Enterprise Transport API Java Edition 3.9.1.L1

DACSLOCK API DEVELOPERS GUIDE

Document Version: 3.9.1.L1
Date of issue: September 2025
Document ID: ETAJ391L1EDAC.250

LSEG DATA & ANALYTICS

Contents

1 Introduction

1.1 Product Description

The goal of permissioning is to control access to data by users. Using an entitlement system such as the Data Access Control System, permission profiles can be defined identifying what each user is allowed to access. The Data Access Control System is the entitlement system for LSEG Data Management Solutions (LSEG Real-Time Distribution System). Data Access Control System permission checks take place in the Market Data Client application. In order to perform a permission check for an item, the Data Access Control System must have 'requirements' information for the item and a profile for the user (a.k.a. content based permissioning).

Data Access Control System requirements information is organized as numeric expressions. Each subservice of a service, e.g. all the data from an exchange, is assigned a (series of) numeric entitlement codes (PEs). The numeric entitlement codes are transported with items as they move from source servers to Market Data Client applications on the LSEG Real-Time Distribution System. In order to accommodate the most general case in which information from multiple sources is combined to form new items (such as in compound servers), the requirements information for an item is a Boolean expression containing these entitlement codes. For an item obtained directly from a source, this expression is usually a single term. When a compound server combines two items to form a new (compound) item, it must also combine the requirements information to form a new (compound) requirement.

The name of the subservice associated with an entitlement code is maintained in tables within the Data Access Control System database and operational permission checking subsystem. The table that relates the entitlement codes for a service to the subservice names for that service is called the map for the service.

Requirements are transported on LSEG Real-Time Distribution System in protocol messages called locks. The DACSLock API provides functions to manipulate locks in a manner such that the source application need not know any of the details of the encoding scheme or message structure. For a source server to be Data Access Control System compliant, based on content, it must publish locks for the items it publishes; i.e., the source server application must produce lock events. Any item published without a lock or with a null lock is available to everybody permissioned for that service, even those without subservice permissions.

If a source server introduces (new) data to LSEG Real-Time Distribution System that originates outside the network on which the application is running, then the application developer is also responsible for providing the map information for the service.

In addition to source servers that publish new data directly from a vendor, there are also compound servers that gather market information from other sources, manipulate that information, and then re-publish it on LSEG Real-Time Distribution System. These servers must read locks from the Transport API Java Edition to combine them before publishing compound items. Again, the DACSLock API provides functions to assist with this process. Compound sources must also use a special form of user ID when connecting to the LSEG Real-Time Distribution System network.

NOTE: Subject-based sources do not require locks.

1.2 IDN Versus LSEG Real-Time

LSEG's new ultra-high speed network LSEG Real-Time has replaced the older IDN network. All references herein are made to LSEG Real-Time. However, for historical reasons in the Data Access Control System administrative screens, this network is still referred to as IDN. For this reason, the terms LSEG Real-Time and IDN are interchangeable throughout this document.

1.3 Audience

This guide is intended for software programmers who wish to incorporate the DACSLocks into the development of their source applications.

1.4 Organization of Manual

The material presented in this guide is divided into the following sections:

CHAPTER	CONTENT / TOPIC
Chapter 1, Introduction	A description of this manual and its conventions.
Chapter 2, Requirements for Compliant Source Server Applications	For subservice-level permissioning, a DACS-compliant source server must satisfy a series of requirements.
Chapter 3, DACSLock API	Explains the data flow of a DACSLock and its operations.
Chapter 4, DACSLock API Interface	Describes DACSLock API components and their required properties.
Appendix A, Example Program	Lists an example program that was created using the DACSLock API.

Table 1: Manual Overview

1.5 References

- *Transport API Java Edition Developers Guide*
- *DACSLock API Reference Manual*

1.6 Conventions

1.6.1 Typographic

This manual observes the following typographic conventions:

- Java classes, methods, and types are shown in **`Courier New`** font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.
- Longer code examples (one or more lines of code) are shown in Courier New font against an orange background. Code comments are shown in green font color. For example:

```
/* calculate the length of the new lock */
int lock = dacsInterface.calculateLength(serviceId, operation, productEntityList,
    productEntityListLength, error);
```

1.6.2 Programming

Enterprise Transport API Java Edition Standard conventions were followed.

1.7 Glossary

TERM	DESCRIPTION
API	Application Programming Interface
Application	A program that accesses data from and/or publishes data to the system.
Compound Item	A data item prepared from data items retrieved from the system.
Concrete service	A set of real-time data items published by a source server. Each concrete service is identified on the Data Access Control System by a unique name (known as a network).
Data Access Control System	An entitlement tool that allows customers to automatically control who is permitted to use which sets of data in their LSEG Real-Time Distribution System deployment.
LDF Direct	Data Feed Direct
LSEG Real-Time	LSEG's open, global, ultra-high-speed network and hosting environment, which allows users to access and share various types of content.
Entitlement Code	If a vendor service is permissioned down to the subservice level, an entitlement code must be provided with each item. Based on mapping tables provided by the vendor, the Data Access Control System uses this code to determine the information provider and/or vendor product associated with an item.
Exchange	A commercial establishment at which or through which trading of financial instruments takes place. Exchanges are information providers.
Exchange Map Logic	The logic used to construct requirements when an item is supplied by more than one exchange. If OR logic is used, the user only needs permission to access one of the exchanges that supplies the item. If AND logic is used, the user must have permission to access all of the exchanges that supply the item.
Map Program	An application associated with a particular vendor service which requests permissioning data from the vendor host and then uses that data to construct various mapping tables required by the Data Access Control System.
Mapping Tables	These tables are used by the Data Access Control System to derive the requirement for an item. They map entitlement codes to vendor products and, if applicable, to information providers (exchanges and specialist services).
Network Service	See concrete service.
PE	Permissionable Entity. A number used to designate the permissioning basis of a data item on LSEG Real-Time (or LSEG Real-Time Distribution System). (Same as entitlement code.)
Permissioning	The control of access to and publication of data items by users.
Product	A subset of the data items delivered by an information vendor for which there is a single charge (based on vendor criteria).
Product Map Logic	The logic used to construct requirements when an item is supplied by more than one product. If OR logic is used, the user only needs permission to access one of the products that supplies the item. If AND logic is used, the user must have permission to access all of the products that supply the item.
Profile	Information, including a list of subservices, that is used during permission checking. There is a profile associated with each user.
Service	This term is used in two ways by the Data Access Control System in a market data system environment. See concrete service and vendor service.
Service ID	A unique, numeric ID assigned to each network service. On an LSEG Real-Time Distribution System network, all valid service IDs for a particular system are listed in the global configuration file rmds.cnf.

Table 2: Glossary and Acronyms

TERM	DESCRIPTION
Source	An application or server capable of supplying or transmitting information.
Source Server	An application program which provides a concrete service. More than one source server can provide the same concrete service.
Specialist Data Service	A set of data items provided by a third party (i.e. not from an exchange and not from the vendor delivering the data items to the site).
Subservice	A named set of items delivered by a vendor which are authorized as a group; e.g. all instruments traded on NYSE or all items that make up the product Securities 2000.
Subservice Type	There are three types of subservices: • Products • Exchanges • Specialist Service
User	A person with a unique, system-wide name.
Vendor Service	A vendor may offer more than one type of data delivery service to a customer. For example, LSEG provides the LSEG Real-Time service. Each vendor service is identified on the Data Access Control System by a unique name. Each vendor service is associated with one or more concrete services. For example the LSEG Real-Time service may be published on the network by any combination of these concrete services: IDN Selectserver and Reuters Data Feed.

Table 2: Glossary and Acronyms (Continued)

2 Requirements for Compliant Source Server Applications

2.1 Overview

To permission at the subservice level, the item's permissioning requirements must be available at locations other than just the source. For the Data Access Control System to permission a source service below the service level, the source must:

1. Create locks containing permissioning information.
2. Map entitlement codes in the locks to subservice names.

For subservice level permissioning, a DACS-compliant source server must:

- Establish Subservice Names
- Define Entitlement Codes
- Create and Write Locks
- Publish the Map
- Read the Map and Supply it to the Data Access Control System Station

2.2 Establish Subservice Names

If a source server provides a service that is subdivided into subservices, each subservice must have a symbolic name. For LSEG Real-Time, symbolic names represent LSEG products, exchanges, or specialist data services, and are subsets of the service being provided by the vendor. An item might be in zero, one, or more subservices.

At the time a source publishes an item on the LSEG Real-Time Distribution System, the source must associate with the item the identities of any subservices to which the item belongs. For example, an LSEG Real-Time source might identify that an item belongs to the subset of items from the New York Stock Exchange and is part of the Equities 2000 product.

For users, the system administrator grants or denies access to items in the various subservices using symbolic names. The system administrator performing permissioning will not deal with arbitrary numeric encodings such as via PEs (Permissionable Entity).

2.3 Define Entitlement Codes

The second requirement is that an entitlement code (a number) must be associated with each subservice name. A subservice can have one or more associated entitlement codes. Whenever a source publishes an item, the source designates the subservice(s) to which the item belongs as a Boolean expression in entitlement codes. Entitlement codes are needed when combining permissioning information from multiple sources to permission compound items (i.e., made from more than one source).

The PE used for permissioning on IDN (FID 1, PROD_PERM) is an example of an entitlement code.

The list of subservice names and associated entitlement codes is called the ***map*** for the source.

2.4 Create and Write Locks

Whenever a source server opens a data stream, it must write a lock containing the item's entitlement code expression. The source server must use an existing LSEG Real-Time API to publish permissionable data on the LSEG Real-Time Distribution System.

One of the capabilities of such an API is to create the lock. The arguments to the API function include a list of entitlement codes and the Boolean operator (AND or OR), which indicates how to logically combine them. After the lock is created, it must be posted to the LSEG Real-Time Distribution System.

A source application can post a revised lock at any time.

2.5 Publish the Map

The source must publish, as data items, the map of its entitlement codes and subservice names (or use **map_generic** to load a map through the use of a file). As an example, the map for the LSEG Real-Time service is published as a series of RICs referred to as the Reuters Product Definition Pages.[1]

Within the map data there must be a readily available data item with a date-time stamp. The value of this date-time stamp must be the date and time at which the map was last changed. The objective is to permit the application that reads the map to read a single item and determine whether the remaining data has changed (and thus needs to be reread).

The map items (records or pages) must be available to a user and application that have no subservice permissions so that the map can be retrieved at a new site that does not yet have permissions distributed.

NOTE: For LSEG Real-Time services, template files containing preliminary mapping information are provided with the Data Access Control System software so that subservice permissioning can be set up before the latest map is retrieved from the source. If a template file is not provided with a third-party source application, it is important to not require subservice permissioning so that the map can be retrieved from the source.

1. Refer to the *Reuters Product Definition Pages User Guide* to see how LSEG communicates permissioning information for its products.

2.6 Read the Map and Supply it to the Data Access Control System Station

There are two ways to load map data:

- Use the Generic map collect program (**map_generic**).
- Download the latest map.

The Data Access Control System database must contain the source's map data so that:

- The Data Access Control System administrator can assign permissions to subservices for a service
- The Data Access Control System operational subsystem can perform permission checks

As mentioned previously, the source should publish this map. Additionally, the source application developer should provide a map collection program designed to:

- Request the map items from the source.
- Determine if there were any changes since the last map was received.
- Convert any revised information into a file that can be loaded into the Data Access Control System database.

The source application developer may also want to provide a map monitor program that can run periodically to retrieve the latest map and see if any changes have occurred since the last map collection (by checking the date-time stamp). Based on the status reported by the monitor program, the administrator knows when the map collection program needs to be run.

The Data Access Control System does not care about the format of the map published by the source as long as the map collection program for that source produces an appropriately formatted permission map file.

For further details on the map collect and proper permission map file formatting, refer to the *DACSLock API Reference Manual* specific to the version of the Data Access Control System that you run.

TIP: The Data Access Control System software package comes with a map collection program for the LSEG Real-Time service.

3 DACSLock API

3.1 DACSLock Operation

The DACSLock contains requirements for an item that a vendor source deems necessary. On the Market Data Client LAN, users are entitled to specific capabilities. Therefore, when a Lock, which contains requirements, is tested against the capabilities of the user, enough information is available to permission the following:

PERMISSIONING OPERATION
USER -> APPLICATION
USER -> SERVICE
USER -> SUB-SERVICE (entitlement codes)

Table 3: Data Access Control System Permissioning Capabilities

DACSLocks are critical to the operation of the Data Access Control System for content-based sources. Subject-based sources do not require locks. The DACSLock contains the requirements for the requested item. The data flow of a DACSLock and its operation are depicted in the remaining sections of this chapter.

3.2 Forming a DACSLock

The source server is responsible for creating a DACSLock. What information is encoded within the DACSLock is vendor-specific. Two examples are presented to clarify the formation of DACSLocks with respect to a source server. Figure 1 demonstrates the input requirements, the DACSLock API to be called, and the transport mechanism of the DACSLock from the source server to the market data client application.

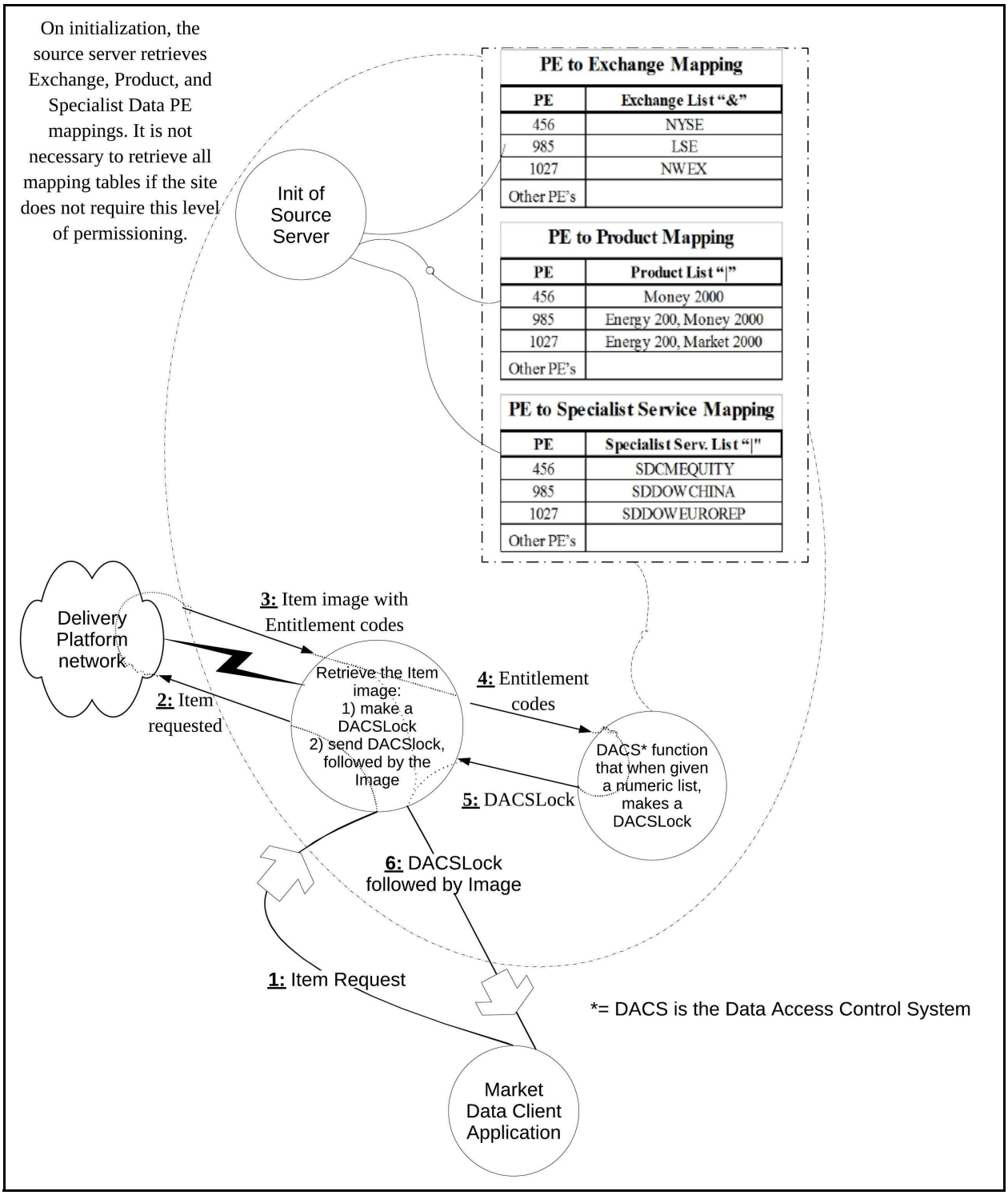

Figure 1. Forming a DACSLock for LSEG Real Time Direct

Figure 1 presents the following information:

- The Data Access Control System Station (via use of its map collect utility) is responsible for retrieving Exchange, Product, and Subservice mappings. For dynamic mapping, the LSEG Real-Time server must retrieve mapping tables from the datafeed line.
- It is up to the LSEG Real-Time server if it retrieves all the mapping tables based on the customer site requirements and on the capabilities of the server's datafeed line.
- The DACSLock API function that makes the DACSLock receives the item's list of entitlement codes. For most Items, this is a single PE. However, other items (e.g., a NEWS2000 Item) can have PE lists that include up to 256 entitlement codes. For this reason, the DACSLock API function might include an operator (**AND**/**OR**) with its PE list. Thus, in the case of a NEWS2000 item, the PE list is assigned an **OR** operator, while other PE lists might include the **AND** operator.
- The DACSLock must be sent before the image so that the Market Data Client application (via the Transport API) can permission the item as soon as the image arrives, instead of having to hold the image and wait for the Lock.

3.3 DACSLock Contents

A DACSLock must contain information relevant to the requirements for the requested item. Because a DACSLock needs to minimize communication costs, a source server encodes these requirements into a single numeric number. In Figure 1, you can see that the source server creates a table that maps these textual requirements to associated numeric values. By supplying an operator to the DACSLock API function that creates DACSLocks, complex requirements can be managed by the source server. Thus an item's requirements are determined by comparing its PE to the mapping tables.

Figure 1 illustrates the following requirements in the source server mapping tables:

PE	REQUIREMENTS
456	NYSE & Money 2000 & (NY | LONDON)
985	LSE & (Energy 2000 | Money 2000) & NY
1027	NWEX & (Energy 2000 | Markets 2000) & LONDON

Table 4: Item Requirement Formulations

A source server can add requirements to the DACSLock for an item by including extra entitlement codes in the PE list assigned to the appropriate DACSLock API class. For example, if a Source Server specifies that only a **Page_Call** application can use an item, then the source server creates a new PE with that requirement and attaches it to the PE for that item using the AND operator.

PE	REQUIREMENTS
5000	Page_Call

Table 5: PE Example that can Add Extra Requirements to an Item

So if an Item has a **PE = 456**, and the Source Server requires only **Page_Call** access, then the Source Server passes entitlement codes **456** and **5000** as parameters to the appropriate DACSLock API function that builds the DACSLocks from the PE lists.

3.4 Compression of DACSLocks

To minimize physical size, DACSLocks are compressed based on the Binary Coded Decimal (BCD) algorithm. For example, the DACSLock API must make a lock with the following PE requirements:

LSEG REAL-TIME (IDN)	BRIDGE
1027 & (456 | 985) & 5000	1 & 2

Table 6: PE Requirements for a Compound Item

The BCD compression algorithm converts numeric PE values and interprets operator functions according to the following table:

OPERATION	REPLACED NIBBLE VALUE
numeric 0 – 9	0 - 9
“&” and operation	A
“|” or operation	B
“EOF” End of DACSLock	C
“EOS” End of Source Server PE List	D

Table 7: Operator and BCD Interpretation Table

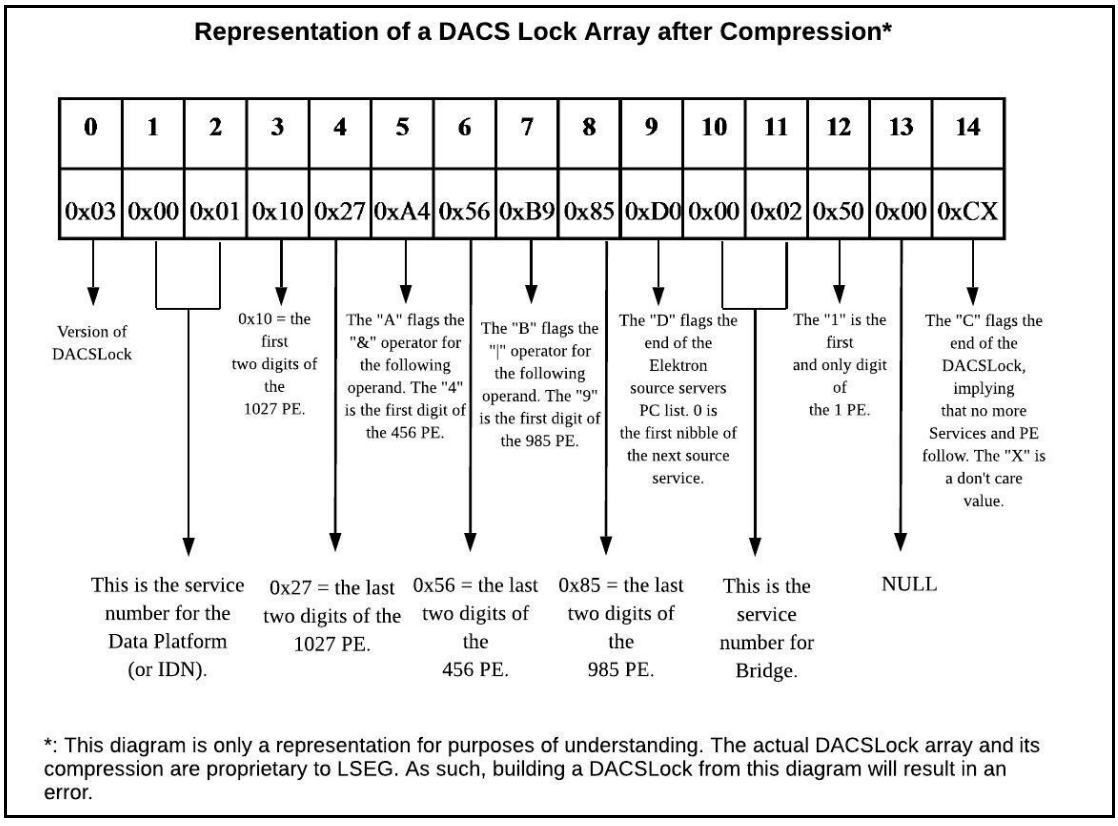

Figure 2. DACSLock Array after Compression

By using the compression mapping in Table 7, the DACSLock API creates the unsigned character array in Figure 2, which illustrates some of the advantages to using this compression:

- The compression does not limit the maximum value of a PE, other than a possible machine maximum that can be manipulated easily with the software language (i.e., (2^{32})-1 for an unsigned long).
- BCD compression is simple and fast without resulting in excessive computation.
- The compressed DACSLock is smaller then the actual ASCII string if it were sent.

3.5 Compounding DACSLocks

The previous example, in Figure 2, had two source server PE lists within the DACSLock. This situation is possible when a compound server combines items from more than one source server type. To create a compound DACSLock, the compound server calls the DACSLock API function that, when given the constituent item DACSLocks, creates a DACSLocks that contains all of the constituent item requirements. To minimize the size of a DACSLock, the DACSLock API uses Boolean algebra rules to minimize entitlement codes within the PE list whenever possible.

Some rules are:

```
(A | B) & A = A
A & A & B = A & B
A | A | B = A | B
```

A compound server stores all constituent item DACSLocks until the item is no longer required. This functionality is required in the event a new DACSLock is received by the compound server for an open item, in which case the compound server must update the compound item's DACSLock. After updating a DACSLock, the compound server must forward its new compound item DACSLock to those servers that have the compound item open.

3.6 Transport of DACSLocks

DACSLocks for compound servers might grow larger then the maximum size of a single message, in which case the server splits the DACSLock across multiple messages.

3.7 Compound DACSLock in Relation to Permissioning

The previous sections explain the rules for constructing and transporting the DACSLocks. Figure 3 shows the data flow functionality of a DACSLock in an operating environment with two source servers (Bridge and LSEG Real Time Direct), a compound server, and a Market Data client application. In this example the Market Data client application requests an item from the compound server. The item is made from constituent items from the LSEG Real Time Direct and Bridge Servers.

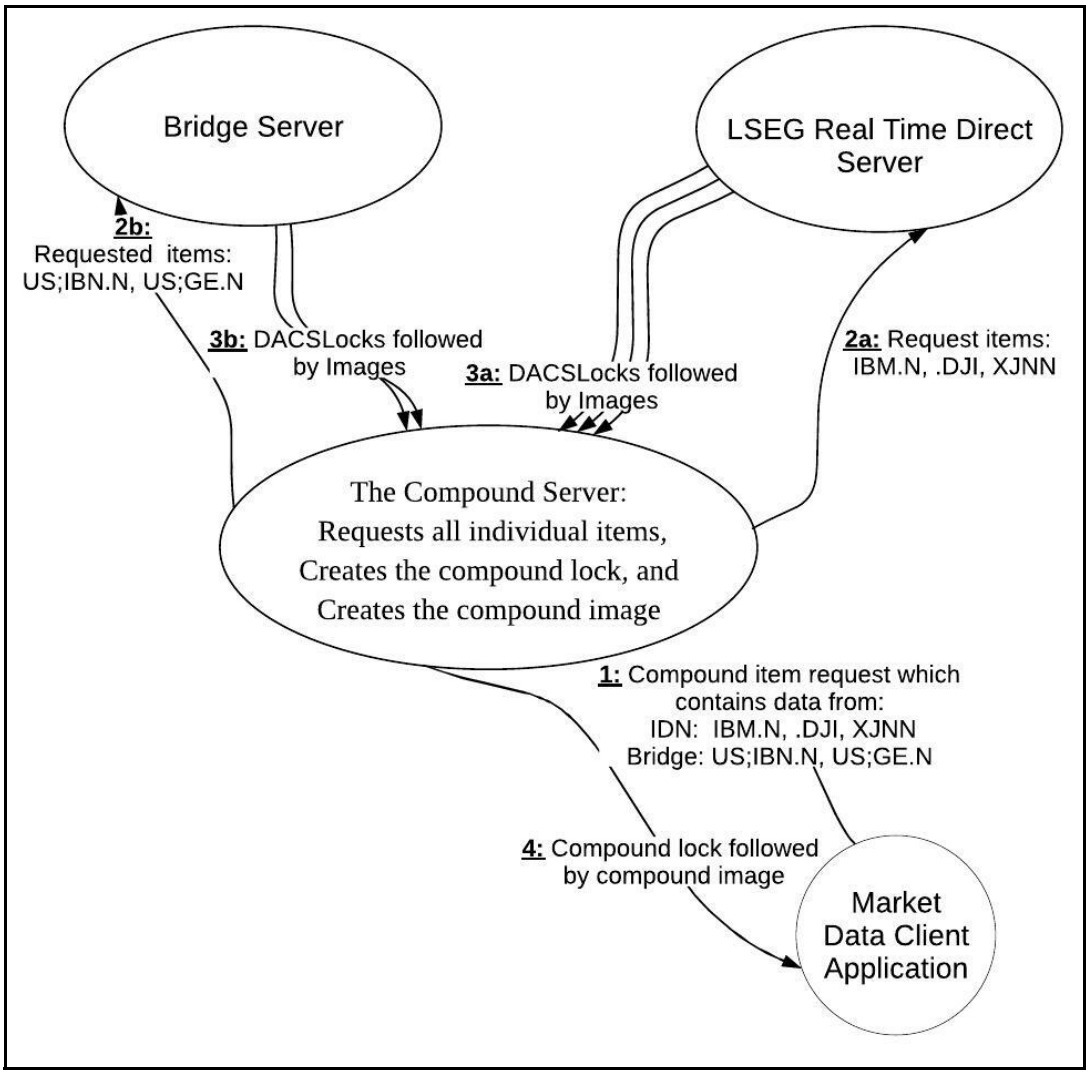

Figure 3. Compound Locks

4 DACSLock API Interface

This chapter describes the following components of the DACSLock API Interface:

- JDACSLock Methods
- Supporting Objects
- Constant Definitions

4.1 JDACSLock Methods

The **`JDacsLock`** interface allows the application to obtain lock data created by specific parameters or to obtain a combined lock. The table below summarizes the methods provided by the **`JDacsLock`** interface.

METHOD	DESCRIPTION
createJDacsLock	This static method returns an instance of the **`JDacsLock`** that provides all the methods in this table.
createLock	This static method returns an instance of **`DacsLock`**.
createDacsError	This static method returns an instance of **`DacsError`**.
createLock	This method generates a new lock from the following: • **serviceId**: identifies the serviceId with which the lock is associated. • **operation**: a character, that defines a valid operation. Refer to **`DacsOperation`** for a definition of operations. • **ProductEntityList** is a sorted array of product entities. The **productEntityListLength** is set by the application to identify how many array elements apply. Note that the actual array length might be greater than the **productEntityListLength**. The application can reuse this array when creating several locks to avoid garbage collection. The lock instance is owned by an application and obtained via the **`createLock()`** method and the application should allocate the data buffer. The **`createLock()`** method will populate lock data into the buffer. If the buffer is too small, a return value and **`errorId`** will indicate this. The data (**ByteBuffer**) will have its position set to the start of lock bytes and is limited by the end of the lock. This method takes a **`DacsError`** object and populates the **`errorId`** and text an error occurs. An error instance is own by an application, and obtained via the **`createError()`** method. A return value will indicate whether the operation was successful. For a list of Data Access Control System return codes, refer to Section 4.3.1.
calculateLockLength	This method calculates the length of a lock from the following information: **serviceId**, **operation**, and **productEntityList** (refer to the **`createLock`** method for descriptions of these parameters). These parameters are the same parameters from the **`createLock`** method, with the exception of **`DacsLock`**, as it does not populate the lock. The application can use this method prior to **`createLock()`** to find the length of the lock data and thus determine the ideal size of the **`DacsLock.data()`** buffer.

Table 8: JDacsLock Interface

METHOD	DESCRIPTION
combineLock	This method generates a new lock from the **LockList**. **LockList** in an array of Locks. The **lockListLength** is set by application to identify how many array elements apply. Note that the actual array length can be greater than the **lockListLength**. The application may reuse this array when creating several locks to avoid garbage collection. The lock instance is owned by the application and obtained via the **createLock()** method. The application should allocate the data buffer. This method populates lock data into this buffer. If the buffer is too small, a return value and **errorId** will indicate this. The data (**ByteBuffer**) will have its position set to the start of the lock bytes and limited by the end of lock. If error conditions happen, the method populates a **DacsError** object with an **errorId** and associated text. The error instance is owned by the application, and obtained via the **createError()** method. A return value will indicate whether the operation was successfully. For a list of Data Access Control System return codes, refer to Section 4.3.1.
calculateCombineLockLength	This method calculates the length of a new lock generated from the **LockList**. The parameters are identical to the parameters of the **combineLock** method, with the exception of **DacsLock**, as it does not populate the lock. An application can use this method prior to **combineLock()** to determine the length of lock data and thus the ideal size of the **DacsLock.data()** buffer.
equals	A static utility method that compares two **ByteBuffer**s for equality. The method returns **true** if the length of buffers (from position to limit) is equal and the data is identical. This method is useful when comparing two locks.

Table 8: JDacsLock Interface (Continued)

4.2 Supporting Objects

The **JDacsLock** interface utilizes the following interfaces:

- DacsLock Interface
- DacsError Interface

4.2.1 DacsLock Interface

The **DacsLock** interface consists of the following methods:

METHOD	DESCRIPTION
data	Accessor methods that allow the application to set and get an instance of **ByteBuffer** associated with this object.
length	Returns the length of lock data.
startPosition	Returns the starting position of the lock data.
equals	Compares the lock data with the **DacsLock** parameter. If the data is equal, it returns true; false otherwise. In the event of an error, **equals** populates a **DacsError** object with an **errorId** and **text** as appropriate. The error instance is owned by the application and obtained via the **createError()** method. A return value will indicate whether the operation was successful. For a list of Data Access Control System return codes, refer to Section 4.3.1.
copy	Performs a deep copy of this DACSLock into the **DacsLock** parameter. In the event of an error, **copy** takes a **DacsError** object and populates its **errorId** and **text** as appropriate. The error instance is owned by the application, and obtained via the **createError()** method. A return value will indicate whether the operation was successful. For a list of Data Access Control System return codes, refer to Section 4.3.1.

Table 9: DacsLock Interface Methods

4.2.2 DacsError Interface

The **DacsError** interface consists of the following methods:

errorId	A DACS-specific return code, that specifies an error if it occurs. For a list of Data Access Control System return codes, refer to Section 4.3.1.
text()	Returns text describing the error identified by the specific **errorId**.

Table 10: DacsError Interface Methods

4.3 Constant Definitions

The **JDacsLock** interface uses constants defined in:

- DacsReturnCodes Definitions
- The DacsOperations Interface

4.3.1 DacsReturnCodes Definitions

The **DacsReturnCodes** class uses the following return codes:

ERROR	DESCRIPTION
NO_ERROR	The operation was successful.
FAULT	An unexpected fault occurred.
BAD_OPERATOR	The operator parameter is out of range.
NO_LOCKS	A locks parameter was not supplied.
INVALID_LOCK	An invalid lock parameter is supplied.
INVALID_ARGUMENT	A null argument was supplied.
V4_COMBINE_NOT_ALLOWED	A lock list contained locks with different versions.
BUFFER_TOO_SMALL	The lock buffer is too small for lock data.

Table 11: DacsReturnCodes Definitions

4.3.2 DacsOperations Interface

The **DacsOperations** interface uses the following operations:

OPERATION	DESCRIPTION
AND_OPERATION	The type of ‘AND’ operator.
OR_OPERATION	The type of ‘OR’ operator.

Table 12: DacsOperations Operations

Appendix A Example Program

The following is some example code created using the DACSLock API:

```
/*|-----------------------------------------------------------------------------
 *|            This source code is provided under the Apache 2.0 license      --
 *|  and is provided AS IS with no warranty or guarantee of fit for purpose.  --
 *|                See the project's LICENSE.md for details.                  --
 *|           Copyright (C) 2019-2022 LSEG. All rights reserved.        --
 *|-----------------------------------------------------------------------------
 */

package com.refinitiv.eta.examples.authlock;

import com.refinitiv.eta.dacs.*;
import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * This is the main file for the AuthLockExample application. Its purpose is to
 * demonstrate the functionality of the DACS library.
 *
 * <p>
 * <em>Running the application:</em>
 * <p>
 * Change directory to the <i>Java</i> directory and issue the following <i>Gradle</i> command.
 * <p>
 * Linux: ./gradlew runAuthLockExample<br>
 * Windows: gradlew.bat runAuthLockExample
 */
public class AuthLockExample
{
    JDacsLock _dacsInterface = JDacsLock.createJDacsLock();
    DacsError _error = JDacsLock.createDacsError();
    PrimitiveByteBufferPool _pool = new PrimitiveByteBufferPool();

    class PrimitiveByteBufferPool
    {
        //create a primitive ByteBuffer pool
        private List<ByteBuffer> _smallBuffersPool = new ArrayList<ByteBuffer>();
        private List<ByteBuffer> _mediumBuffersPool = new ArrayList<ByteBuffer>();
        private List<ByteBuffer> _largeBuffersPool = new ArrayList<ByteBuffer>();

        private static final int _smallBufferSize = 10;
        private static final int _mediumBufferSize = 100;
        private static final int _largeBufferSize = 1000;
```

```
        // create 6 buffers of each size and add them to pools
        ByteBuffer buffer;
        {
            for (int i = 0; i < 6; i++ )
            {
                buffer = ByteBuffer.allocate(_smallBufferSize);
                _smallBuffersPool.add(buffer);
                buffer = ByteBuffer.allocate(_mediumBufferSize);
                _mediumBuffersPool.add(buffer);
                buffer = ByteBuffer.allocate(_largeBufferSize);
                _largeBuffersPool.add(buffer);
            }
        }

        ByteBuffer getBuffer(int len) {
            if (len <= _smallBufferSize)
            {
                if (_smallBuffersPool.size() == 0)
                    return null;
                else
                    return _smallBuffersPool.remove(0);
            }
            else if (len <= _mediumBufferSize)
            {
                if (_mediumBuffersPool.size() == 0)
                    return null;
                else
                    return _mediumBuffersPool.remove(0);
            }
            else if (len <= _largeBufferSize)
            {
                if (_largeBuffersPool.size() == 0)
                    return null;
                else
                    return _largeBuffersPool.remove(0);
            }
            else
                return null;
        }

            void bufferToPool(ByteBuffer buffer, int len) {
                buffer.position(0);
                buffer.limit(buffer.capacity());

                if (len == _smallBufferSize) {
                    _smallBuffersPool.add(buffer);
                } else if (len == _mediumBufferSize) {
                    _mediumBuffersPool.add(buffer);
                } else if (len == _largeBufferSize) {
                    _largeBuffersPool.add(buffer);
```

```
                } else {
                    System.err.println("application error, buffer size not poolable ");
                }
            }
        }

        /**
         * Auth lock.
         */
        public void authLock() {
            // test calculateLockLength method
            // set the input parameters: serviceId, operation, productEntityList
            int serviceId = 5000;
            char operation = DacsOperations.OR_OPERATION;
            long[] productEntityList = new long[256];
            int productEntityListLength = 1;
            productEntityList[0] = 62;

            // calculate the length of the new lock
            System.out.println("calculate lock1 length");
            int len = _dacsInterface.calculateLockLength(serviceId, operation,
                productEntityList, productEntityListLength, _error);
            if (len >= 0) {
                System.out.println("calculateLockLength() Success, length = " + len);
            } else {
                System.err.println("calculateLockLength() failed " + _error.errorId() + " - " +
                    _error.text());

            }

            // test createLock method
            // create the lock object
            DacsLock lock1 = JDacsLock.createLock();

            // use the calculated lock data length to get the ByteBuffer from pool
            ByteBuffer lockData = _pool.getBuffer(len);
            if (lockData == null) {
                System.err.println("No buffers of this size in pool");
            }
            lock1.data(lockData);

            // populate the lock data
            System.out.println("\ncreate lock1 ");
            int ret = _dacsInterface.createLock(serviceId, operation,
                    productEntityList, productEntityListLength, lock1, _error);
            if (ret == DacsReturnCodes.NO_ERROR) {
                System.out.println("createLock() - Success");
            } else {
                System.err.println("createLock() failed " + _error.errorId() + " - " + _error.text());
            }
```

```
// create another lock
productEntityList[1] = 144;// note that the array must be sorted
productEntityListLength = 2;

// create the lock object
DacsLock lock2 = JDacsLock.createLock();

// get the ByteBuffer of arbitrary from pool without checking the lock length first
lockData = _pool.getBuffer(9);
lock2.data(lockData);

// populate the lock data
System.out.println("\ncreate lock2 ");
ret = _dacsInterface.createLock(serviceId, operation,
        productEntityList, productEntityListLength, lock2, _error);
if (ret == DacsReturnCodes.NO_ERROR) {
    System.out.println("createLock() - Success");
} else {
    System.err.println("createLock() failed " + _error.errorId() + " - " + _error.text());
}

// create another lock
productEntityList[1] = 63;// note that the array must be sorted
productEntityList[2] = 144;
productEntityListLength = 3;

// create the lock object
DacsLock lock3 = JDacsLock.createLock();

// get the ByteBuffer of arbitrary from pool without checking the lock length first
lockData = _pool.getBuffer(30);
lock3.data(lockData);

// populate the lock data
System.out.println("\ncreate lock3 ");
ret = _dacsInterface.createLock(serviceId, operation,
        productEntityList, productEntityListLength, lock3, _error);
if (ret == DacsReturnCodes.NO_ERROR) {
    System.out.println("createLock() - Success");
} else {
    System.err.println("createLock() failed " + _error.errorId() + " - " + _error.text());
}

// create another lock of a single "0" PE
productEntityList[0] = 0;
productEntityList[1] = 0;
productEntityList[2] = 0;
productEntityListLength = 1;
```

```
        // create the lock object
        DacsLock lock4 = JDacsLock.createLock();

        // get the ByteBuffer of arbitrary from pool without checking the lock length first
        lockData = _pool.getBuffer(10);
        lock4.data(lockData);

        // populate the lock data
        System.out.println("\ncreate lock4 ");
        ret = _dacsInterface.createLock(serviceId, operation,
                productEntityList, productEntityListLength, lock4, _error);
        if (ret == DacsReturnCodes.NO_ERROR){
            System.out.println("createLock() - Success");
        } else {
            System.err.println("createLock() failed " + _error.errorId() + " - " + _error.text());
        }

        // copy lock3 into new lock5
        DacsLock lock5 = JDacsLock.createLock();
        lock5.data(_pool.getBuffer(lock3.length()));
        System.out.println("\ncopy lock3 to lock5 ");
        lock3.copy(lock5, _error);
        if (ret == DacsReturnCodes.NO_ERROR) {
            System.out.println("Lock.Copy() - Success");
        } else {
            System.err.println("Lock.Copy() failed " + _error.errorId() + " - " + _error.text());
        }

        // compare the locks using method on DacsLock
        System.out.println("\ncompare lock3 and lock5 ");
        boolean equals = lock3.equals(lock5);
        if (equals == true) {
            System.out.println("locks equal ");
        } else {
            System.out.println("locks not equal ");
        }

        // compare the locks data using method on JDacsLock
        System.out.println("\ncompare lock3 and lock5 ");
        equals = JDacsLock.equals(lock3.data(), lock5.data());
        if (equals == true) {
            System.out.println("locks equal ");
        } else {
            System.out.println("locks not equal ");
        }

        System.out.println("\nChange Service id in lock2");
        serviceId = 30;
        productEntityList[0] = 62;// note that the array must be sorted
        productEntityList[1] = 144;
```

```
productEntityListLength = 2;

// reuse lock2, a new lock data will be populated inside the ByteBuffer
// populate the lock data
System.out.println("\nreuse lock2 - create ");
ret = _dacsInterface.createLock(serviceId, operation,
        productEntityList, productEntityListLength, lock2, _error);
if (ret == DacsReturnCodes.NO_ERROR) {
    System.out.println("createLock() - Success");
} else {
    System.err.println("createLock() failed " + _error.errorId() + " - " + _error.text());
}

// Compare is not equal since different PE lists
System.out.println("\nCompare lock1 and lock2");
equals = lock1.equals(lock2);
if (equals == true) {
    System.out.println("locks equal ");
} else {
    System.out.println("locks not equal ");
}

// Compare is not equal since different serviceID's and PE's
System.out.println("\nCompare lock2 and lock3");
equals = lock2.equals(lock3);
if (equals == true) {
    System.out.println("locks equal ");
} else {
    System.out.println("locks not equal ");
}

DacsLock [] lockList = new DacsLock[100];
int lockListLength = 3;
lockList[0] = lock1;
lockList[1] = lock2;
lockList[2] = lock3;

// get the length of combined lock
System.out.println("\nCalculate combined lock length ");
len = _dacsInterface.calculateCombineLockLength(lockList, lockListLength, _error);
if (len >= 0) {
    System.out.println("calculateCombineLockLength() Success, length = " + len);
}
else
{
    System.err.println("calculateCombineLockLength() failed " + _error.errorId() + " - " +
        _error.text());
}

// populate the combined lock
```

```
        DacsLock combinedLock = JDacsLock.createLock();
        combinedLock.data(_pool.getBuffer(len));

        System.out.println("\nCombine lock1, lock2, and lock3 into combineLock");
        ret = _dacsInterface.combineLock(combinedLock, lockList, lockListLength, _error);
        if (ret == DacsReturnCodes.NO_ERROR) {
            System.out.println("combineLock() Success");
        } else {
            System.err.println("combineLock() failed " + _error.errorId() + " - " + _error.text());
        }

        // return lock1 data to pool
        int bufSize = lock1.data().capacity();
        _pool.bufferToPool(lock1.data(), bufSize);
    }

    /**
     * The main method.
     *
     * @param args the arguments
     */
    public static void main(String[] args)
    {
        AuthLockExample dacsExample = new AuthLockExample();
        dacsExample.authLock();
        System.exit(0);
    }

}
```

LSEG DATA & ANALYTICS

Document ID: ETAJ391L1EDAC.250
Date of issue: September 2025