White Paper

EMC® DOCUMENTUM® FOUNDATION CLASSES SESSION MANAGEMENT

Abstract

This white paper explains session management in Documentum Foundation Classes (DFC). It provides detailed information about sessions, session manager, pooling, transaction, and object management. This document highlights the best practices for DFC session management, and provides useful tips to diagnose session-related problems.

July 2011

Part Number H8843

Table of Contents

Executive summary

This white paper provides a general overview of the following features:

- Session and Session Manager
- Best practices for Session Management
- Pooling
- Transaction handling
- Object management using sessions

The last section of this paper provides information about analyzing session-related issues and collecting relevant data to verify and resolve session-related issues.

Audience

This paper is intended for application developers using Documentum Foundation Classes (DFC). It assumes that the readers possess a basic knowledge of DFC. This document will help Documentum developers to engineer custom DFC-based applications, and support engineers to understand and resolve production issues.

The paper focuses on functional aspects and presents technical information explaining the working of DFC Session Management. This paper also defines the terminology used. However, you are recommended to refer to the Documentum Content Server and Documentum Foundation Classes Guides for more information.

Overview

Documentum architecture follows the client/server model. DFC-based applications are client programs even if DFC runs on the same machine as Content Server.

The getLocalClient() method of DfClientX initializes DFC. The instantiation of DFC using the getLocalClient() method performs much of the runtime context initialization related tasks such as assigning policy permission, loading the DFC property file, creating the DFC identity on the global repository, and registering the DFC identity in the global repository.

DFC initialization is performed only once when the user calls the getLocalClient() method for the first time. Calling the getLocalClient() method again returns the same instance of IDfClient and there is no performance impact.

The getLocalClient() method returns an object of IDfClient type that provides methods to retrieve Repository, Docbroker, and Server information. In addition, the object provides methods to create a business object framework (BOF) module, session manager, and so on. It also contains IDfSession-related methods such as the newSession() method for backward compatibility.

Note: You are recommended to use IDfSessionManager to get or release sessions. Do not use the IDfClient exposed session-related methods.

Session and Session Manager Overview

A session represents a connection with the repository because it encapsulates the underlying socket connection to the repository. All interaction between a DFC client application and the repository takes place through a session. Sessions are obtained using the session manager.

A session provides the following benefits:

- Provides functionality to create and retrieve persistent objects
- Maintains the state of the interaction between DFC and Content Server
- Maintains a local cache of persistent objects to improve performance
- Provides methods to obtain repository configuration information

A typical DFC application comprises multiple sessions pointing to the same or different repositories. Each session is represented by a unique Session ID.

A session manager is a DFC object that manages sessions. The session manager is logically associated with users and it serves a single user for one or more repositories.

A session manager provides the following functionality:

- Create and release sessions
- Provide session pooling capability
- Handle transactions
- Provide session diagnostic data

Using sessions

A session manager is created using the IDfClient.newSessionManager() method that acts as a factory for sessions.

A session is scoped by the session manager. A session manager cannot use or share a session owned by another session manager.

DFC applications request a session from the session manager using the getSession() method or newSession() method on the IDfSessionManager interface.

Shared and private sessions

The getSession() method returns a shared session, and the newSession() method returns a private session.

Multiple threads share a shared session. The session manager checks the shared session registry that the session manager owns to determine whether a session is available for every getSession() request. If a session is available, the session manager returns the available session. However, if a session is not available, the session manager retrieves a session from the session pool, if pooling is enabled. If pooling is not enabled the session manager creates a session.

The getSession() method does not allow sharing sessions among users or among different session managers. If the developer resets the user identity of the session manager and calls the getSession() method, the session manager returns a new shared session. This behavior is also relevant to different instances of the session manager.

You are recommended to use the getSession() method in multithread applications that perform short-lived activities. Shared sessions use the same underlying socket connection to the repository, and the same server session. While extensive use of the getSession() method can result in performance degradation, it helps to control the growth of unnecessary sessions.

The getSession() method returns the shared session regardless of whether session pooling (dfc.session.pool.enable) is enabled or not.

A private session cannot be shared. Private sessions give complete control of the session state for a specific task. A private session is commonly used for a long running task.

Both private sessions and shared sessions use the pool if pooling is enabled. If private session requests are not carefully controlled, the application will use excessive number of repository sessions.

In a multithread application, it is always safe to use the same shared session across all threads for the read activity. You can use a shared session for the write activity if the threads do not update the same object, simultaneously. For example, using a shared session in multiple threads to update different objects, say thread t1 writes on object x, and thread t2 writes on object y, must not generate any inconsistency.

If two threads share the same session for write access on the same object, then the two threads must be aware of each other and coordinate their activities. This is because sessions are stateful. If one thread establishes an object state while performing a write operation, the other thread may misuse or destroy that state. This issue is beyond the awareness of DFC. This higher level of thread safety is the responsibility of the DFC client code that is sharing the session.

The following sample code snippet illustrates the basic steps involved in getting a session, using it, and releasing it:

```
//create client objects
IDfClientX clientx = new DfClientX();
IDfClient client = clientx.getLocalClient();

//create a Session Manager object
IDfSessionManager sMgr = client.newSessionManager();

//create an IDfLoginInfo object for user creddentials
IDfLoginInfo loginInfo = clientx.getLoginInfo();
loginInfo.setUser(<user>);
loginInfo.setPassword(<pass>);

//bind the Session Manager to the login info
sMgr.setIdentity(<repositoryName>, loginInfo);
IDfSession session = null;

try
{
//get the IDfSession instance by using getSession or newSession
session = sMgr.getSession(<repositoryName>);
//use the session to perform repository functions
……….
……….
}
// catch DFC and application specific exception
Catch ()
{
}
//release the session
Finally
{
If (session != null)
               sMgr.release(session);
}
```

Best practices for using sessions

- Ensure that sessions are always obtained or released using the session manager.
- It is recommended to release the session immediately after use.
- Always release a session in the “finally” block to ensure that the release call is always executed.
- Do not store session objects at a location where the release of objects cannot be guaranteed. For example, it is a bad practice to store a session object as a class member variable or in a cache.
- It is not recommended to use the IdfSession.disconnect() method to disconnect a session. A session must always be released with the same session manager release() method through which the session was acquired. The IDfSessionManager.release() method performs additional tasks such as closing related sessions or sub-connections, aborting unfinished session transactions, and so on.

Ensure that released sessions are not used, released, or disconnected.

Authentication

You must provide authentication information to the server to acquire a session. Since the session manager sends requests for the sessions, you must set the authentication information on the session manager.

DFC provides the following authentication mechanisms:

Setting session manager identity with explicit user credentials

You must encapsulate user credentials in an IDfLoginInfo object and pass the object with the repository name to the session manager setIdentity() method. You can also set the same IDfLoginInfo object for multiple repositories using the “*” character. If you call the setIdentity() method on the session manager that has an identity, the DfServiceException occurs. Use the clearIdentity() or clearIdentities() method to reset old credentials. Ensure that you use the clearIdentities()method carefully. Do not call the clearIdentities()method while the session manager is still in use.

Using login tickets

A login ticket is a token string that you can pass instead of a password to obtain a repository session. You can generate login tickets from the IDfSession object using the following Application Programming Interfaces (APIs):

- **getLoginTicket()**: Use this API to provide additional sessions for a user who has an authenticated session. This technique is employed when a web application that has a Documentum session must build a link to another WDK-based application. This enables a user who is already authenticated, to link to the second application without going through an additional login screen. The ticket is embedded in a URL.
- **getLoginTicketEx or getLoginTicketForUser**: Use this API to allow a session authenticated for a super user, to grant other users access to the repository. You can use this strategy in a workflow method in which the method must perform a task for a user without requesting the user password. This API is also used in J2EE Principal Authentication support that allows a single login to the Web Server and Content Server.

Principal authentication

Principal authentication is the process of getting a session for the principal, who is the user, on the specified repository using a login ticket generated by the login credentials of a trusted authenticator, who is a super user. DFC does not turn on principal support, by default. Use the IDfClient.setPrincipalSupport() method to pass an instance of your IDfPrincipalSupport implementation class that uses super user credentials to act as the trusted authenticator. In addition, set the principal name in the session manager using the setPrincipalName() method. When you set principal support, the behavior of any session manager generated from the IDfClient will change. As a result, the session manager delegates the task of obtaining the session to the principal support object getSession() method to obtain a session for the principal on a specified repository.

DFC includes a default implementation of the IDfPrincipalSupport implementation class that demonstrates a design pattern you can use to build your principal support implementation. Direct use of these classes is not supported because these classes do not provide security for the trusted authenticator password, and the sample may change in future releases.

Trusted authentication

By default, applications running on the Content Server host are allowed to make repository connections as the installation owner without specifying a password. This is known as a trusted login. The domain name of the Installation owner must be the same as the user_auth_target entry of the **server.ini** file. If you do not want to allow trusted logins, set the dfc.session.allow_trusted_login preference in the **dfc.properties** file to false.

Unified login

Unified login allows users to connect to a repository using their Windows login credentials. When the unified login is enabled, the a_silent_login property of the server config object is set to true. The unified login is implemented for the Documentum Desktop application on the Windows platform. Unified login relies on the Windows SID authentication mechanism. Therefore, the client and repository machines must be in the same workgroup or domain.

Session Listener

A Session Listener allows a developer to execute customized code during session creation and release. The developer must write a custom listener class that implements the IDfSessionManagerEventListener interface and defines the onSessionCreate() and onSessionDestroy() methods. The developer can then register the custom listener class with the session manager using the IDfSessionManager.setListener() method.

Consider the following simple implementation of the session listener that prints the session ID at every session creation and destruction:

```
static class SessionListenerImpl implements
IDfSessionManagerEventListener

{

public void onSessionCreate(IDfSession sess) throws DfException

{

System.out.println("Session created: " + sess.getSessionId());

}

public void onSessionDestroy(IDfSession sess) throws DfException

{
```

```
System.out.println("Session Destroyed: " + sess.getSessionId());
}
}
```

The session manager calls these methods while creating or destroying a session object.

Session and object management

In DFC, the session manages persistent object creation and retrieval. Developers must not create a persistent object using a constructor such as the IDfSysObject sysObject = new SysObject() method. Instead, the developer must use the session.newObject() or session.getObject() method. Each object has a reference to the session from which it was created. When an original session is released, you can obtain a reference to it using the getObjectSession() method.

An object whose session has been released, is called a “disconnected object”. DFC handles any operation on a disconnected object, transparently. Developers must not call the getObjectSession() method to release the session obtained.

Although this mechanism is convenient to use, it is not recommended. Instead of relying on disconnected objects to achieve the best performance, the developer is recommended to acquire the session, create or retrieve an object, manipulate the object, and release the session.

Related sessions or sub-connections

Operations such as copy, link, and move can operate on two different repositories. For example, the destination folder can belong to a repository other than the source repository in a copy operation. DFC transparently copies the session to the other repository and completes the operation. The session created for the other repository, which DFC created internally, is called a related session.

You can create the other repository session in one of the following ways:

1. Use the getSession() method from the same session manager associated with the original session. This second session is a peer of the original session and is managed by the session manager as any other session. This approach gives more power over session lifetime and it is the responsibility of the developer to release both sessions.

   ```
   IDfSession peerSession =
   session.getSessionManager().getSession(repository2Name);
   try
   {
   doSomethingUsingRepository2(peerSession);
   }
   finally
   {
   ```

```
session.getSessionManager().releaseSession(peerSession);
}
```

2. Use a related session or sub-connection by calling the IDfSession.getRelatedSession() method. Sub-connections are connections to another repository, initiated from the context of an original session. The lifetime of a related session is dependent on the lifetime of the original session. Developers must not explicitly release a related session.

   ```
   IDfSession relatedSession =
   session.getRelatedSession(repository2Name);  //(WeakSessionHandle)
   Doesn’t affect original session
   ```

   ```
   doSomethingUsingRepository2(relatedSession);
   ```

Both techniques allow developers to use identities stored in the session manager. Users of versions earlier than version 6.0 of DFC must note that it is not recommended to use setDocbaseScope to create sub-connections because it is stateful and renders sessions, unshareable.

Caching objects in a session

DFC maintains a local cache of objects on the client to avoid calls to the server for each request. The DFC cache is scoped by session. An object is put in the cache when the client session accesses it the first time. Subsequent access to the same object in the application through the same session, is returned from the cache.

The local DFC cache is synchronized with the server in the following scenarios:

- Fetch/Checkout: If an object that is in the cache is fetched or checked out again, DFC requests the version stamp of the object from Content Server. If the version stamp of the object in the client-side cache does not match the version stamp returned by the server, DFC fetches the updated object and updates the cache accordingly.
- Save/Checkin: After a Save or Checkin operation, DFC does not update the local cache immediately. It marks the local cache as stale. DFC directly fetches the updated object and updates the cache for subsequent requests for the same object with the same session. A lazy cache update helps to improve the performance.

An updated or dirty object that is not yet saved in the repository, is not discarded. Updated objects are not visible to other sessions because the cache is maintained for each session, separately. You can modify the size of the cache using the dfc.cache.object.size property in the **dfc.properties** file based on the object size and memory , to avoid OutOfMemory conditions. The default value of dfc.cache.object.size, is 1000 objects.

DFC Session Pooling

DFC provides two levels of pooling. The first level is Level-1 pooling, while the second level is Level-2 or connection pooling.

Consider the following advantages of using DFC Session pooling:

- Since sessions are finite resources, pooling allows reuse of an existing session.
- Each session in DFC is a socket connection to the repository. Creating a socket connection is an expensive process. DFC keeps the open connection in a pool and uses it instead of creating and closing a socket connection for each request.
- Authentication is an expensive process. Therefore, DFC keeps the authenticated session in the pool and when a new session request is made for the same user and repository, DFC returns the session available in the pool.
- There is a limit on the number of open file descriptors in some operating systems. Since a socket consumes a file descriptor, pooling helps to manage resources effectively.

Level-1 pooling

In level-1 pooling, the session manager scopes the sessions in the pool. Every session manager has its local pool. After a user releases a session by calling the release() method of the session manager, the session is placed in the level-1 pooling.

Note: Sessions are stateful. A session encapsulates connection and maintains its state, such as object cache, until it is flushed. So, if a session in level-1 pooling is available for use, it indicates that the session is authenticated and ready for use.

When a user calls a getSession() or newSession() method, DFC returns the session from the pool, if pooling is enabled and a session is available in the pool. If the session is not available in the level-1 pool, DFC creates a new session and returns it to the user. That session is placed in the pool after the session is released. The first getSession() request retrieves the session from the pool and all subsequent getSession() requests for the same user and repository will return a reference to the same "shared session", that the first getSession() call returned.

Sessions in a level-1 pool are short-lived and monitored by a daemon thread called "Session pool worker" that flushes all expired sessions in to a level-2 pool based on the dfc.session.pool.expiration_interval DFC property. The default value of dfc.session.pool.expiration_interval is 5 seconds. Developers must ensure that they control the property based on the application performance requirement and deployment.

Consider a scenario where Web-based developers create a session manager for each http request and use level-1 pooling. This is an incorrect usage because level-1 pooling is scoped for every session manager but each request uses its own session manager. In addition, this approach introduces problems because requests to create a new session using a new session manager create a new socket connection and session. This approach does not reuse previous sessions created by the previous session manager. The sessions created by previous requests defined by the dfc.session.pool.expiration_interval property remain as they are, until the Session Worker Thread cleans up the sessions. As a result, the code may reach the maximum session limit when there is a heavy load of requests.

Level-2 or connection pooling

After a session from a level-1 pool times out, the session object is flushed and a connection encapsulated in a session is placed in the level-2 pool. Hence, it is called the connection pool. Here, connections are pooled globally. Unlike level-1 pooling, connection in the level-2 pool can be reused by any session manager to construct a new session object. If there are no connections in the level-2 pool, DFC creates a new connection, internally.

Level-2 pooling works only if the dfc.session.pool.mode pooling mode is set to level2. The dfc.session.pool.mode pooling mode is set to level2, by default.

If a free connection is available in the pool, DFC authenticates the connection if the user credentials of the session manager do not match, or if force authentication or periodic authentication are set to true.

Connections are moved in to the level-2 pool if dfc.session.reuse_limit has not exceeded. The default value of the dfc.session.reuse_limit property is 100. This property limit indicates the maximum number of times a session can be reused from the level-2 session pool. When the session reuse limit is reached, the session is disconnected and a new session is created to release server resources associated with the previous session. The dfc.session.reuse_limit preference helps prevent the repository server process from becoming unwieldy, since memory usage of the session gradually grows on the server every time a session is reused.

In DFC versions earlier than 6.5 SP1, we did not close connections in the pool. This was an optimization implemented to reuse the connection from the pool and to avoid unnecessary connection creation and closure, which are costly. If connections in the level-2 pool are not used for a long time, such connections are retained in the close_wait state because the server closes its side of the socket after the server session timeout is reached. This value is defined as client_session_timeout in the **server.ini** file. The default value of client_session_timeout is 5 minutes.

The netstat command run on the application server machine indicates the connections that are in the close_wait state.

Some customers have reported that connections in the close_wait state impact the scalability of their applications. So, in DFC 6.5 SP2 Patch 02, connections in the close_wait state are closed periodically based on the dfc.connection.unused_connection_timeout property in the **dfc.properties** file. The default value of the dfc.connection.unused_connection_timeout property is 5 minutes. The value of this preference must be approximately the same as the Content Server session timeout value. If DFC connects to multiple servers with different session timeouts, it is recommended to keep this value close to the average of the session time out values of the different servers.

Transaction Handling

The Documentum platform supports the following transactions:

- **Implicit transactions:** Content Server uses implicit transactions internally. The server opens and closes implicit transactions. Each operation that requires a database change, uses the implicit transaction. For example, save, checkin, checkout, and destroy operations are performed automatically in a server-defined transaction.
- **Explicit transactions:** DFC users define explicit transactions. Users can open, commit, or rollback transactions using DFC.

  Explicit transactions are supported by the session and session manager.

Sessions support transactions that are based on the underlying relational database transaction mechanism of the repository. Sessions cannot handle interactions with more than one repository. Prior to DFC version 6.0, transactions were supported only at the session level. DFC did not support transactions across repositories. However, from DFC version 6.0, the session manager transaction allows transactions across more than one repository.

The session manager keeps the session as long as the transaction remains open. In a session manager transaction, only sessions obtained after the call to the beginTransaction() method can participate in the transaction.

Caution: Session manager transactions do not use a two-phase commit algorithm. They rely on the transaction facilities of the underlying databases of the respective repository to which they are connected. As a result, a multi-repository transaction can fail occasionally, after the system has committed the transaction in one of the databases. Developers must be aware of this limitation when transactions span multiple repositories.

The IDfSession interface comprises the beginTrans(), commitTrans() and abortTrans() methods that support transactions at the session level. IDfSessionManager comprises the beginTransaction(), commitTransaction(), and abortTransaction() methods for handling explicit transactions.

The developer must start a transaction in try, catch, or finally block. If an exception occurs in the try block, the developer must call abortTrans in the catch block to abort the transaction.

It is recommended to use the isTransactionActive() method that is common for transaction handling at the session and session manager levels before starting a new transaction. It helps to know if there is any other active transaction that is open. Any operation such as save or checkin is not committed to the repository until the commitTrans() or commitTransaction() method is called.

From DFC version 6.0, if a thread, say t1, opens a transaction using a session, say s1, and if another thread, say t2, performs an activity such as executing a read query using the same session, the other thread, t2, is put on a wait state until the first thread, t1 commits the transaction. Although two different threads share the same session, they may not share the same transaction. This helps to maintain session cache consistency and is necessary because two threads do not have direct knowledge of each other.

Analyzing DFC session problems

You can employ one of the following options to resolve or diagnose the root cause of most session-related issues.

Detecting session or collection leaks

Developers must always release a session to make efficient use of the available sessions, resources associated with a session, and pooling. If a session is not released, a large number of sessions are created and eventually, DFC and Content Server will run out of available sessions. Therefore, it is important to find the source of session leaks.

Include the dfc.resources.diagnostics.enabled property and set it to true in the **dfc.properties** file to enable DFC session diagnostics. Session leaks in custom code are easily identified if this diagnostics property is enabled. A log message with complete stack trace is generated when some resources are garbage collected when they are not closed or released by the application. However, enabling this diagnostic property imposes a performance overhead. This property must not be enabled in the production system unless it is required to diagnose a critical issue.

The following diagnostic messages are displayed when this diagnostic property is set. Do not ignore these messages but use them to correct the issue in the client application:

- DFC_SESSION_NOT_RELEASED: This is an alert indicating that the client did not release a previously obtained session.
- DFC_QUERY_NOT_CLOSED: Indicates that the client code forgot to close a collection properly after use.
- DFC_SESSION_UNMATCHED_RELEASE: Indicates that the client code attempted to release an already released session. This is true only if the compatibility option is set.

Despite the above messages, DFC takes care of cleaning up the underlying sessions or disposable resources while running in the diagnostic mode. However, the developer must correct the application code where the issue originated, to ensure resources are managed properly. Correcting the application code will ensure all sessions are released and all collections are closed.

It is recommended to enable DFC resource diagnostics for development to reduce the risk of leaks going undetected until issues occur in the production system.

Dumping and verifying session information

Execute the list_sessions or show_sessions command using API or DQL.

Sample API

```
apply,c,NULL,LIST_SESSIONS,BRIEF_INFO,B,F
next,c,q0
dump,c,q0
```

This API provides a list of sessions and their status. You can view properties such as session state and the DFC host from which the session originated.

DM_API_E_NO_SESSION error

This API indicates that DFC is running out of sessions. This error occurs if sessions are not released properly or if all sessions are used concurrently. The developer can increase the maximum number of sessions allowed, by modifying the dfc.session.max_count property in the **dfc.properties** file. If you are connecting to a single repository, ensure that the maximum number of sessions in DFC is the same as or less than the maximum concurrent sessions defined in the concurrent_sessions property in the **server.ini** file. If you are connecting to multiple DFC client applications to the same Content Server, the number of dfc.session.max_count for all DFC instances must not exceed the maximum number of concurrent sessions.

The concurrent_sessions property must be defined considering the value of the maximum database processes allowed. Each Content Server session creates an average of two database connections. In addition, do not fail to maintain a count of methods, jobs, and so on because they also consume sessions on Content Server. If you have several workflows or Java methods running, the number of sessions on the JMS will impact the total session usage on Content Server.

DFC_SESSION_HANDLE_STALE or DFC_SESSION_HANDLE_RESTRICTION exception

This exception indicates that the code is using a deprecated session management implementation that exposes the developer to several session-related risks. It is highly recommended to modify the code to use the current functionality. If developers are unable to upgrade their code, DFC provides backward compatibility and the following options that developers can enable in the **dfc.properties** file to allow the older program to function:

- dfc.compatibility.allow_weak_disconnect
- dfc.compatibility.preserve_session_info_messages
- dfc.compatibility.allow_weak_setrepositoryscope

DFC application hangs on the Linux machine

If the client machine running the DFC client application is a Linux machine, and if the server to which it is connected goes off the network, the socket takes a very long time to timeout. Occasionally, it causes the application server or the DFC client application to stop responding. This problem is related to low-level operating system socket handling. The developer must change the operating system level networking kernel settings and enable the dfc.session.keepalive.enable property in the **dfc.properties** file to resolve this problem. It is also recommended that application administrators collaborate with server administrators before making the change. Server administrators must identity appropriate settings for the kernel parameters based on their performance requirement and deployment.

Operating system level settings for Linux:

- net.ipv4.tcp_keepalive_time=1
- net.ipv4.tcp_keepalive_intvl=5
- net.ipv4.tcp_keepalive_probes=3
- net.ipv4.tcp_retries2=3

Conclusion

This white paper covers several important aspects of session management. Incorrect DFC Session management can lead to performance problems and system outages or unexpected results. This document helps developers understand session handling and write better code. It also helps those who deploy a DFC-based application to provide correct deployment settings based on the application environment.

This paper is not a replacement for any existing DFC-related guide. However, this paper attempts to consolidate most of the information available in the referenced documents in this article.

References

- Documentum Foundation Classes Development Guide
- Web Development Kit Development Guide
- Content Server Fundamentals Guide
- Content Server Administration Guide