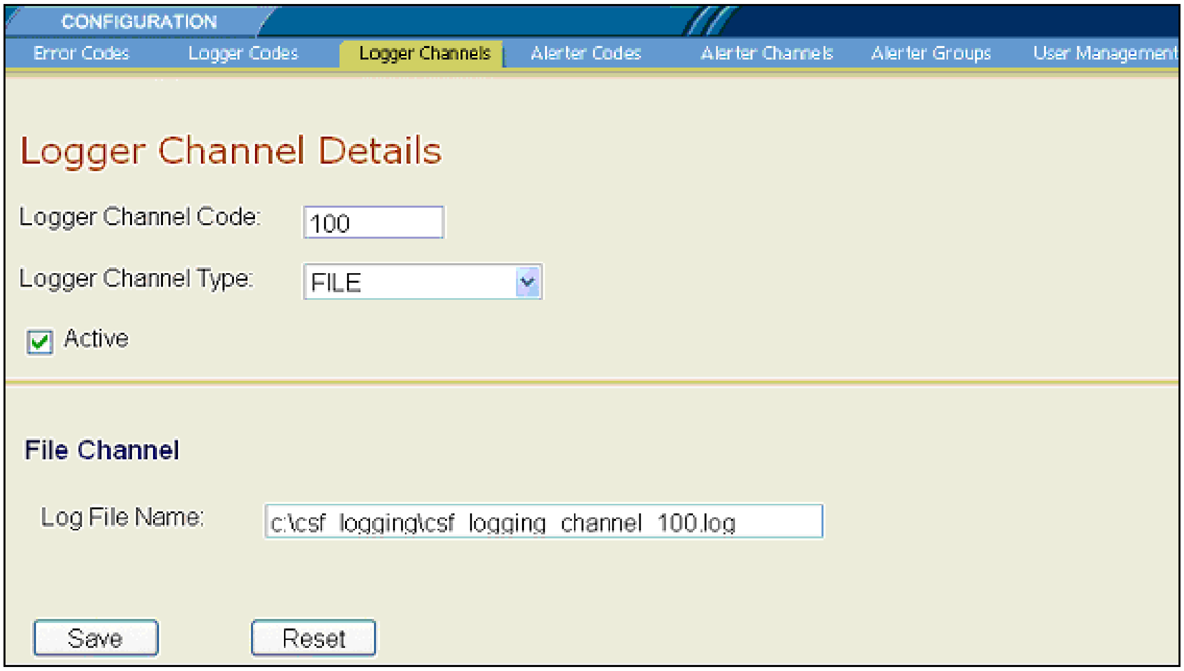
Figure 13 Logger Channels Summary Screen



Logger Channels

Logger Channel Code	Logger Channel Type	Log File Name	Active
100	FILE	c:\csf_logging\csf_logging_channel_100.log	Y
101	ORADB	N/A	Y

Figure 14 Logger Channel Details



Logger Channel Details

Logger Channel Code: 100

Logger Channel Type: FILE

☑ Active

File Channel

Log File Name: c:\csf_logging\csf_logging_channel_100.log

Alerter Codes Management

Alerter Code Management provides ability to view, change, activate, deactivate and create alerts. The alerts are used and referenced by errors as well. Alerts are stored in the alerted log. Alerts are delivered according to the specified alerter channel definition determined through the mapping for the alert at runtime. The alerts are delivered to the group defined in *CSF_ALERTER_GROUP* table that is mapped at the runtime.

The CSF runtime keeps track of the alerts and their resolutions.

1 Click the Alerter Codes tab to view the list of all alerts from the *CSF_ALERTER_CODES* table. The screen shows summary information for each alerter code defined in the CSF. This screen is read-only.

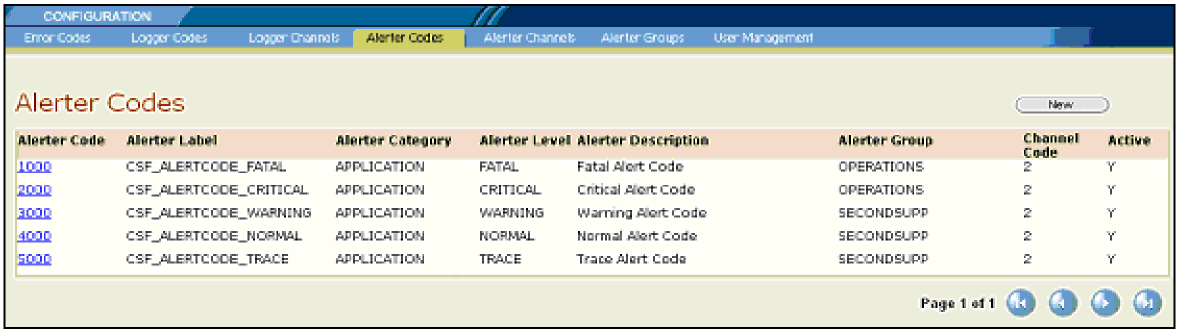
2 Clicking the Alerter Code link will drill down to the Alerter Code Detail screen. The detail screen displays detailed information about the selected alert definition and allows to modify any field. To save changes click *Save* and to revert changes to previous values click Reset. You may use the browser's *Back* button to return to the summary screen or you may click the Alerter Codes to display all alerts again. The list of values in Alerter Channel dropdown list is based on values defined for the Alerter Channels. Similarly the list of Alerter Groups is based on the values defined for Alerter Groups. Thus it is suggested to define the channels and groups before the alerts, otherwise you will need to update the mapping at a later time. It may be easier to decide on and define a group that will be changed later, than change all alerts at a later time.

3 To define a new alert code click the New button located on top of the Alerter Codes summary screen.

The alerter detail screen provides capability to specify all required information for the given alert code. Alerter code definition may be active or deactivated.

The following two screens are sample illustrations of the alerter code view and definition screens:

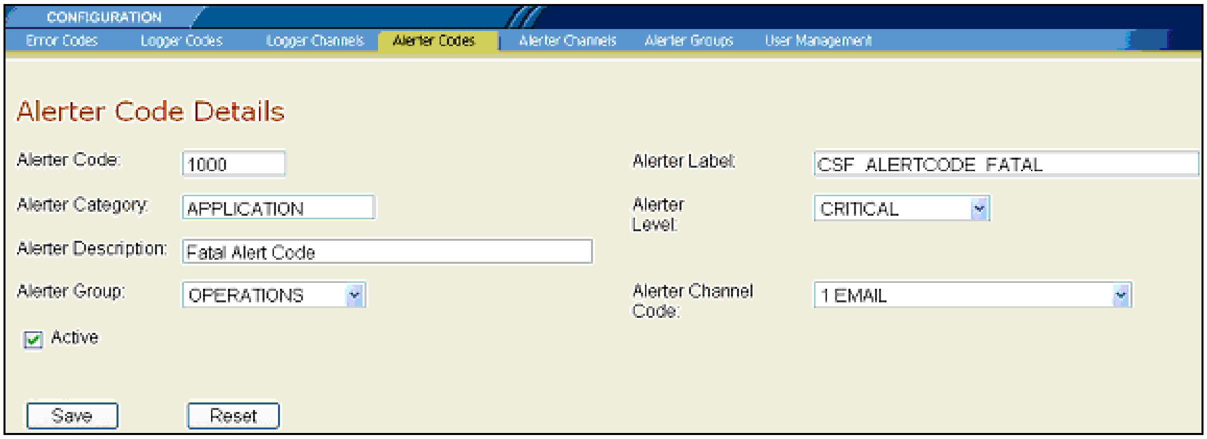
Figure 15 Alerter Codes Summary Screen



Alerter Codes

Alerter Code	Alerter Label	Alerter Category	Alerter Level	Alerter Description	Alerter Group	Channel Code	Active
1000	CSF_ALERTCODE_FATAL	APPLICATION	FATAL	Fatal Alert Code	OPERATIONS	2	Y
2000	CSF_ALERTCODE_CRITICAL	APPLICATION	CRITICAL	Critical Alert Code	OPERATIONS	2	Y
3000	CSF_ALERTCODE_WARNING	APPLICATION	WARNING	Warning Alert Code	SECONDSUPP	2	Y
4000	CSF_ALERTCODE_NORMAL	APPLICATION	NORMAL	Normal Alert Code	SECONDSUPP	2	Y
5000	CSF_ALERTCODE_TRACE	APPLICATION	TRACE	Trace Alert Code	SECONDSUPP	2	Y

Figure 16 Alerter Code Details



Alerter Code Details

Alerter Code: 1000
Alerter Label: CSF_ALERTCODE_FATAL
Alerter Category: APPLICATION
Alerter Level: CRITICAL
Alerter Description: Fatal Alert Code
Alerter Group: OPERATIONS
Alerter Channel Code: 1 EMAIL
☑ Active

Alerter Channels Management

Alerter Channel Management provides ability to view, change, activate, deactivate and create alerter channels. The alerter channels are used for specifying the delivery mechanism used for sending out alerts. The channel is mapped to the alert at runtime.

At this time alerter channel supports two types of channels: Email and SNMP. Email channel is used together with the alerter group in order to resolve where to deliver the alert.

1. Click the Alerter Channels tab to view the list of all Alerter Channels from the *CSF_ALERTER_CHANNELS* table. The screen shows summary information for each Alerter Channel defined in the CSF. This screen is read-only.
2. Clicking on alerter channel code link will drill down to the Alerter Channel Detail screen. The detail screen displays detailed information about the selected alerter channel and allows to modify any field. To save changes click *Save* and to revert changes to previous values

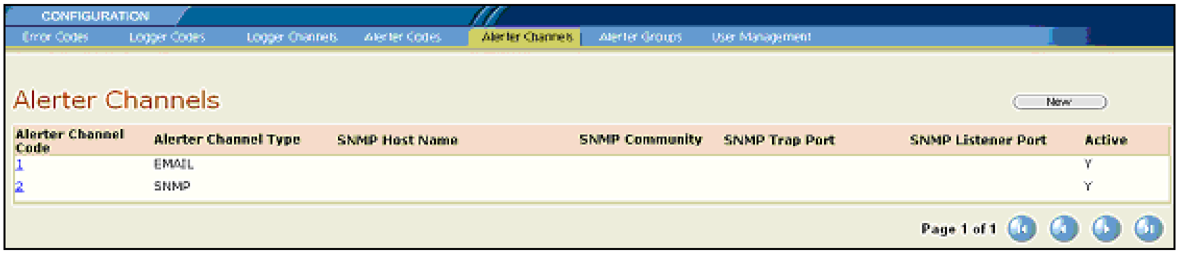
click Reset. You may use the browser’s *Back* button to return to the summary screen or you may click Alerter Channels to display all Alerter Channels again.

3 To define a new logger channel click the New button located on top of the Alerter Channels summary screen.

The Alerter Channel detail screen provides capability to define a channel and allows specifying if it is active or deactivated.

The following two screens are sample illustrations of the alerter channel view and definition screens:

Figure 17 Alerter Channels Summary Screen

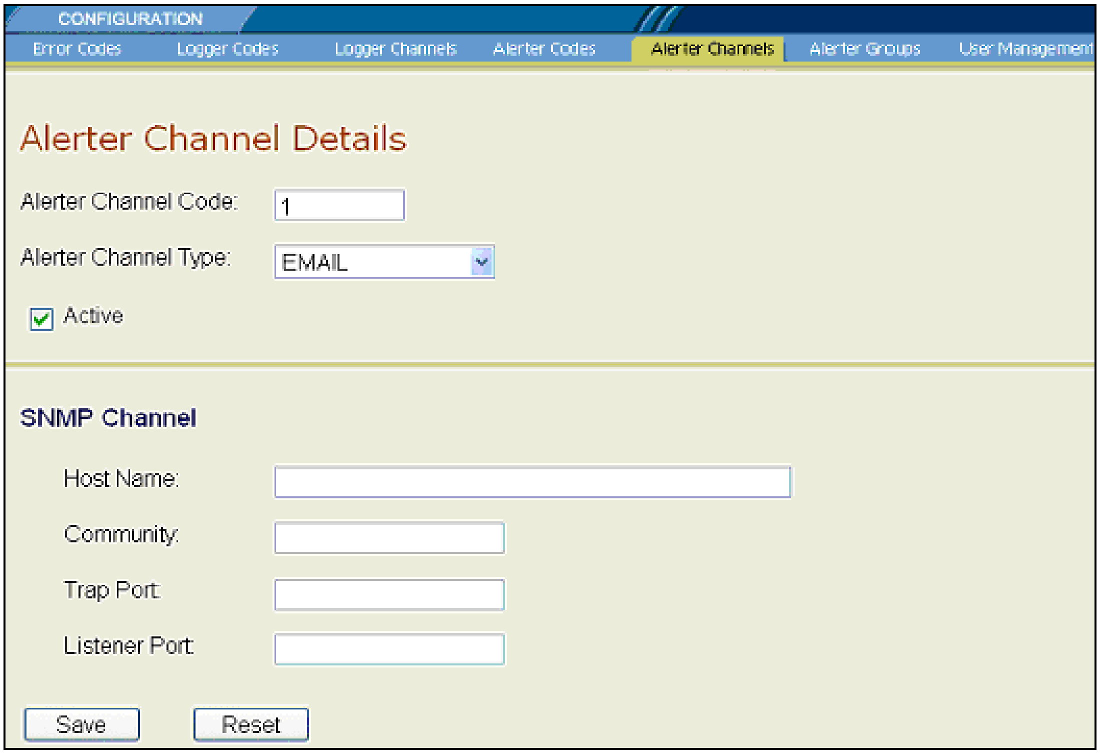


Alerter Channels

Alerter Channel Code	Alerter Channel Type	SNMP Host Name	SNMP Community	SNMP Trap Port	SNMP Listener Port	Active
1	EMAIL					Y
2	SNMP					Y

Page 1 of 1

Figure 18 Alerter Channel Details



Alerter Channel Details

Alerter Channel Code: 1

Alerter Channel Type: EMAIL

☑ Active

SNMP Channel

Host Name:

Community:

Trap Port:

Listener Port:

Alerter Groups Management

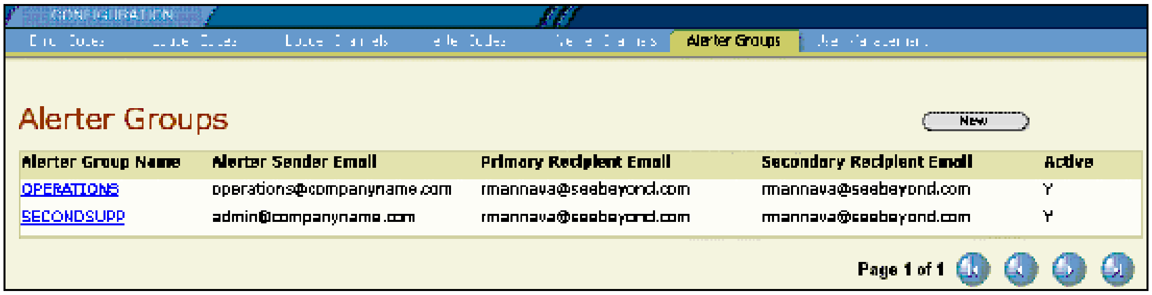
Alerter Group Management provides ability to view, change, activate, deactivate and create alerter groups. The alerter groups are used for specifying the logical name for primary and secondary email recipient. It is used at runtime to map the proper recipient with an alert.

1. Click the Alerter Groups tab to view the list of all Alerter Groups from the `CSF_ALERTER_GROUPS` table. The screen shows summary information for each alerter group defined in the CSF. This screen is read-only.
2. Clicking on alerter group name link will drill down to the Alerter Group Detail screen. The detail screen displays detailed information about the selected alerter group and allows to modify any field. To save changes click *Save* and to revert changes to previous values click Reset. You may use the browser's *Back* button to return to the summary screen or you may click Alerter Groups to display all Alerter groups again.
3. To define a new Alerter Group, click the New button located on top of the Alerter Groups summary screen.

The Alerter Group detail screen provides capability to define a group and allows specifying if it is active or deactivated.

Figure 19 illustrates of the Alerter Group view screen.

Figure 19 Alerter Groups Summary Screen



Alerter Groups

Alerter Group Name	Alerter Sender Email	Primary Recipient Email	Secondary Recipient Email	Active
OPERATIONS	operations@companyname.com	rmannava@seebeyond.com	rmannava@seebeyond.com	Y
SECONDSUPP	admin@companyname.com	rmannava@seebeyond.com	rmannava@seebeyond.com	Y

User Management

User Management provides ability to view, change, activate, deactivate and create CSF users.

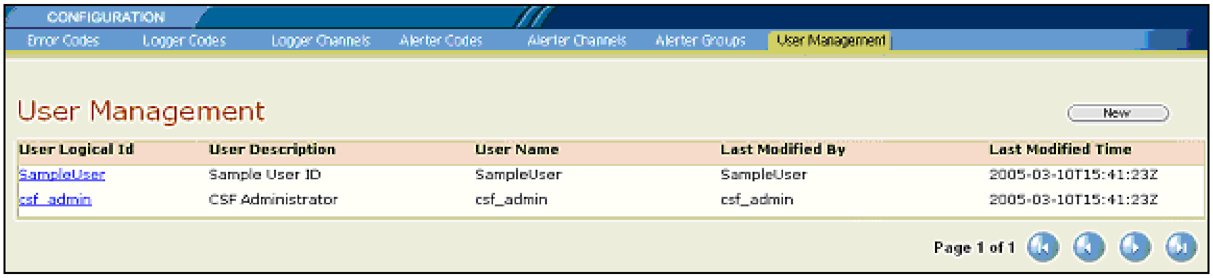
1. Click the User Management tab to view the list of all users from the `CSF_REP_USERS` table. The screen shows summary information for each user defined in the CSF. Use this screen to view and all users that are allowed to log into CSF GUI. The list of users should match user list defined in CAPS Suite Environment.

Note: *In order to define user list in the environment, start the Enterprise Designer, switch to the environment explore view, right click on the deployment environment, and select User Management. Use the User Management screen to add new users, assign passwords and roles. Each user must be assigned the* `csf_admin` *role to be able to use CSF Configuration Management screens. Please see the CSF Installation Guide for complete details.*

2 Clicking on *user logical id* link will drill down to the User Management screen. The detail screen displays detailed information about the selected user and allows to modify any field. To save changes click *Save* and to revert changes to previous values click *Reset*. You may use the browser's *Back* button to return to the summary screen or you may click User Management to display all users again.

3 To define a new user click the New button located on top of the User Management summary screen.

Figure 20 User Management Summary Screen

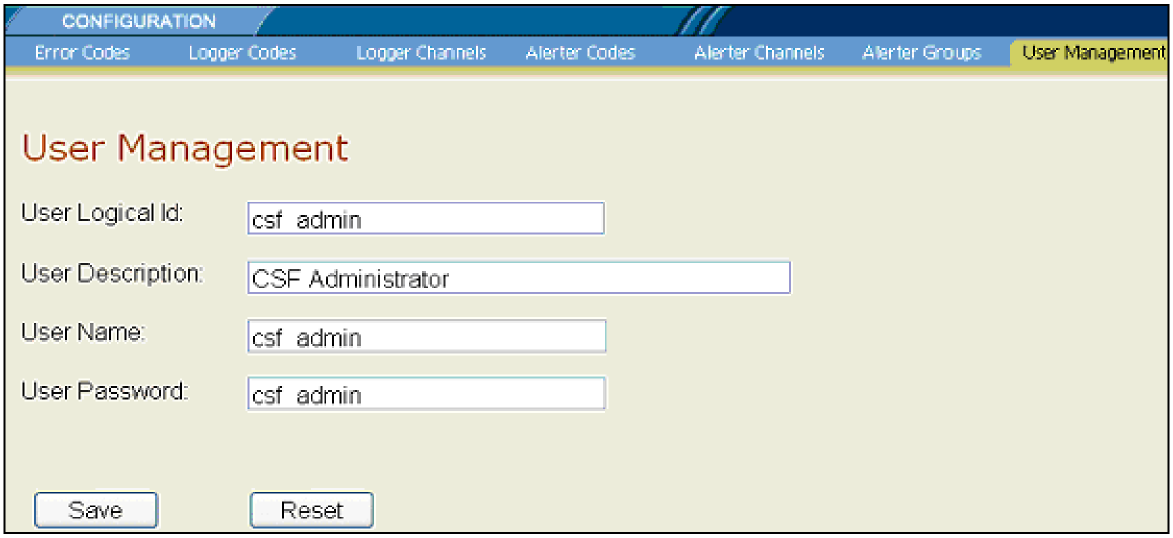


User Management

User Logical Id	User Description	User Name	Last Modified By	Last Modified Time
SampleUser	Sample User ID	SampleUser	SampleUser	2005-03-10T15:41:23Z
csf_admin	CSF Administrator	csf_admin	csf_admin	2005-03-10T15:41:23Z

Page 1 of 1

Figure 21 User Management Screen



User Management

User Logical Id: csf admin

User Description: CSF Administrator

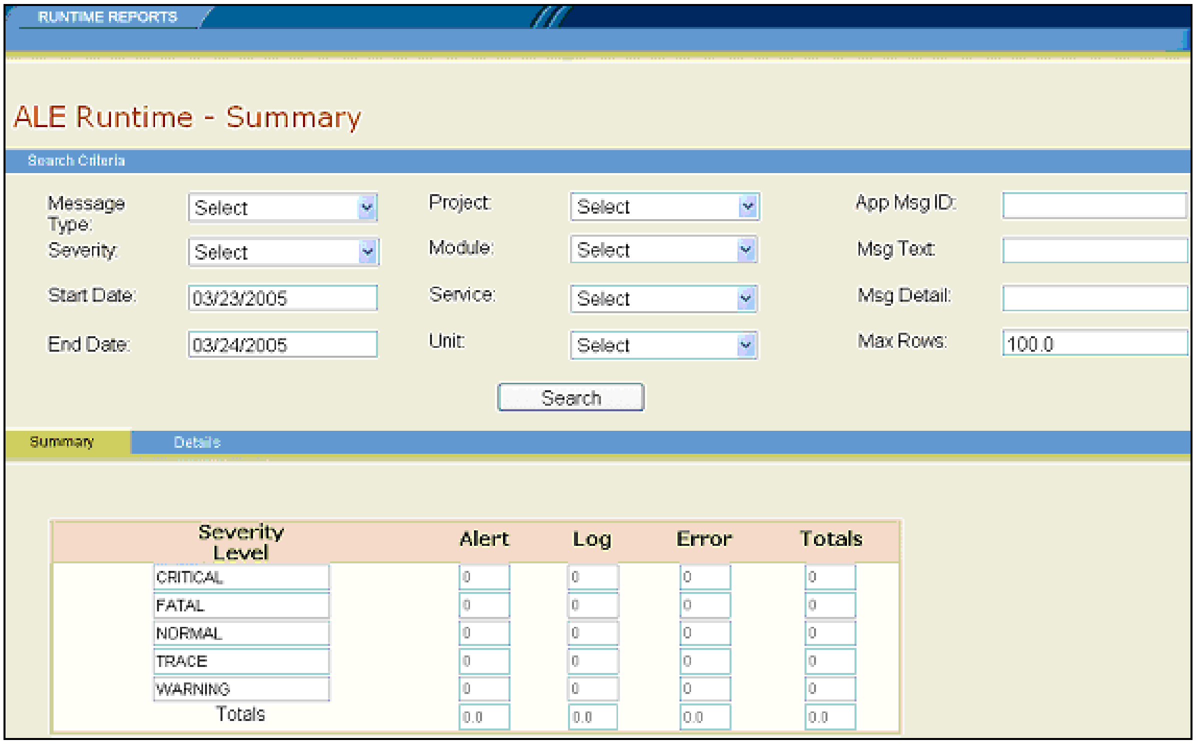
User Name: csf admin

User Password: csf admin

ALE Runtime GUI

ALE Runtime GUI offers a view into the Runtime Repository. It provides ability to define the selection criteria, view the message detail, and statistics for the runtime. When users navigate to the ALE Runtime page, the following view is presented to them:

Figure 22 ALE Runtime Home Page



ALE Runtime - Summary

Search Criteria

Message Type:	Select	Project:	Select	App Msg ID:	
Severity:	Select	Module:	Select	Msg Text:	
Start Date:	03/23/2005	Service:	Select	Msg Detail:	
End Date:	03/24/2005	Unit:	Select	Max Rows:	100.0

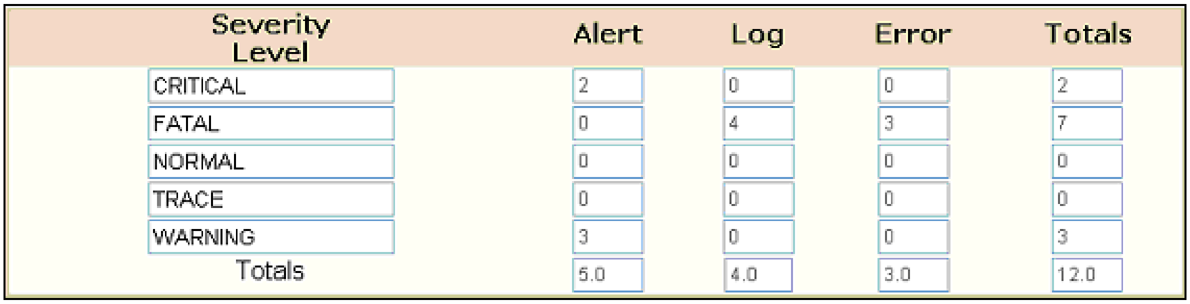
Severity Level	Alert	Log	Error	Totals
CRITICAL	0	0	0	0
FATAL	0	0	0	0
NORMAL	0	0	0	0
TRACE	0	0	0	0
WARNING	0	0	0	0
Totals	0.0	0.0	0.0	0.0

To start the GUI click the Runtime Reports tab to navigate to Runtime screens. The screens provide access to the following areas:

To use the Runtime Management GUI:

1. Select some search criteria (by default to all events of the last day) and click Search.
2. The summary section at the bottom of the screen updates to display message counts that meet your search criteria. The counts are broken down by severity and event type.

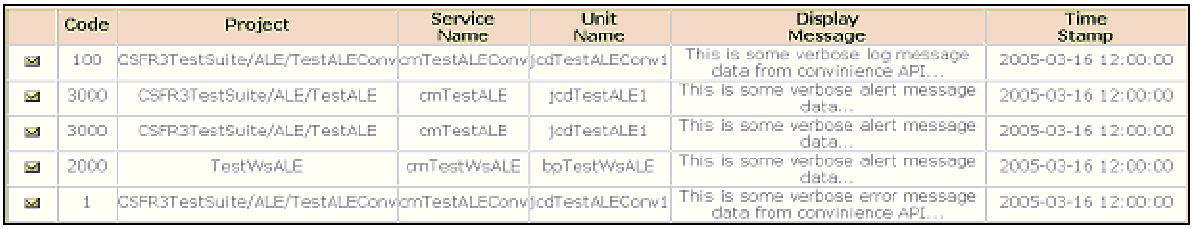
Figure 23 Message Search Results – Summary Tab



Severity Level	Alert	Log	Error	Totals
CRITICAL	2	0	0	2
FATAL	0	4	3	7
NORMAL	0	0	0	0
TRACE	0	0	0	0
WARNING	3	0	0	3
Totals	5.0	4.0	3.0	12.0

3. Click on details tab to get the list of messages that meet you criteria. The messages are sorted and displayed in the reverse chronological order by CME timestamp with newest entries at the top of the screen.

Figure 24 Message Search Results



	Code	Project	Service Name	Unit Name	Display Message	Time Stamp
✉	100	CSFR3TestSuite/ALE/TestALEConv	cmTestALEConv	jcdTestALEConv1	This is some verbose log message data from convinience API...	2005-03-16 12:00:00
✉	3000	CSFR3TestSuite/ALE/TestALE	cmTestALE	jcdTestALE1	This is some verbose alert message data...	2005-03-16 12:00:00
✉	3000	CSFR3TestSuite/ALE/TestALE	cmTestALE	jcdTestALE1	This is some verbose alert message data...	2005-03-16 12:00:00
✉	2000	TestWsALE	cmTestWsALE	bpTestWsALE	This is some verbose alert message data...	2005-03-16 12:00:00
✉	1	CSFR3TestSuite/ALE/TestALEConv	cmTestALEConv	jcdTestALEConv1	This is some verbose error message data from convinience API...	2005-03-16 12:00:00

4 Use pagination controls to scroll through the results.

5 Click the Message Details icon to open a new window displaying detailed data from the Repository for the message.

Figure 25 Pagination Controls

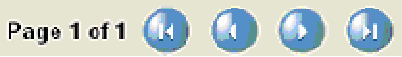
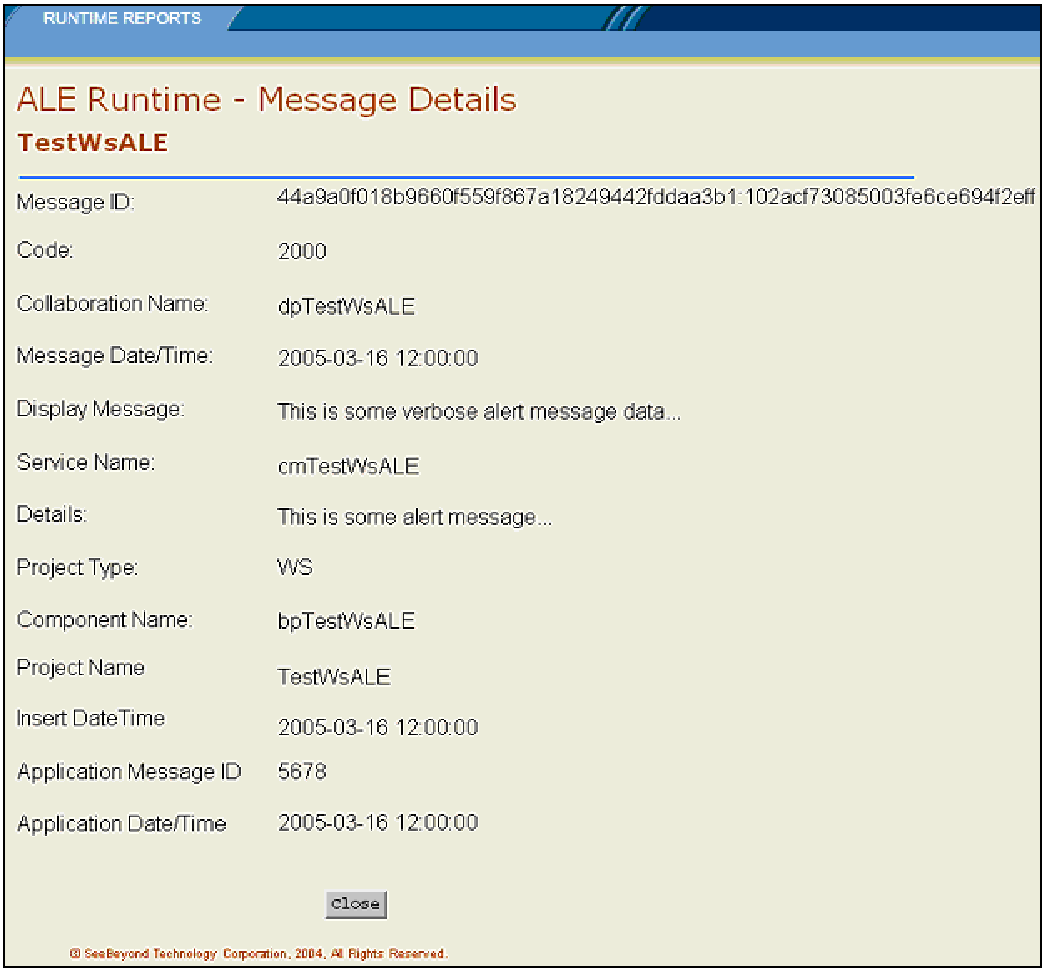


Figure 26 ALE Message Details



ALE Runtime - Message Details

TestWsALE

Message ID:	44a9a0f018b9660f559f867a18249442fddaa3b1:102acf73085003fe6ce694f2eff
Code:	2000
Collaboration Name:	dpTestWsALE
Message Date/Time:	2005-03-16 12:00:00
Display Message:	This is some verbose alert message data...
Service Name:	cmTestWsALE
Details:	This is some alert message...
Project Type:	WS
Component Name:	bpTestWsALE
Project Name	TestWsALE
Insert DateTime	2005-03-16 12:00:00
Application Message ID	5678
Application Date/Time	2005-03-16 12:00:00

© SeeBeyond Technology Corporation, 2004, All Rights Reserved.

6.1.5 CSF Alert Resolution

The CSF Alerting Service offers both logging of alert messages generated at runtime and updating of alert resolution data to the CSF Runtime Repository. Documenting the resolution of an alert starts with a Work List Manager process with assignments made according to the alerter group configured in the *CSF_ALERTER_GROUPS* table. Once a task is checked out, the user may enter data such as resolution details, resolution status, and resolved by to be tracked in the Runtime Repository. This section walks through the steps in using the Alert Resolution service.

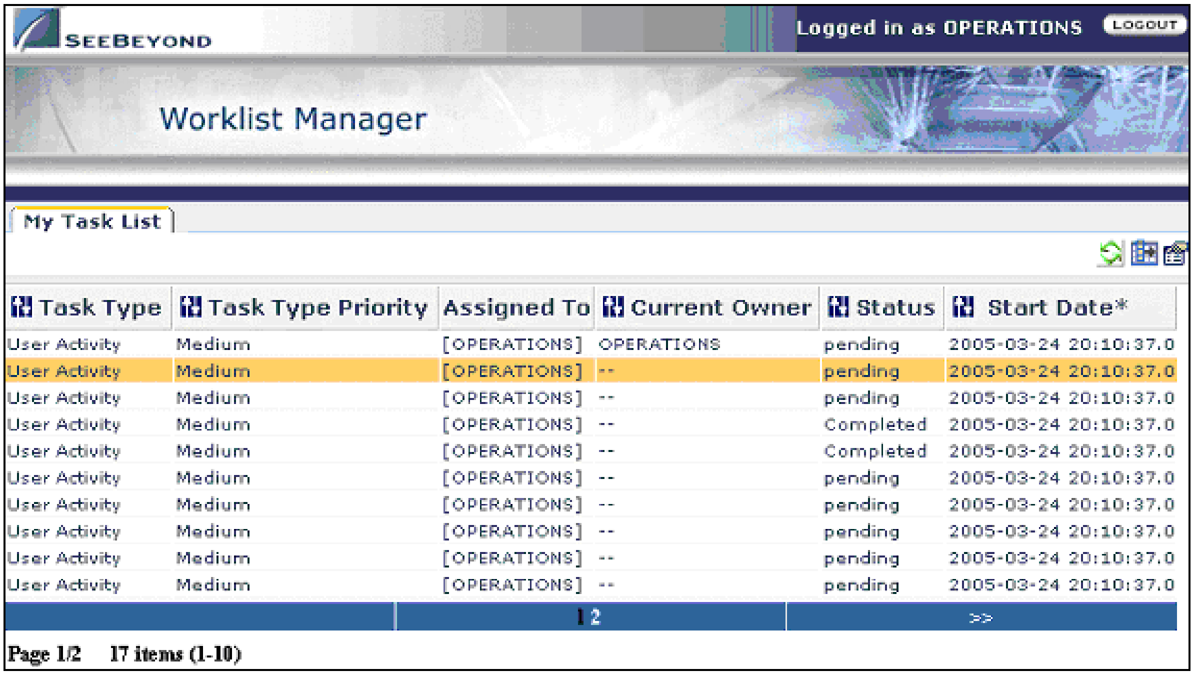
Note: *The Alert Resolution service has several application and configuration requirements. Make sure that both* `SunCAPSCSFCore` *and* `SunCAPSCSFGUI` *projects are properly installed. Be sure to review the sections on installation for CSF and installation for Alert Resolution in the CAPS CSF Installation Guide for complete details.*

Alert Resolution Process

The Alert Resolution process consists of the following high level steps:

1. LDAP and the Runtime Repository must already be started.
2. Send an Alert to ALE service using one of the CSF interfaces.
3. Start Work List Manager at the entry URL (e.g. **http://localhost:18007/wlm**);.
4. Log in as the desired user (see the following figure).

Figure 27 Start Worklist Manager with Pending Task

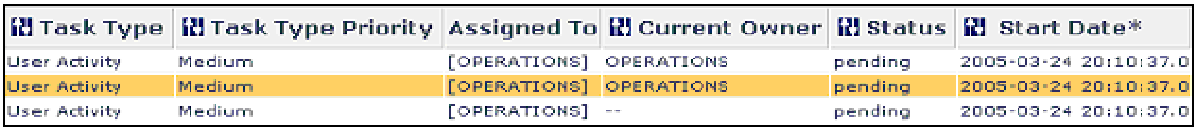


Task Type	Task Type Priority	Assigned To	Current Owner	Status	Start Date*
User Activity	Medium	[OPERATIONS]	OPERATIONS	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	Completed	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	Completed	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-24 20:10:37.0

Page 1/2 17 items (1-10)

5. Checkout the appropriate task by clicking Checkout (see the figure below).

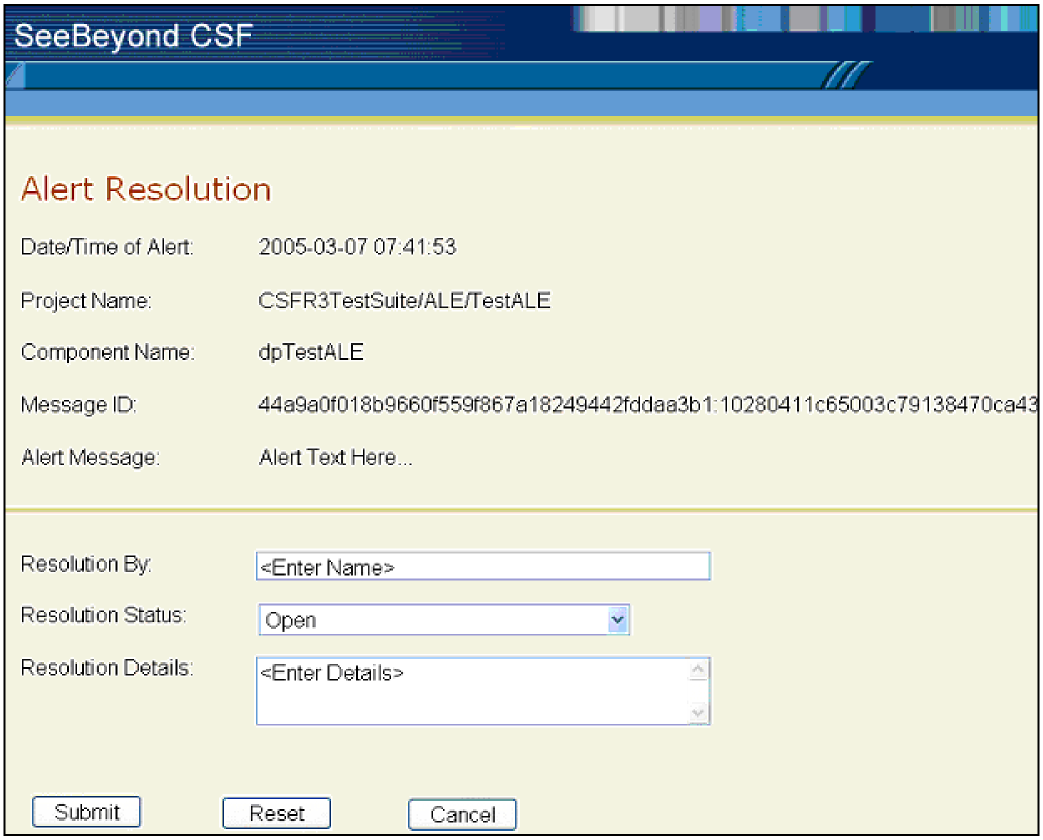
Figure 28 Checkout Task



Task Type	Task Type Priority	Assigned To	Current Owner	Status	Start Date*
User Activity	Medium	[OPERATIONS]	OPERATIONS	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	OPERATIONS	pending	2005-03-24 20:10:37.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-24 20:10:37.0

6 Execute the task by clicking Execute. This opens an Alert Resolution window (see the Figure below).

Figure 29 Alert Resolution Data



Alert Resolution

Date/Time of Alert: 2005-03-07 07:41:53

Project Name: CSFR3TestSuite/ALE/TestALE

Component Name: dpTestALE

Message ID: 44a9a0f018b9660f559f867a18249442fddaa3b1:10280411c65003c79138470ca43

Alert Message: Alert Text Here...

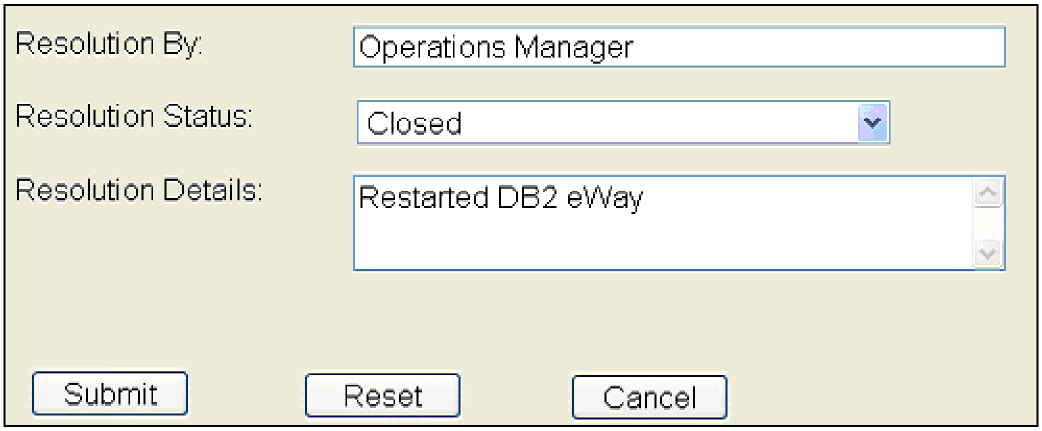
Resolution By: <Enter Name>

Resolution Status: Open

Resolution Details: <Enter Details>

7 Enter the resolution data (see the Figure below) and click the Submit button.

Figure 30 Entering Resolution Data



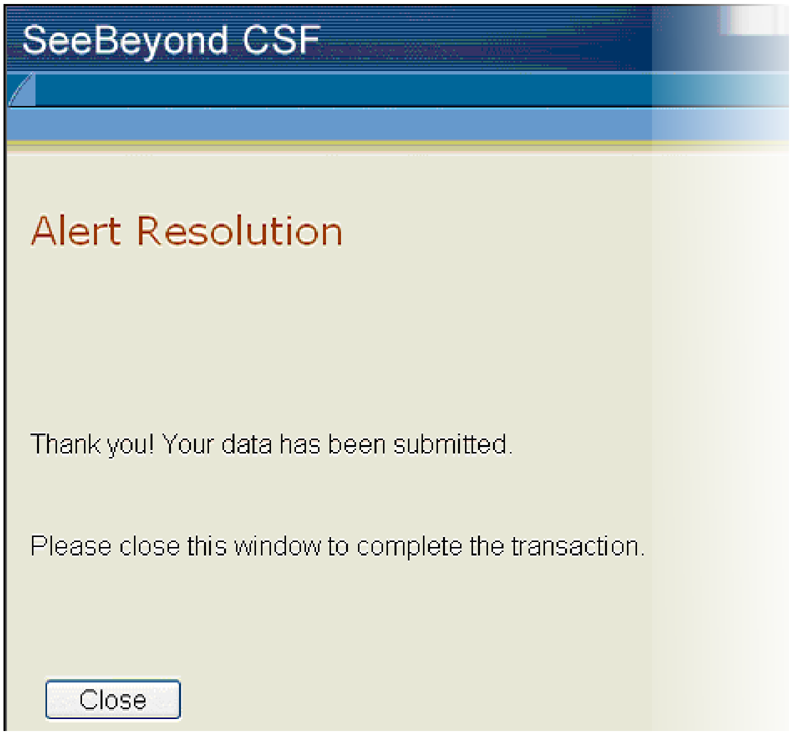
Resolution By: Operations Manager

Resolution Status: Closed

Resolution Details: Restarted DB2 eWay

The confirmation screen is displayed (see the Figure below). Review the status and click Close.

Figure 31 Close the Alert Resolution GUI



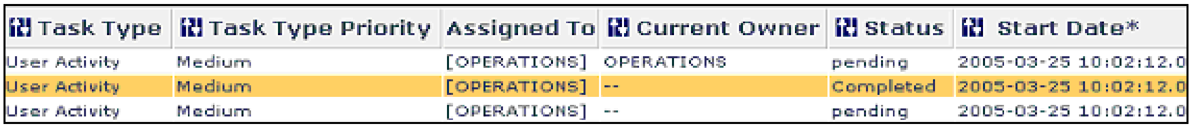
Alert Resolution

Thank you! Your data has been submitted.

Please close this window to complete the transaction.

8 Click the Complete button to complete the task (see the Figure below).

Figure 32 Complete the Task



Task Type	Task Type Priority	Assigned To	Current Owner	Status	Start Date*
User Activity	Medium	[OPERATIONS]	OPERATIONS	pending	2005-03-25 10:02:12.0
User Activity	Medium	[OPERATIONS]	--	Completed	2005-03-25 10:02:12.0
User Activity	Medium	[OPERATIONS]	--	pending	2005-03-25 10:02:12.0

9 The alert is resolved and the Runtime Repository is updated with the resolution information.

6.1.6 Java API Code Samples

This section provides some ALE client code samples that can be used as examples of the correct usage of the service.

ALE via Geneopric Java API from JCD Sample

```
public class jcdTestALE
{
  private com.stc.csf.client.api.CSFClient csf = null;
  public void receive( com.stc.connector.appconn.file.FileTextMessage
input, com.stc.connector.appconn.file.FileApplicationFileClient_1,
xsd.ALERequest_Request1172318575.RequestALERq ) throwsThrowable
  {
    try {
      csf =
com.stc.csf.client.api.CSFClientFactory.getInstance().getClient(this,
com.stc.csf.CSFConstants.CSF_CAPS_JCD);
      ALERq.unmarshalFromString(input.getText());
      java.util.Map serviceParameters = new java.util.HashMap();
      serviceParameters.put("AlertUseFlag", "Y");
      serviceParameters.put("AlertCode", ALERq.getCode());
      serviceParameters.put("AlertDetails", ìSome Alert Detailsî);
      serviceParameters.put("AlertDisplayMessage", ìA Display
Messageî);
      csf.initialize( "7890", "TestALE" );
      com.stc.csf.cme.CSFRequest csfreq = csf.prepareRequest("ALE",
serviceParameters, com.stc.csf.CSFConstants.CSF_SERVICE_PRIORITY);
      String oMsg =
ALERq.getPayload().getX_sequence_C().getOriginalMessage();
      if ( oMsg != null ) {
        csfreq.setEncodeFlag(com.stc.csf.CSFConstants.CSF_USE_NO);
        csfreq.setPersistFlag(com.stc.csf.CSFConstants.CSF_USE_NO);

csfreq.setPayloadType(com.stc.csf.CSFConstants.CSF_PAYLOAD_ORIGINAL);
        csfreq.setOriginalMessage(oMsg);
      }
      // Override default ALE behavior for Alerts
      csfreq.setRequiresXA( false );
      String mesgID = csf.doSend( csfreq );
     FileClient_1.setText("Successfully executed ALE Service. Message
ID: "+mesgID);
      FileClient_1.write();
    } catch ( Exception e ) {
      logger.error("Failed to execute ALE Service -  Error : " +
e.getMessage());
      FileClient_1.setText("Failed to execute ALE Service -  Error:
"+e.getMessage());
      FileClient_1.write();
    }
  }
    public com.stc.codegen.logger.Logger logger;
    public com.stc.codegen.alerter.Alerter alerter;
    public com.stc.codegen.util.CollaborationContext collabContext;
    public com.stc.codegen.util.TypeConverter typeConverter;
}
```

ALE via Convenience Java API from JCD Sample

```
public class jcdTestALEConv
{
  private com.stc.csf.client.api.CSFClient csf = null;

  public void
receive(com.stc.connector.appconn.file.FileTextMessageinput,
xsd.ALERequest_Request1172318575.RequestALERq,
com.stc.connector.appconn.file.FileApplicationFileClient_1)
throwsThrowable
  {
    csf =
com.stc.csf.client.api.CSFClientFactory.getInstance().getClient(this,
com.stc.csf.CSFConstants.CSF_CAPS_JCD);
    csf.initialize( "12345", "TestALEConv" ); // App Message ID & Name
    com.stc.csf.client.convenience.CSFALEConv aleConv =
newcom.stc.csf.client.convenience.CSFALEConv(csf);
    ALERq.unmarshalFromString(input.getText());
    String mesgID = aleConv.sendAlertWithDetail(ALERq.getCode(),
ALERq.getDetails(), ALERq.getDisplayMessage());
    if (mesgID != null )
     FileClient_1.setText("Successfully executed ALE Service Message
ID: "+mesgID);
    else {
      logger.error("Failed to execute ALE Service.");
      FileClient_1.setText("Failed to execute ALE Service.");
    }
    FileClient_1.write();
  }
  public com.stc.codegen.logger.Logger logger;
  public com.stc.codegen.alerter.Alerter alerter;
  public com.stc.codegen.util.CollaborationContext collabContext;
  public com.stc.codegen.util.TypeConverter typeConverter;
}
```

6.2 User Actions Audit Service

The User Actions Audit Service tracks actions of specific users as they interact with an application. It records the actions of specific users, identified by their unique `userId`, as well as the results of those actions.

Once a user has attempted an auditable action from an application, it will call the CSF User Actions Audit Service, passing `userId`, action type and action result. Additionally, the service will rely on common CME fields, including *`SourceInfo`* that will identify client component and optionally user transaction, and `Payload`, that can be used to capture the entire user request. Messages from the same business transaction can be correlated via a common business transaction id. Once the information is captured, it can be used for a number of purposes, including:

1. User Audit.
2. Non-repudiation.
3. Statistical analysis.

6.2.1 Java Convenience API

UAA convenience API is implemented by the class *CSFUserActionsAuditConv* from package *com.stc.csf.client.convenience*. A typical UAA usage scenario involves the following steps:

- Create an instance of the API object.
- Initialize it with an instance of *CSFClient.*
- Call the *sendUserAction()* method when a recordable action has been completed in the application.

6.2.2 Web Service Convenience API

Web Service User Actions Audit Convenience API specified in *UserActionsAuditService.wsdl* contains port type *UserActionsAuditServicePort*, which consists of a single operation *sendUserActionsAudit*. This operation takes a two-part message containing a *UserActionsAuditRequest* with three elements: *userID*, *actionType*, and *actionResult*, defined in *UserActionsAudit.xsd*. Similarly to the generic web Services interface, upon successful completion, the operation returns *CSFResponse* message that contains assigned Message ID and timestamp. If an error occurs, *executeService* returns a fault, containing description of the problem.

6.2.3 User Actions Audit Service Repository

User Actions Audit Service Repository schema consists of one table (see the table below). The *CSF_USER_AUDIT_LOG* table is where the user actions are recorded at runtime.

Table 23: CSF User Audit Log

Column Name	Data Type	Description
MESG_ID	*VARCHAR(128)*	The unique message ID assigned by CSF to this CSF message. Foreign Key to CSF_CME_LOG.
USER_ID	*VARCHAR2(128)*	Application-specific ID of the user who performed or attempted the action.
ACTION_TYPE	*VARCHAR2(512)*	Application-specific description of the action performed.
ACTION_RESULT	*VARCHAR2(128)*	Application-specific result of the action performed.

6.2.4 User Actions Audit GUI

There is no GUI interface for the UAA Service.

6.2.5 Java API Code Samples

This section provides some User Actions Audit client code samples that can be used as examples of the correct usage of the service.

User Actions Audit Convenience API from JCD

```
public class jcdTestUserActionsAuditConv
{
  private com.stc.csf.client.api.CSFClient csf = null;

  public void receive(com.stc.connector.appconn.file.FileTextMessage input,
       xsd.UserActionsAudit_UserActionsAudit_1205865378.UserActionsAudit
       uaa, com.stc.connector.appconn.file.FileApplication FileClient_1)
       throwsThrowable
   {
    csf =
          com.stc.csf.client.api.CSFClientFactory.getInstance().getClient(t
          his, com.stc.csf.CSFConstants.CSF_CAPS_JCD);
    csf.initialize("34789", "TestUserActionsAuditConv");
    uaa.unmarshalFromString(input.getText());
   com.stc.csf.client.convenience.CSFUserActionsAuditConv actionConv = new
          com.stc.csf.client.convenience.CSFUserActionsAuditConv(csf);
    actionConv.sendUserAction(uaa.getUserID(), uaa.getActionType(),
          uaa.getActionResult());
  }
  public com.stc.codegen.logger.Logger logger;
  public com.stc.codegen.alerter.Alerter alerter;
  public com.stc.codegen.util.CollaborationContext collabContext;
  public com.stc.codegen.util.TypeConverter typeConverter;
        }
```

6.3 Reconciliation Service

The reconciliation service is intended to provide an easy way for an application to keep message counts. It can be used in two modes: for a batch or a persistent (online) application. The difference is how the counts are collated: a batch application collates counts by ApplicationName, UnitName, and MessageID from SourceInfo. A persistent application collates counts just by ApplicationName and UnitName, but only if the sendTime is within a specified interval. This allows CSF users to perform a health check of each batch application and an availability check and performance check of the persistent applications.

Reconciliation service maintains six different counters:

- inCount: the number of messages received by the application;
- copyCount: the number of messages copied by the application;
- createCount: the number of messages created by the application;
- outCount: the number of messages sent out by the application;
- filterCount: the number of messages dropped by the application;
- errorCount: the number of messages dropped with an error by the application;

In theory, the following constraint should always hold true:

inCount + copyCount + createCount ? outCount + filterCount + errorCount

Applications accumulate the counters locally as they process the messages and then send the counter increments for the interval to CSF in a single reconciliation message. The reconciliation service then processes the counts based on the parameters in the CSF_RECONCILIATION_APPS table, which determines the application type and reconciliation interval for persistent applications. The service performs lookup in the table based on the value of ApplicationName. If no configuration entry is found, the reconciliation service will operate in persistent mode with an interval of 1 day.

6.3.1 Java Convenience API

Reconciliation convenience API is implemented by the class CSFReconciliationConv from package com.stc.csf.client.convenience. It maintains the counter values and initialization timestamp. A typical reconciliation usage scenario is shown in Figure 34 below. The user creates an instance of the API object, initializing it with their instance of CSFClient, calls appropriate reconcile<XX>() methods while processing the messages and sendReconciliation() at the end of batch or session.

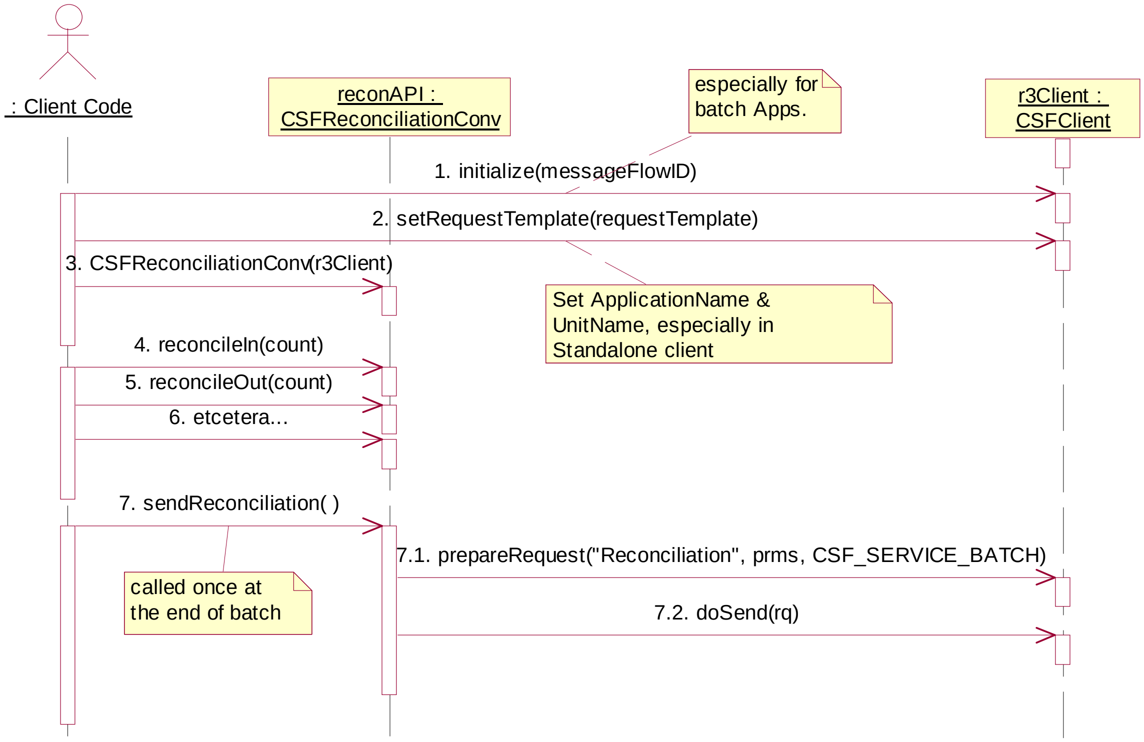

: Client Code
reconAPI : CSFReconciliationConv
especially for batch Apps.
r3Client : CSFClient
1. initialize(messageFlowID)
2. setRequestTemplate(requestTemplate)
3. CSFReconciliationConv(r3Client)
Set ApplicationName & UnitName, especially in Standalone client
4. reconcileIn(count)
5. reconcileOut(count)
6. etcetera...
7. sendReconciliation( )
7.1. prepareRequest("Reconciliation", prms, CSF_SERVICE_BATCH)
called once at the end of batch
7.2. doSend(rq)

Figure 33 Reconciliation Usage Scenario

CSFReconciliationConv is a stateful API. When used within a JCD it automatically follows the collaboration lifecycle, so the user does not have to worry about resetting the state. When invoked from a standalone client, it’s up to the user to reset the counters by calling clearReconcile() after each call to sendReconciliation() method. Another way to reset the state is to discard the API object and construct a new instance; however this would also reset the initialization timestamp. The proper method depends on the semantics of the standalone application being monitored.

6.3.2 Web Service Convenience API

Web Service Reconciliation Convenience API specified in ReconciliationService.wsdl contains port type ReconciliationServicePort, which consists of a single operation sendReconciliation. This operation takes a two-part message containing a SourceInfo structure and a ReconciliationRequest with six counter values and both timestamps defined in Reconciliation.xsd. Similarly to the generic web Services interface, upon successful completion, the operation returns CSFResponse message that contains assigned Message ID and timestamp. If an error occurs, executeService returns a fault, containing description of the problem.

6.3.3 Reconciliation Service Repository

Reconciliation Service Repository schema consists of two tables (see tables below). CSF_RECONCILIATION_APPS table contains information for configuring the reconciliation service. When a reconciliation message is received, the application name in the message is looked up in this table in order to determine whether to treat the message as a batch interface or an online interface message.

Column Name	Data Type	Description
`APP_NAME`	`VARCHAR(128)`	Name of application from `SourceInfo.SourceAppName` (defaults to the Connectivity Map name in ICAN).
`PERSISTENT`	`CHAR(1)`	Y or N flag to indicate batch or online (persistent) interface.
`INTERVAL`	`INTEGER`	Only used in persistent (on-line) mode: the interval during which record counts will be summed (in minutes)
Audit trail fields		

CSF_RECONCILIATION_LOG table contains the summed record counts for the reconciliation service.

Column Name	Data Type	Description
APP_NAME	VARCHAR(128)	Name of application which generated the count. From SourceInfo.SourceAppName.
APP_MESG_ID	VARCHAR(128)	For batch applications: the batch ID from SourceInfo.MessageID. For persistent applications: the insert time of the first record in minutes.
UNIT_NAME	VARCHAR(128)	Name of the unit (e.g. collaboration) that generated the count. From SourceInfo.UnitName.
PROJECT_NAME	VARCHAR(128)	Name of the project of the application. From SourceInfo.ProjectName.
MODULE_NAME	VARCHAR(128)	Name of the application module. From SourceInfo.ModuleName.
APP_TYPE	VARCHAR(128)	Type of the application. From SourceInfo.ApplicationType.
IN_COUNT	INTEGER	The number of records/messages received by the application.
COPY_COUNT	INTEGER	The number of records/messages copied by the application.
CREATE_COUNT	INTEGER	The number of records/messages created by the application.
OUT_COUNT	INTEGER	The number of records/messages sent by the application.
FILTER_COUNT	INTEGER	The number of records/messages dropped by the application.
ERROR_COUNT	INTEGER	The number of records/messages dropped with an error by the application.
MESG_DATESTAMP	DATE	The timestamp of the message that caused the last update of the record.
APP_DATESTAMP	DATE	The timestamp of the message that caused the initial insert of the record.
INSERT_DATE_TIME	DATE	The time of the last update of the record.

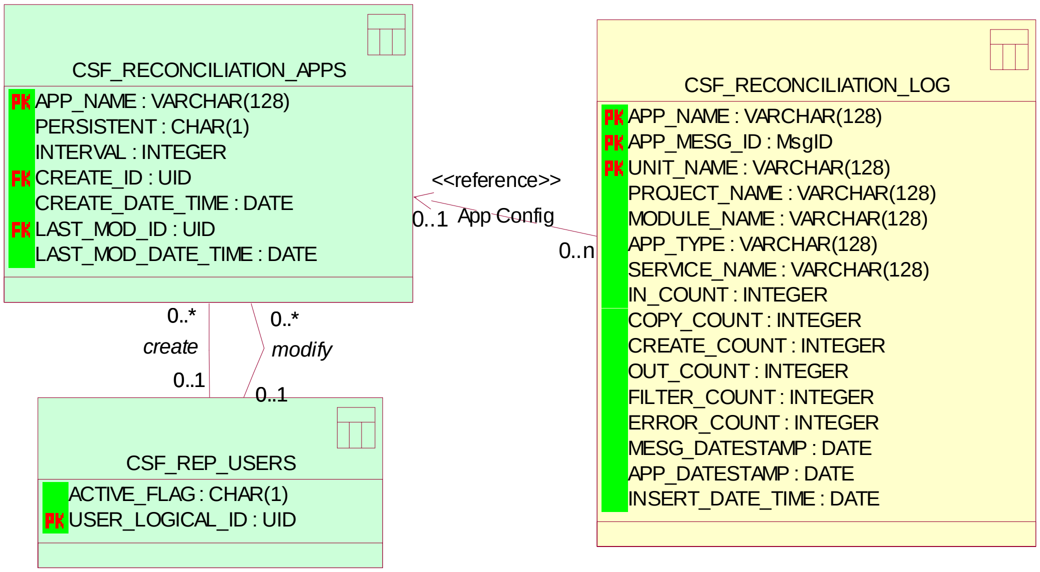


Figure 34 Reconciliation Service Repository schema

6.3.4 Reconciliation GUI

The Reconciliation GUI is a Web based application built in eVision that provides a user friendly interface to view CSF Repository runtime data generated by the Reconciliation service. Through this interface, users can get a quick summary of which applications have outstanding transactions that have not been reconciled.

Pre-requisites

Installation of the SeeBeyond ICAN Suite including eVision and eInsight is required to deploy the Configuration Management GUI.

Installation of ePortal, if portal functionality is desired

Complete installation and configuration of CSF per the CSF Installation Guide, including a JDBC compliant Repository database

Starting The GUI

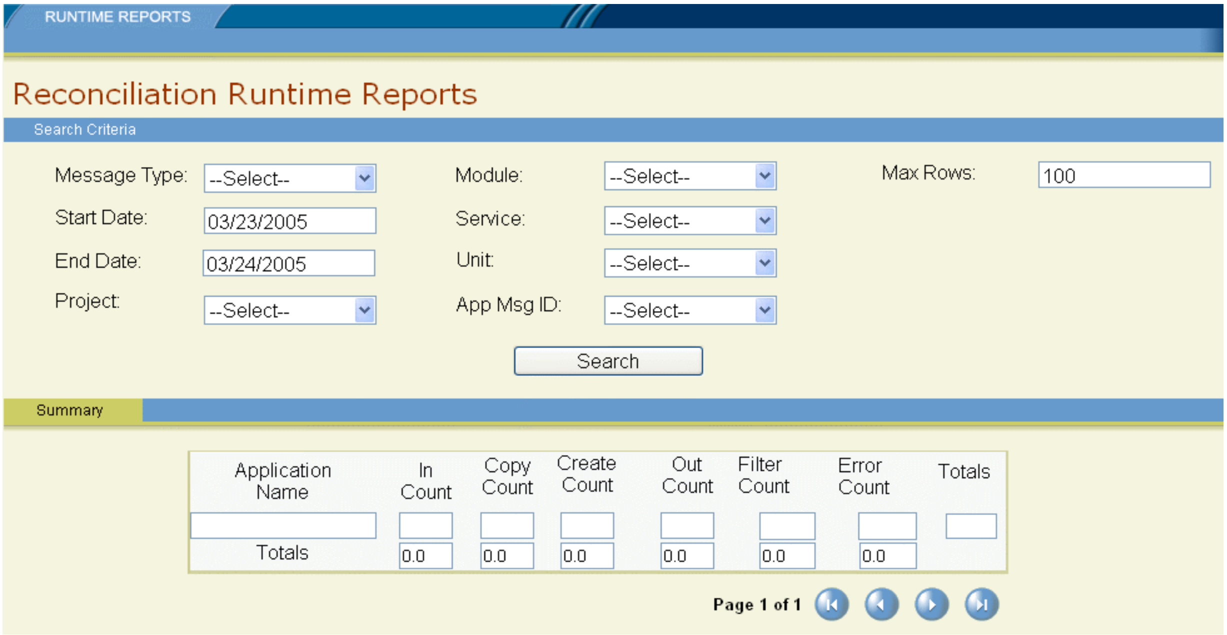
To access the Reconciliation GUI, first activate and deploy the Runtime GUI ICAN project (please see the ICAN CSF Installation Guide for complete instructions on deploying CSF projects). Once the project has been applied to a logical host, the GUI should be accessible through the following URL:

http://MyHost:19001/CSFReconRuntime

MyHost and the port number will be based on your CSF deployment. MyHost should be substituted with the name of the host machine on which your logical host domain is running. The port number is usually the domain base port + 1. Simply type the URL into an internet browser and hit Enter. During project activation a dialog box will pop up with the correct URL with host and port already filled in. You may cut and paste that URL directly into an internet browser.

GUI Navigation

The initial screen of the Reconciliation GUI (see Figure 36 below) presents options for querying Reconciliation messages in the Repository. The default selection will show a summary of all reconciliation messages for the current day. Queries can be further narrowed by selecting other options and clicking Search. The options presented are populated dynamically from the Reconciliation tables, so only those options which have available data will show in the drop down menus.

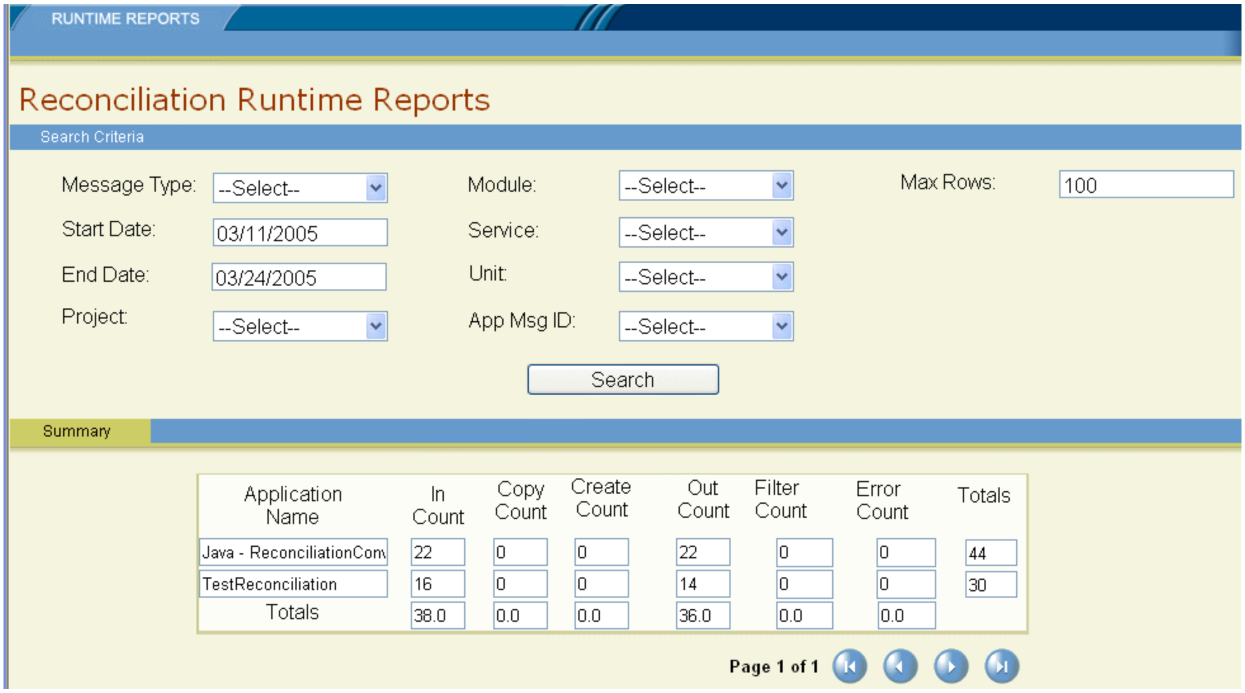


Message Type: --Select--
Start Date: 03/23/2005
End Date: 03/24/2005
Project: --Select--
Module: --Select--
Service: --Select--
Unit: --Select--
App Msg ID: --Select--
Max Rows: 100

Application Name	In Count	Copy Count	Create Count	Out Count	Filter Count	Error Count	Totals
Totals	0.0	0.0	0.0	0.0	0.0	0.0	

Figure 35 Reconciliation Query Screen

Selecting a date range, in this example 3/11/05 - 3/23/05, and clicking [Search] will display a summary of all Reconciliation counts (Figure 37 below) for those dates, for all applications.



Reconciliation Runtime Reports

Search Criteria

Message Type:	--Select--	Module:	--Select--	Max Rows:	100
Start Date:	03/11/2005	Service:	--Select--		
End Date:	03/24/2005	Unit:	--Select--		
Project:	--Select--	App Msg ID:	--Select--		

Summary

Application Name	In Count	Copy Count	Create Count	Out Count	Filter Count	Error Count	Totals
Java - ReconciliationConv	22	0	0	22	0	0	44
TestReconciliation	16	0	0	14	0	0	30
Totals	38.0	0.0	0.0	36.0	0.0	0.0	

Figure 36 Reconciliation Results Screen

In this example we see two applications which have generated Reconciliation messages. The first has matching numbers for In and Out counts. This indicates that all of the transactions reconciled (there was a response received for every request). The second application shows a discrepancy of 2, indicating that two transactions did not reconcile. We can also see a summary across all the applications shown, which tells us that 38 requests went out and 36 responses were received.

GUI Tips

7.2.4.5Reconciliation GUI Tips

Depending on the cache settings of your browser, it may be necessary to hit the browser refresh button after running a search to avoid old information being re-displayed in the current query.

Chapter 7

Operational Optimizations

CSF has been designed with a primary goal for service isolation and transparency. To achieve this we had to maintain all service-related artifacts carefully separated for each service. This resulted in increased number of libraries, deployments, and APIs then would have been required for a monolithic solution like CSF 2.0. As a result the use of the framework, particularly the operational aspects, can become complicated especially when the number of SOA services grows into the dozens. This section provides a number of recommendations how to offset these inefficiencies by consolidating service-specific assets into framework-wide ones.

Please note that there is no need for GUI consolidation, since it is already handled by ePortal.

What's in this Chapter

7.1 JAR File Consolidation

As described in section 3.1, Java clients require a number of JAR files to access CSF APIs. A typical client would need *stccf_<type>client.jar*, *stccf_properties.jar*, and a pair of *stccf_<service>_ez.jar and stccf_<service>_res.jar* for each SOA service the client intends to use. The specific combination could be different for different clients and CSF implementations, and as the number of services increase can become quite a handful. All of these libraries except the first one are generated from the CSF code, and one or more custom consolidated JARs can be generated automatically by adding corresponding targets to *Build.xml*. Please refer to *build_csf_compact* – the CSF 2.0 compatibility JAR target, as a template for such consolidated library.

Although merging existing JAR files is not generally considered safe, *cme_otd.jar* does not appear to have a manifest, and if further reduction of library count is paramount, its content can be merged with the CSF library. To do so, please extract the content of *cme_otd.jar* into to build directory, and add the extracted directories to the composite target.

The drawback to consolidating JARs is that the resulting library will combine the executable class files with mutable property files. So every time the properties have to be changed, they have to be extracted and repackaged back. In addition, for distributed deployments of CSF, where different clients require different property values, multiple variants of the libraries would have to be maintained.

7.2 Deployment Consolidation

Each service can have up to four separate Connectivity Maps, one for: service implementation, convenience Web Services API, Configuration and run-time GUIs. This could mean activating up to 50 individual Deployment Profiles to fully deploy a single CSF implementation with a dozen services. The simple solution to this problem is to create consolidated Connectivity Maps for all the services, Web Service Interfaces, and the GUIs. An example of such consolidated maps for the two services shipped with CSF 3.0 can be found in *SeeBeyondCSFCore/deplolyCsfServices and SeeBeyondCSFCore/deployCsfWebServices*.

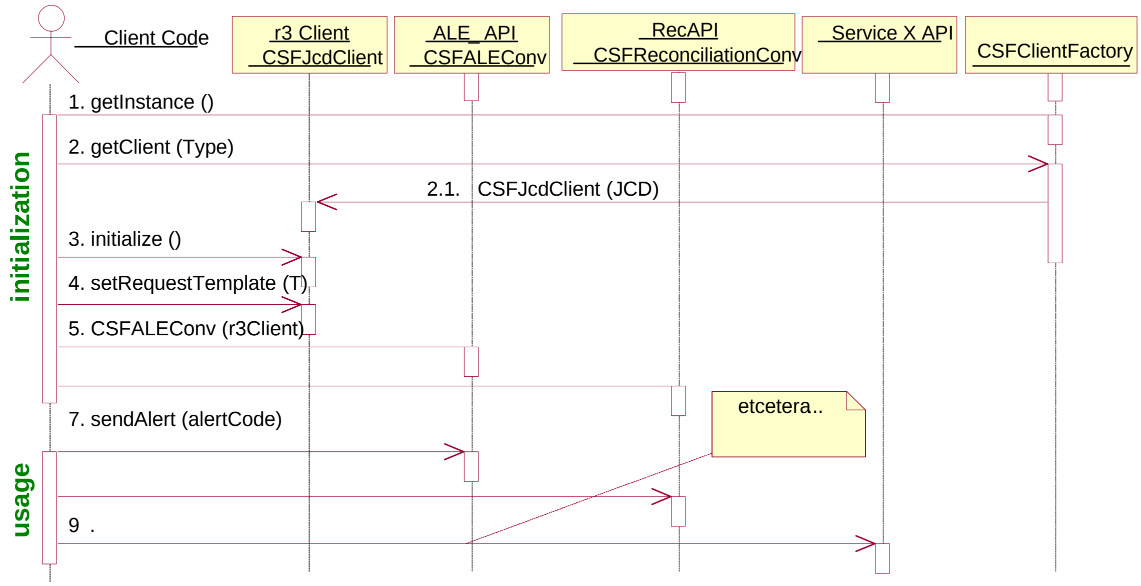
This Connectivity Map would have to be created from scratch for each CSF installation based on the combination of services, and changed every time services are added or removed, however after the initial development phase changes in the service architecture should be very infrequent and the benefit of streamlined deployment process should outweigh the extra development efforts.

The drawback to this approach is that the tradeoff for ease of deployment is the ability to control deployment of each individual service. Generally this should not be an issue in production environments, however if certain services require individual deployment, their Connectivity Maps should not be consolidated, and they should be deployed via individual maps that come with each service.

7.3 API Consolidation

A typical Java API usage scenario, that is illustrated in the figure below, contains a number of steps required for initialization of the client. Again, as the number of services increase, so will the complexity of the initialization code, since each service has its own convenience API that has to be initialized with a shared instance of generic client.

Figure 37 Typical Java API Usage

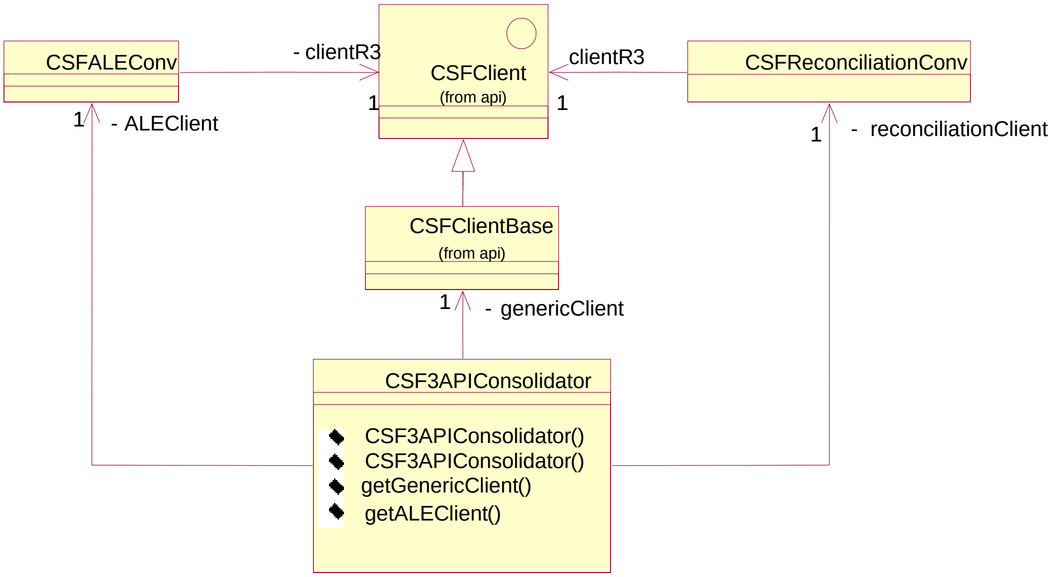


You can easily create an API consolidator class as shown in Figure 38 that would encapsulate initialization and manage the instances of service-specific convenience APIs for your configuration of CSF. The code that performs the same function as in Figure 40 would then look something like:

```
APIConsolidator csfAPI = new APIConsolidator(type, this);
csfAPI.getGenericClient().setRequestTemplate(T);
csfAPI.getALEClient().sendAlert(alertCode);
csfAPI.getReconciliationClient().reconcileIn(count);
csfAPI.getXClient().doSomething(params);
```

In this example the first line hides all initialization code, except client-specific step 4, which is performed explicitly in the next line. The remaining lines demonstrate how to call service-specific API methods without the need to keep around the instances of convenience APIs.

Figure 38 CSF API Consolidator



A sample API consolidator *com.stc.csf.client.convenience.CSF3APIConsolidator* is provided with the framework and can be customized to match your configuration of CSF.

Chapter 8

CSF Tests

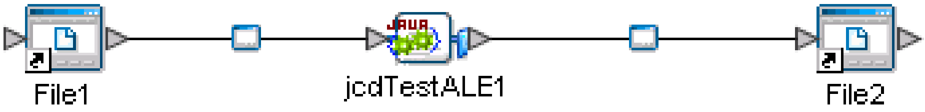
8.1 CAPS Test Suite

The *SeeBeyondCSFTestSuite* CAPS project contains several pre-built testing scenarios. CSF test suite consists of individual projects, arranged by service and interface type, that exercise the service, interface and framework functionality. Although they are not intender for regression testing, they offer a good coverage of the entire CSF deployment. These test projects allow you to verify your CSF installation quickly, as well as serve as excellent implementation examples. This section summarizes what tests are available.

8.1.1 Overview

Individual test projects are mini CAPS applications. The typical test consists of a JCD, which reads request parameters from an input file via, invokes a CSF operation and posts invocation results into an output file.

Figure 39 Typical CSFTest (Map)



Web Services tests also use eInsight business process to serve as a web services client.

ALE Test Projects

There are three CAPS test scenarios for testing the ALE service. In the Enterprise Designer, the projects are located under *SunCAPSCSFTestSuite/ALE/<project name>*

1. *TestALE:* Invokes the ALE service using the standard CAPS java client.
2. *TestALEConv:* Invokes the ALE service using the standard CAPS java client + convenience API.
3. *TestWsALE:* Invokes the ALE service via web services.

Reconciliation Test Projects

There are three CAPS test scenarios for testing the Reconciliation service. In the Enterprise Designer, the projects are located under `SunCAPSCSFTestSuite/Reconciliation/<project name>`

1. `TestReconciliation:` Invokes the Reconciliation service using the standard ICAN java client.
2. `TestReconciliationConv:` Invokes the Reconciliation service using the standard ICAN java client + convenience API.
3. `TestWsReconciliation:` Invokes the Reconciliation service via web services.

User Actions Audit Test Projects

There are three CAPS test scenarios for testing the User Actions Audit service. In the Enterprise Designer, the projects are located under `SunCAPSCSFTestSuite/UserActionsAudit/<project name>`

1. `TestUserActionsAudit:` Invokes the User Actions Audit service using the standard ICAN java client.
2. `TestUserActionsAuditConv:` Invokes the User Actions Audit service using the standard ICAN java client + convenience API.
3. `TestWsUserActionsAudit:` Invokes the User Actions Audit service via web services.

8.1.2 Exporting the Sample Data

Each test project has a sub-folder named `DataFiles`. This folder contains sample test input files that can be used to run that test project. These files will need to be exported and placed onto the same physical machine that will be running the Logical Host. To export the files:

1. In Enterprise Designer, switch to the Project Explorer view, and navigate to the desired test project. Expand the `DataFiles` sub-project folder.
2. Right-click on a data file and select *Export*.
3. Select a location to save the file to (`C:/TestCSFR3/In` is the default input directory).
4. If necessary, copy the file to the Logical Host machine.

8.1.3 Configuring Tests for your Environment

The CSF test projects use java properties files to get details on where the `SeeBeyondCSFCore` project is deployed. These properties are located within JAR files. You will need to extract the properties files form the JARs, edit them to match your environment, and then re-archive them and re-import the resulting JARs into the test projects. For detailed description of how to edit these properties files, see the *CSF Installation Guide*.

Once you have edited the files and re-built the JAR files, replace any JARs that you edited in the following projects:

- SunCAPSCSFTestSuite/ALE/JarFiles
- SunCAPSCSFTestSuite/Reconciliation/JarFiles
- SunCAPSCSFTestSuite/UserActionsAudit/JarFiles

To do this:

1. Right-click the existing JAR file and select *Import*;
2. Select the newly built JAR from the proper location and click *Import*;
3. Click the *Save All* button in the eDesigner.

8.1.4 Configuring CSF Repository for your Environment

The sample data included in the CSF Repository scripts pre-populates the repository with data that can be used for testing, and which matches the sample input files provided with the test projects. However, there are a few updates which need to be made to the repository data before tests can be run against it.

OPERATIONS Alert Group

In the *`CSF_ALERTER_GROUPS`* table there is a row for a group named Operations. This is the default group to which alert messages will be sent. Change the following fields as indicated:

- *`ALERTER_FROM:`* ensure this field has an email address in it, i.e., **CSFOperations@csfalerts.com**. The address does not have to be valid but does need to be in the *format* of an email address. This is the address tat will appear in the `From:` field in alert emails.
- *`ALERTER_TO_PRIMARY:`* provide a valid email address to which you would like to send the alert emails. You should have access to this account so that you can verify the alerts are being sent correctly.
- *`ALERTER_TO_SECONDARY:`* provide a second email address that will receive alert emails, if desired. Do not leave this field blank, though. If no secondary email is required simply fill in a fake email such as **noemail@csfalerts.com**.

FILE Log Channel

In the *`CSF_LOG_CHANNELS`* table there is a row for a channel named FILE. This is the channel that defines where log messages saved to a file will be recorded. If you plan to use the FILE log channel, change the following fields as indicated:

- FILE_NAME: The full path and file name of the file to which the log data should be written. The file location must be on the same physical machine as the Logical Host which runs CSF.

8.1.5 Running the tests

Please see the *CSF Installation Guide* for a walkthrough of how to execute the test projects. All tests can be re-run as many times as required by repeatedly placing the input files in the input directory.

8.1.6 Expected results

Each test file has an associated service that it invokes (open the input file in a text editor to see/edit its contents). When the test file is run, the *`CSF_CME_LOG`* table and the *`_LOG`* table for the associated service should be updated with a new record. In addition there will be an output file which is updated with the results of the test. For complete detail on this topic see the *CSF Installation Guide*.

Chapter 9

CME Structure

This chapter shows the CME structure and explains the various fields in the following table.

9.0.1 Component Management Entity Fields

The Component Management Entity (CME) fields are explained below.

Table 24: CME Field Description and Usage

Field	Purpose	Required	Value is Set					Value is Used				
			In Client Code	In Client API			On the Server	In Client API	CSF		Services	
				JCD	Stand Alone	WS			Used	Saved	Used	Saved
MessageHeader		y										
MessageID	Unique ID of the message given by the CSF.	n	no	yes	yes	yes	no	no	no	yes	yes	yes
DateTimeStamp	Time Stamp of Event from CSF.	n	no	yes	yes	yes	no	no	no	yes	yes	yes
Priority	Priority level for processing of the request.	n	can be	can be	can be	no	no	yes	no	no	can be	no

Table 24: CME Field Description and Usage (Continued)

Field	Purpose	Required	Value is Set					Value is Used				
			In Client Code	In Client API			On the Server	In Client API	CSF		Services	
				JCD	Stand Alone	WS			Used	Saved	Used	Saved
`SourceInfo`		y										
`ProjectName`	Business area, e.g. EDI, Procurement, Eligibility	n	can be	yes	can be	no	no	no	no	yes	can be	yes
`ApplicationType`	Type of the client application, e.g. CAPS, J2SE, .Net, SRE.	n	can be	yes	can be	yes	no	no	no	yes	can be	yes
`ApplicationName`	The name of a business application. This field will be used to reference application-specific configuration tables, such as those used by the Reconciliation service.	n	can be	yes	can be	no	no	no	no	yes	yes	yes
`ServiceName`	Represents a deployable set of functionality	y	can be	yes	can be	no	no	no	no	yes	no	yes

Table 24: CME Field Description and Usage (Continued)

Field	Purpose	Required	Value is Set					Value is Used				
			In Client Code	In Client API			On the Server	In Client API	CSF		Services	
				JCD	Stand Alone	WS			Used	Saved	Used	Saved
ModuleName	Represents a runnable unit, such as instantiation of a Service. In component-based architectures it could also be a Component.	y	can be	yes	can be	no	no	no	no	yes	no	yes
UnitName	Represents the lowest level identifiable logical component.	y	can be	yes	can be	no	no	no	no	yes	yes	yes
MessageID	Application Message (Transaction) ID. Used by CSF Services to correlate messages.	n	can be	yes	yes	yes	no	no	no	no	yes	yes
DateTimeStamp	Time stamp of application event.	n	can be	yes	yes	yes	no	no	no	no	can be	yes
ServiceRequest		y										

Table 24: CME Field Description and Usage (Continued)

Field	Purpose	Required	Value is Set					Value is Used				
			In Client Code	In Client API			On the Server	In Client API	CSF		Services	
				JCD	Stand Alone	WS			Used	Saved	Used	Saved
RequestName	Name of the service request. Determines routing to the specific service.	y	yes	no	no	no	no	yes	yes	no	yes	no
ParameterPayload	Base64 encoded XML document with service-specific parameters.	n	yes	can be	can be	can be	no	no	no	no	yes	yes
ServiceLevel	Priority or Batch. Intended for service bus but will not currently be used.	y	can be	yes	yes	yes	no	yes	yes	no	can be	no
ServiceUserFields		n										
Field	Contains request-specific, untyped, or repeating data.	n	yes	no	no	no	no	no	no	no	yes	yes
Name	Name of the parameter.	y	yes	no	no	no	no	no	no	no	yes	yes
Value	Parameter value.	y	yes	no	no	no	no	no	no	no	yes	yes
Payload		n										

Table 24: CME Field Description and Usage (Continued)

Field	Purpose	Required	Value is Set					Value is Used				
			In Client Code	In Client API			On the Server	In Client API	CSF		Services	
				JCD	Stand Alone	WS			Used	Saved	Used	Saved
PersistFlag	Determines whether payload(s) should be persisted or discarded after processing.	y	yes	yes	yes	yes	no	no	yes	no	no	no
PersistMode	Defines where the payloads should be persisted.	n	yes	yes	yes	yes	no	yes	yes	no	no	no
EncodeFlag	Determines whether payload(s) are encoded.	y	yes	yes	yes	yes	yes	yes	yes	no	no	no
EncodeMode	Defines the payload encoding method.	n	yes	yes	yes	yes	yes	yes	yes	yes	no	no
PayloadType	Defines payload are included: Original, Transformed or both.	y	yes	yes	yes	yes	no	yes	yes	yes	no	no
OriginalMessage	Original payload.	n	yes	no	no	no	no	no	no	yes	no	no

Table 24: CME Field Description and Usage (Continued)

Field	Purpose	Required	Value is Set					Value is Used				
			In Client Code	In Client API			On the Server	In Client API	CSF		Services	
				JCD	Stand Alone	WS			Used	Saved	Used	Saved
TransformedMessage	Processed payload.	n	yes	no	no	no	no	no	no	yes	no	no
HeaderUserFields	Used for CSF customization.	n										
Field	Contain generic untyped or repeating data.	n	yes	can be	can be	can be	no	no	can be	yes	no	no
Name	Name of the parameter.	y										
Value	Parameter value.	y										
VersionInfo		y										
CMEVersion	Version of the CME message.	y	no	yes	yes	yes	no	no	yes	no	no	no
ClientVersion	Version of the client that sent the request.	n	no	yes	yes	yes	no	no	can be	no	no	no

9.0.2 SourceInfo Values

Table 25: Automatic and Explicit SourceInfo Fields Values

Field	Client Types									
	JCD Value		**SRE Value**		**BP Value**		**Non-JIS Java Value**		**Web Wervice Value**	
	Automatic	**Explicit**	**Automatic**	**Explicit**	**Automatic**	**Explicit**	**Automatic**	**Explicit**	**Automatic**	**Explicit**
`ProjectName`	always explicit	"Payroll"	always explicit	"Payroll"	always explicit	"Payroll"	always explicit	"Payroll"	always explicit	"Payroll"
`ApplicationType`	"CAPS-JCD"	"Library XYZ"	"SRE-JCD"	"Library XYZ"	"CAPS-BP"	"eVision"	"Java"	"Portal XYZ"	"WS"	".NET"
`ApplicationName`	always explicit	Connectivity Map Name[a]	always explicit	"OSS Bridge"	always explicit	"OSS Bridge"	always explicit	Web / Enterprise App name	always explicit	Web App Name
`ServiceName`	Connectivity Map Name	Use default	Schema Name	Use default	Connectivity Map Name	Use default	always explicit	Web App Name	always explicit	DLL Name
`ModuleName`	Deployment Profile Name	Use default	Collaboration Name	Use default	Deployment Profile Name	Use default	always explicit	Action or Portlet Name	always explicit	COM / DCOM Name
`UnitName`	Collaboration Name	Use default	Collaboration Rule Name	Use default	BP Name	Use default	Java class Name	JSP Name	always explicit	ASP Name

[a] For compatibility with CSF 2.0.

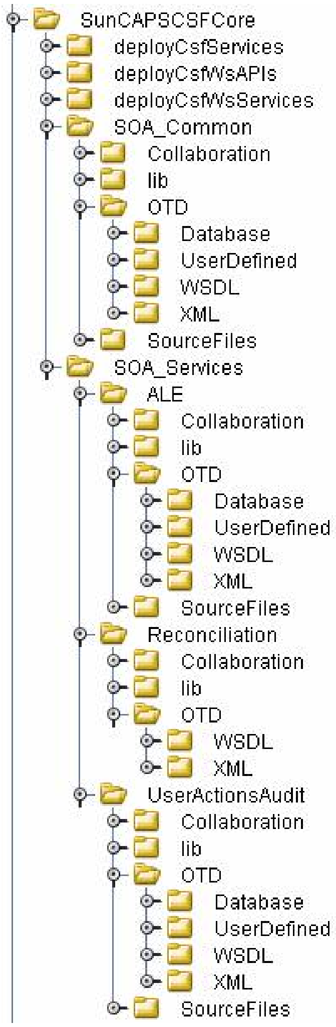
Chapter 10

Structure of CSF Packages

This chapter shows the tree structure of the folders.

10.0.1 CSF Core

Figure 40 Package Structure of CSF Core and Services



Contains consolidated Connectivity maps, Deployment profiles, topics and external systems

Contains service-independent and shared artifacts

Java and other types of collaborations, BPs

Contains Jar files and other deployable artifacts

Object Type Definitions

WSDL, XSD, DTD, DDL, Templates, ...

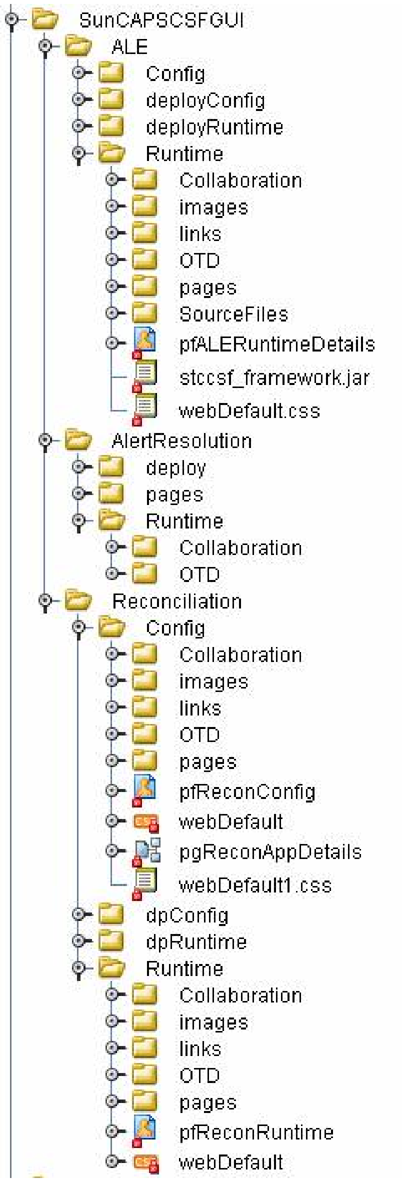
Contains individual services

Contains services' Connectivity Maps, Deployment profiles, etc.

Repeats the above structure

10.0.2 CSF GUIs

Figure 41 Package Structure of CSF GUIs



Common GUI root

Service GUI root

Service GUI's Connectivity Map and Deployment profiles

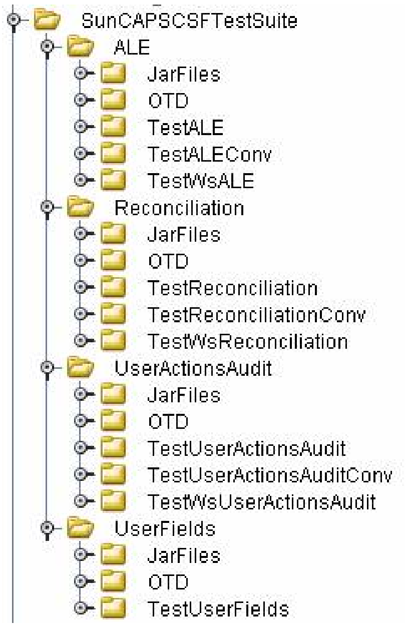
Service Configuration Portlet

Service Runtime Portlet

Repeated for each SOA Service

10.0.3 CSF Test Suite

Figure 42 Package Structure of CSF Test Suite



```
SunCAPSCSFTestSuite
    ALE
        JarFiles
        OTD
        TestALE
        TestALEConv
        TestWsALE
    Reconciliation
        JarFiles
        OTD
        TestReconciliation
        TestReconciliationConv
        TestWsReconciliation
    UserActionsAudit
        JarFiles
        OTD
        TestUserActionsAudit
        TestUserActionsAuditConv
        TestWsUserActionsAudit
    UserFields
        JarFiles
        OTD
        TestUserFields
```

Common Test Suite root

ALE Files

Reconciliation

User Actions Audit

User Fields

Chapter 11

CSF Client Configuration

This chapter provides the default content of the property files and XML templates used to configure the CSF client. Bold typeface is used for changeable values.

11.0.1 Property Files

csfPriorityService.properties

```
# The jdni entry for STCMS running under the control of RTS
JMS_XA_CONNECTION_FACTORY_JNDI_ENTRY=CsfJmsXA
JMS_NONXA_CONNECTION_FACTORY_JNDI_ENTRY=CsfJmsNoXA
TOPIC=tpcPriority
# The follwoing properies used only if security is turned on in STCMS.
JMS_CONNECTION_USER_NAME=Administrator
JMS_CONNECTION_PASSWORD=STC
```

csfBatchService.properties

```
# The jdni entry for STCMS running under the control of RTS
JMS_XA_CONNECTION_FACTORY_JNDI_ENTRY=CsfJmsXA
JMS_NONXA_CONNECTION_FACTORY_JNDI_ENTRY=CsfJmsNoXA
TOPIC=tpcBatch
# The follwoing properies used only if security is turned on in STCMS.
JMS_CONNECTION_USER_NAME=Administrator
JMS_CONNECTION_PASSWORD=STC
```

csfj2seclient.properties

```
INITIAL_CONTEXT_FACTORY=com.stc.jms.jndispi.InitialContextFactory
PROVIDER_URL=stcms://localhost:18007
# The following properies used only if security is turned on in STCMS.
# and default setting used.
#SECURITY_PRINCIPAL=Administrator
#SECURITY_CREDENTIALS=STC
```

log4j.properties

```
log4j.rootCategory=DEBUG, dest1
log4j.appender.dest1=org.apache.log4j.ConsoleAppender
log4j.appender.dest1.layout=org.apache.log4j.PatternLayout
log4j.appender.dest1.layout.ConversionPattern=%-5p: %m%n
```

csfALE.properties

```
SERVICE_LEVEL=BATCH
MSG_PRIORITY=5
REQUIRES_XA=TRUE
```

csfReconciliation.properties

```
SERVICE_LEVEL=BATCH
MSG_PRIORITY=5
REQUIRES_XA=TRUE
```

csfUserActionsAudit.properties

```
SERVICE_LEVEL=BATCH
MSG_PRIORITY=5
REQUIRES_XA=TRUE
```

11.0.2 XML Template

ALE.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ALE>
   <Alert>
    <AlertUseFlag>N</AlertUseFlag>
    <AlertCode>0</AlertCode>
    <AlertDetails> </AlertDetails>
    <AlertDisplayMessage> </AlertDisplayMessage>
   </Alert>
   <Log>
    <LogUseFlag>N</LogUseFlag>
    <LogCode>0</LogCode>
    <LogDetails> </LogDetails>
    <LogDisplayMessage> </LogDisplayMessage>
   </Log>
   <Error>
    <ErrorUseFlag>N</ErrorUseFlag>
    <ErrorCode>0</ErrorCode>
    <ErrorDetails> </ErrorDetails>
    <ErrorDisplayMessage> </ErrorDisplayMessage>
   </Error>
</ALE>
```

Reconciliation.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Reconciliation>
   <inCount>0</inCount>
   <copyCount>0</copyCount>
   <createCount>0</createCount>
   <outCount>0</outCount>
   <filterCount>0</filterCount>
   <errorCount>0</errorCount>
   <initTime>2001-12-17T09:30:47.0Z</initTime>
   <sendTime>2001-12-17T09:30:47.0Z</sendTime>
</Reconciliation>
```

UserActionsAudit.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<UserActionsAudit>
    <UserID>String</UserID>
    <ActionType>String</ActionType>
    <ActionResult>String</ActionResult>
</UserActionsAudit>
```

Chapter 12

CSF Constants

There are a number of constant values used in the CSF java code which are defined in the *com.stc.csf.CSFConstants* interface. This section describes the constants that are available and their meanings.

12.0.1 Java Code Constants Table

Table 26: Java Code Constants

Type	Constant	Value	Notes
String	*CSF_DEFAULT*	"CSF_DEFAULT"	No longer used in R3
String	*CSF_NULL*	"CSF_NULL"	Null value
String	*CSF_NO_DATA*	"CSF_NO_DATA"	No data value
String	*CSF_REPOSITORY_DEFAULT*	"CSF_REPOSITORY_DEFAULT"	Use the default value from the Repository
String	*CSF_SUCCESS*	"CSF_SUCCESS"	Service return value indicating success
String	*CSF_FAILURE*	"CSF_FAILURE"	Service return value indicating failure
String	*CSF_ERROR*	"CSF_ERROR"	Service return value indicating an error
String	*CSF_CME_VER*	"3.0.0"	Version of the CME data format
String	*CSF_CHANNEL_EMAIL*	"EMAIL"	Email channel (alerts)
String	*CSF_CHANNEL_SNMP*	"SNMP"	SNMP Channel (alerts)
String	*CSF_CHANNEL_FILE*	"FILE"	File channel (logs)
String	*CSF_CHANNEL_ORADB*	"ORADB"	Oracle Database
String	*CSF_CHANNEL_SYBDB*	"SYBDB"	Sybase Database
String	*CSF_CHANNEL_SQLSDB*	"SQLSDB"	SQL Server Database
String	*CSF_CHANNEL_DB2DB*	"DB2DB"	DB2 Database

Table 26: Java Code Constants (Continued)

Type	Constant	Value	Notes
String	*CSF_JMS_QUEUE*	"QUEUE"	JMS Queue
String	*CSF_JMS_TOPIC*	"TOPIC"	JMS Topic
String	*CSF_PERSIST_MEMORY*	"MEMORY"	Persist data in memory only
String	*CSF_PERSIST_DB*	"DATABASE"	Persist data to database
String	*CSF_PERSIST_FILE*	"FILE"	Persist data to file
String	*CSF_ENCODE_NONE*	"NOENCODING"	No encoding
String	*CSF_ENCODE_ASCII*	"ASCII/Text"	Ascii encoding
String	*CSF_ENCODE_BASE64*	"BASE64"	Base64 encoding
String	*CSF_PAYLOAD_ORIGINAL*	"ORIGINAL_MSG"	Payload is in its original form
String	*CSF_PAYLOAD_TRANSFORMED*	"TRANSFORMED_MSG"	Payload is in its transformed form
String	*CSF_PAYLOAD_ORIG_TRANS*	"BOTH_MSGS"	Payload contains both the original and transformed form
String	*CSF_CATEGORY_SYSTEM*	"SYSTEM"	Category is system
String	*CSF_CATEGORY_APPLICATION*	"APPLICATION"	Category is application
String	*CSF_RESOLUTION_NOTREAD*	"NOT READ"	Alert/Error message has not been read
String	*CSF_RESOLUTION_READ*	"READ"	Alert/Error message has been read
String	*CSF_RESOLUTION_PENDING*	"PENDING"	Resolution status is pending
String	*CSF_RESOLUTION_RESOLVED*	"RESOLVED"	Resolution status is resolved
String	*CSF_SEVERITY_INFO*	"INFO"	Info level
String	*CSF_SEVERITY_WARNING*	"WARNING"	Warning level
String	*CSF_SEVERITY_DEBUG*	"DEBUG"	Debug level
String	*CSF_SEVERITY_ERROR*	"ERROR"	Error level
String	*CSF_SEVERITY_CRITICAL*	"CRITICAL"	Critical level
String	*CSF_SEVERITY_FATAL*	"FATAL"	Fatal level

Table 26: Java Code Constants (Continued)

Type	Constant	Value	Notes
String	CSF_SEVERITY_TRACE	"TRACE"	Trace level
String	CSF_USE_YES	"Y"	Use flag is ìyesî
String	CSF_USE_NO	"N"	Use flag is ìnoî
String	CSF_CLIENT_VERSION	"3.0.0"	Version of the CSF
String	CSF_BYTE_TYPE	"BYTE"	Byte primitive class type
String	CSF_BOOLEAN_TYPE	"BOOLEAN"	Boolean primitive class type
String	CSF_CHARACTER_TYPE	"CHARACTER"	Character primitive class type
String	CSF_DOUBLE_TYPE	"DOUBLE"	Double primitive class type
String	CSF_FLOAT_TYPE	"FLOAT"	Float primitive class type
String	CSF_INTEGER_TYPE	"INTEGER"	Integer primitive class type
String	CSF_LONG_TYPE	"LONG"	Long primitive class type
String	CSF_NUMBER_TYPE	"NUMBER"	Number primitive class type
String	CSF_SHORT_TYPE	"SHORT"	Short primitive class type
String	CSF_STRING_TYPE	"STRING"	String primitive class type
String	CSF_DATE_TYPE	"DATE"	Date primitive class type
String	CSF_XMLINTERFACE_NAME	"XMLINETRFACE"	XML interface name constant
String	XML_RESOURCE_PATH	"XML/"	XML resource path
String	JAXP_SCHEMA_LANGUAGE	"http://java.sun.com/xml/jaxp/properties/schemaLanguage"	JAXP schema language path
String	W3C_XML_SCHEMA	"http://www.w3.org/2001/XMLSchema"	W3C XML schema path
String	JAXP_SCHEMA_SOURCE	"http://java.sun.com/xml/jaxp/properties/schemaSource"	JAXP schema source path

Table 26: Java Code Constants (Continued)

Type	Constant	Value	Notes
String	*CSF_SERVICE_NAME_PROPERTY*	"SERVICE_NAME"	Message selector key
String	*CSF_SERVICE_BATCH*	"BATCH"	Batch service level
String	*CSF_SERVICE_PRIORITY*	"PRIORITY"	Priority service level
String	*CSF_CAPS_JCD*	"CAPS-CSF"	CAPS application
String	*CSF_J2SE_APP*	"JAVA"	Java application
String	*CSF_CHANNEL_TOPICS*	"TOPICS"	JMS Topic channel
String	*CSF_CHANNEL_QUEUES*	"QUEUESî	JMS Queue channel

Glossary

Collaboration

A logical operation performed between some combination of message destinations and external applications. The operation is defined by a Collaboration Definition, which can be encoded in either Java or XSLT.

Also see **"Service"** and **"Collaboration Definition"**.

Collaboration Definition

The encoding of business rules, in Java or XSLT format. Typically, the encoding consists of operations on OTDs (see **"OTD"**). Several Collaborations can have the same Collaboration Definition.

Connection

Consists of the configuration information that enables an eWay to connect to an external system.

Connectivity Map

Contains business logic and routing information about the data transmission. A Connectivity Map usually includes one or more Collaborations, topics, queues, and eWays. A Connectivity Map is created under a Project. A Project may have multiple Connectivity Maps.

Deployment Profile

Contains the information about how the Project components will be deployed in an Environment. A Project can have multiple Deployment Profiles, but only one Deployment Profile can be activated for a Project in any one Environment.

Environment

A collection of physical resources and their configurations that are used to host Project components. An Environment contains logical hosts and external systems.

eWay

A link between a Collaboration and an external connection including the message server connection (topic or queue) or external application.

External Application

A logical representation in an eGate Project of an external application.

External System

A representation in an eGate Project of an external application system.

CAPS Suite

The Sun Java Composite Application Platform Suite.

Integration Server

J2EE software platform that houses the business logic container used to run Collaborations and JCA connectors (eWays). Provides transaction services, persistence, and external connectivity.

JMS IQ Manager

JMS-compliant, guaranteed delivery store, forwarding, and queueing service.

Link

The JMS Connection between a Collaboration and a topic or queue in a JMS-compliant message server.

Logical Host

An instance of the eGate runtime Environment that is installed on a machine. A Logical Host contains the software and other installed components that are required at runtime, such as application and message servers.

Management Agent

Uses J2EE technology to manage and monitor an eGate deployment that may contain other application servers in addition to the Integration Server. Defines management interfaces and services designed for distributed environments, focusing on providing functionality for managing networks, systems, and applications.

Message Destination

A general term for a topic or queue. Two or more Projects can share a message destination that has the same name and is deployed on the same message server. A single Project may also have a single message destination referenced in multiple Connectivity Maps.

OTD

An acronym for Object Type Definition. OTDs contain the data structure and rules that define an object. An OTD is used in Java Collaboration Definitions for creating data transformations and interfacing with external systems.

Project

Contains a collection of logical components, configurations, and files that are used to solve business problems. A Project organizes the files and packages and maintains the settings that comprise an eGate system within the Enterprise Designer.

Queue

A JMS queue is a shareable object that conforms to the *point-to-point* (p2p, or PTP) messaging domain, where one sender delivers a message to exactly one receiver. When the JMS IQ Manager sends a message to a queue, it ensures it is received once and only once, even though there may be many receivers “listening” to the queue. This is equivalent to the subscriber pooling in other queue implementations. You can reference a queue that exists in another Connectivity Map or Project.

Repository

Stores and manages the setup, component, and configuration information for eGate Projects. The Repository also provides monitoring services for Projects, which include version control and impact analysis.

Schema Runtime Environment

A self-contained eGate environment that allows customers to import and run their eGate 4.x projects, and allows these projects to integrate with eGate 5.x projects. Also known as the SRE.

Service

Contains the information about executing a set of business rules. These business rules can be defined in a Java Collaboration Definition, XSLT Collaboration Definition, Business Process, eTL Definition, or other service. A Service also contains binding information for connecting to JMS Topics, Queues, eWays, and other services.

Topic

A JMS topic is a shareable object that conforms to the *publish-and-subscribe* (pub/sub) messaging domain, where one publisher broadcasts messages to potentially many subscribers. When the JMS IQ Manager publishes a message on a topic, it ensures that all subscribers receive the message.

XSLT

An acronym for Extensible Stylesheet Language Transformations. A file format used in eGate to generate Collaboration Definitions.

e*Gate 4.x Terms in eGate 5.0

Table 27 provides definitions for the terms that are new with eGate release 5.0, as well as equivalent terms from eGate release 4.x.

Table 27 eGate 5.0 Terms

5.0 Term	4.x Equivalent Term
Collaboration	Collaboration
Collaboration Definition	Collaboration Definition
Connection	eWay Connection
Connectivity Map	Closest: Network View of an entire Schema
Deploy	Run the Control Broker
Deployment	<none>
Deployment Profile	Closest: Schema
Enterprise Designer	Enterprise Manager
Enterprise Manager	Enterprise Monitor
Environment	Schema (except only includes physical information, not business logic)
eWay	eWay Connection eWay
eWay Configuration	eWay Connection Configuration
External Application	eWay Connection
External System	eWay Connection
JMS Connection	eWay Connection
Integration Server	<none>
Link	JMS eWay Connection
Linked Message Destination	<none>
Logical Host	Participating Host
Message Destination	Topic or queue
Message Server	JMS IQ Manager
Object Type Definition (OTD)	Event Type Definition (ETD)
Process Manager	Control Broker
Project	Schema (except not including physical layer)
Queue	JMS queue
Repository	Registry
Subproject	Schema

Table 27 eGate 5.0 Terms (Continued)

5.0 Term	4.x Equivalent Term
Topic	JMS topic
XSLT	<none>

Index