Alfresco JLAN Server

Programmers Guide

For Alfresco JLAN Server v6.0

Author: GK Spencer

© Alfresco 2007-2011. All rights reserved

Table of Contents

1 Alfresco JLAN Server Overview..... 6
1.1 Java Packages..... 8
2 The JLAN Server Applications..... 10
2.1 org.alfresco.jlan.app.JLANServer..... 10
2.2 org.alfresco.jlan.app.JLANServerService..... 10
2.3 JLAN Server XML Configuration File..... 11
3 Writing a DiskInterface Implementation..... 12
3.1 DiskInterface Methods..... 13
3.1.1 createContext..... 13
3.1.2 openFile..... 14
3.1.3 createFile..... 14
3.1.4 createDirectory..... 15
3.1.5 fileExists..... 15
3.1.6 getFileInformation..... 16
3.1.7 isReadOnly..... 16
3.1.8 startSearch..... 16
3.1.9 deleteFile..... 17
3.1.10 deleteDirectory..... 17
3.1.11 renameFile..... 17
3.1.12 setFileInformation..... 17
3.1.13 readFile..... 18
3.1.14 writeFile..... 18
3.1.15 seekFile..... 19
3.1.16 truncateFile..... 19
3.1.17 flushFile..... 19
3.1.18 closeFile..... 19
3.2 DiskDeviceContext..... 20
3.2.1 DiskDeviceContext Methods..... 21
3.2.1.1 Constructors..... 21
3.2.1.2 hasStateCache..... 21
3.2.1.3 getStateCache..... 21
3.2.1.4 hasVolumeInformation..... 21
3.2.1.5 getVolumeInformation..... 21
3.2.1.6 hasDiskInformation..... 21
3.2.1.7 getDiskInformation..... 21
3.2.1.8 setRequiresStateCache..... 22
3.2.1.9 setStateCache..... 22
3.2.1.10 enableChangeHandler..... 22
3.2.1.11 removeExpiredFileStates..... 22
3.2.1.12 CloseContext..... 22
3.2.1.13 hasChangeHandler..... 22
3.2.1.14 getChangeHandler..... 22
3.2.1.15 addNotifyRequest..... 22
3.2.1.16 removeNotifyRequest..... 22
3.2.1.17 setVolumeInformation..... 23
3.2.1.18 setDiskInformation..... 23
3.2.1.19 hasQuotaManager..... 23
3.2.1.20 getQuotaManager..... 23
3.2.1.21 setQuotaManager..... 23
3.2.1.22 startFilesystem..... 23
3.2.1.23 getDeviceAttributes..... 23
3.2.1.24 setDeviceAttributes..... 23
3.2.1.25 getFilesystemAttributes..... 24
3.2.1.26 setFilesystemAttributes..... 24

3.3 Optional Interfaces....25
3.3.1 DiskSizeInterface....25
3.3.2 DiskVolumeInterface....25
3.3.3 NTFSStreamsInterface....26
3.3.4 FileLockingInterface....26
3.3.5 OpLockInterface....26
3.3.6 FileIdInterface....26
3.3.7 IOCtlInterface....27
3.3.8 SymbolicLinkInterface....28
3.3.9 FTPSiteInterface....29
3.3.10 SecurityDescriptorInterface....29
3.3.11 TransactionalFilesystemInterface....29
3.4 Useful Utility Classes....31
3.4.1 WildCard Class....31
3.4.2 FileName Class....32
4 Database Filesystem Driver....33
4.1 Writing A Database Interface Implementation....34
4.1.1 Implementing The Main Database Interface....34
4.1.1.1 Filesystem Structure Details....35
4.1.1.2 DBInterface Methods....38
4.1.1.2.1 createFileRecord....38
4.1.1.2.2 createStreamRecord....38
4.1.1.2.3 deleteFileRecord....38
4.1.1.2.4 deleteStreamRecord....38
4.1.1.2.5 deleteSymbolicLink....38
4.1.1.2.6 fileExists....39
4.1.1.2.7 getDBInterfaceName....39
4.1.1.2.8 getFileId....39
4.1.1.2.9 getFileInformation....39
4.1.1.2.10 getFileRetentionDetails....40
4.1.1.2.11 getStreamInformation....40
4.1.1.2.12 getStreamsList....40
4.1.1.2.13 getUsedFileSpace....40
4.1.1.2.14 initializeDatabase....40
4.1.1.2.15 readSymbolicLink....41
4.1.1.2.16 renameFileRecord....41
4.1.1.2.17 renameStreamRecord....41
4.1.1.2.18 requestFeatures....41
4.1.1.2.19 setFileInformation....42
4.1.1.2.20 setStreamInformation....42
4.1.1.2.21 shutdownDatabase....43
4.1.1.2.22 startSearch....43
4.1.1.2.23 supportsFeatures....43
4.1.2 Implementing The Data Interface....44
4.1.2.1 Data Storage....44
4.1.2.2 DBDataInterface Methods....45
4.1.2.2.1 deleteFileData....45
4.1.2.2.2 deleteJarData....45
4.1.2.2.3 getFileDataDetails....45
4.1.2.2.4 getMaximumFragmentSize....45
4.1.2.2.5 loadFileData....46
4.1.2.2.6 loadJarData....46
4.1.2.2.7 saveFileData....46
4.1.2.2.8 saveJarData....46
4.1.3 Implementing the Queueing Interface....46
4.1.3.1 File Request Queue Storage....47
4.1.3.2 DBQueueInterface Methods....47

4.1.3.2.1 deleteFileRequest....47
4.1.3.2.2 loadFileRequest.... 48
4.1.3.2.3 loadTransactionRequest.... 48
4.1.3.2.4 performQueueCleanup....48
4.1.3.2.5 queueFileRequest....48
4.1.4 Implementing The Object Id Interface....48
4.1.4.1 Object Id Storage.... 49
4.1.4.2 DBObjectIdInterface Methods....49
4.1.4.2.1 deleteObjectId....49
4.1.4.2.2 loadObjectId.... 49
4.1.4.2.3 saveObjectId....49
4.2 Writing A File Loader Implementation.... 50
4.2.1 Implementing The File Loader Interface....50
4.2.1.1 FileLoader Methods.... 51
4.2.1.1.1 addFileProcessor....51
4.2.1.1.2 closeFile.... 51
4.2.1.1.3 deleteFile....51
4.2.1.1.4 getRequiredDBFeatures.... 52
4.2.1.1.5 initalizeLoader.... 52
4.2.1.1.6 openFile....52
4.2.1.1.7 queueFileRequest....53
4.2.1.1.8 shutdownLoader.... 53
4.2.1.1.9 supportsStreams....53
4.2.1.2 NamedFileLoader Methods.... 53
4.2.1.2.1 createDirectory....53
4.2.1.2.2 deleteDirectory....53
4.2.1.2.3 renameFileDirectory.... 53
4.2.1.2.4 setFileInformation....54
4.2.2 Extending The ObjectIdFileLoader....54
4.2.2.1 ObjectIdFileLoader Methods To Implement....54
4.2.2.1.1 loadFileData.... 55
4.2.2.1.2 saveFileData....55
5 Server Configuration.... 56
5.1 loadConfiguration....56
5.2 saveConfiguration.... 56
6 Cifs Authentication....57
6.1 UsersInterface.... 57
6.1.1 UsersInterface Methods....57
6.1.1.1 initializeUsers....57
6.1.1.2 getUserAccount.... 57
6.2 CifsAuthenticator.... 58
6.2.1 CifsAuthenticator Methods.... 58
6.2.1.1 getAuthContext.... 58
6.2.1.2 authenticateUser.... 59
6.2.1.3 authenticateShareConnect.... 59
7 Share Mapper.... 61
7.1 ShareMapper Methods.... 61
7.1.1 initializeMapper....61
7.1.2 getShareList.... 61
7.1.3 findShare.... 61
7.1.4 deleteShares.... 62
7.1.5 closeMapper.... 62
8 Access Control Manager.... 63
8.1 AccessControlManager Interface.... 63
8.1.1 initialize.... 63
8.1.2 checkAccessControl.... 63
8.1.3 filterShareList.... 63

8.1.4 addAccessControlType....64
8.1.5 createAccessControl....64
8.2 Access Control Rules....65
8.2.1 AccessControlParser Class....65
8.2.1.1 getType....65
8.2.1.2 createAccessControl....65
8.2.2 AccessControl Class....65
8.2.2.1 allowsAccess....65
9 Quota Manager....67
9.1 QuotaManager Interface....67
9.1.1 startManager....67
9.1.2 stopManager....67
9.1.3 allocateSpace....67
9.1.4 releaseSpace....67
9.1.5 getAvailableFreeSpace....67
9.1.6 getUserFreeSpace....68
10 ONC/RPC Authenticator....69
10.1 RpcAuthenticator Methods....69
10.1.1 initialize....69
10.1.2 getRpcAuthenticationTypes....69
10.1.3 authenticateRpcClient....70
10.1.4 getRpcClientInformation....70
11 FTP Authenticator....72
11.1 FTPAuthenticator Methods....72
11.1.1 initialize....72
11.1.2 authenticateUser....72
12 File State Cache and Clustering....73
12.1 File State Cache Basics....73
12.2 File Access Tokens....73
12.2.1 Grant File Access....74
12.2.2 Release Access Token....74
12.3 Using The File State Cache....75
12.3.1 DiskInterface Requirements....75
12.3.1.1 fileExists....75
12.3.1.2 getFileInformation....75
12.3.1.3 openFile....75
12.3.1.4 createFile/createDirectory....75
12.3.1.5 deleteFile/deleteDirectory....76
12.3.1.6 closeFile....76
12.3.1.7 renameFile....76
12.3.1.8 setFileInformation....76
12.3.1.9 startSearch....76
12.3.1.10 writeFile/truncateFile....76
12.3.1.11 readFile/flushFile/seekFile....77
12.3.1.12 getLockManager/getOplockManager....77
12.3.2 DiskDeviceContext Requirements....77
12.3.2.1 Initialization....77
12.3.2.2 startFilesystem....77
12.3.2.3 stateCacheInitializing....78
12.3.2.4 stateCacheRunning....78
12.3.2.5 stateCacheShuttingDown....78

Alfresco JLAN Server Programmers Guide

1 Alfresco JLAN Server Overview

The JLAN Server is a Java based file server implementing the Server Message Block (SMB) protocol, also known as the Common Internet File System (CIFS), File Transfer Protocol (FTP) and Network File Server (NFS) protocol.

SMB/CIFS is the protocol used by Windows networking to provide disk and print shares, plus other network administration and security functions.

The JLAN Server uses a virtual filesystem interface that provides a standard interface to the filesystem for the various protocols. The virtual filesystem may be mapped to a real filesystem, or other repository or media.

Much of the design philosophy behind the JLAN Server is about customization. Many of the key components of the system can be replaced via the main server configuration class. The key components that may be replaced/customized are:-

- Virtual filesystem driver classes
- Authentication classes
- Server configuration classes
- Virtual filesystem mapping class
- Access control manager and access control rules
- Quota manager

The JLAN Server kit contains a virtual filesystem driver class that maps to the local filesystem using the *java.io.File* class and a JDBC based filesystem that stores the filesystem structure in a database table with a custom file loader class used to load and save the file data. There are sample file loader implementations that use the local filesystem and database BLOB fields.

The default authentication class allows anyone to access the JLAN Server virtual fileysystems. The JLAN Server kit contains two sample authentication classes. The *org.alfresco.jlan.jlansrv.LocalAuthenticator* uses a user list defined in the server configuration to control access to the server. The *org.alfresco.jlan.jlansrv.PassthruAuthenticator* uses a domain controller or other server to validate users accessing the JLAN Server.

The sample server applications – *org.alfresco.jlan.app.JLANServer* and *org.alfresco.jlan.app.JLANServerService* – use an XML based server configuration implementation.

The default virtual filesystem mapping class provides access to the filesystems defined in the server configuration plus allows access to a HOME area if the user accessing the server has a home directory defined in the server configuration.

There are two approaches to implementing a virtual filesystem on the JLAN Server:-

- Implement the *org.alfresco.jlan.server.filesys.DiskInterface* interface and associated classes

- Extend one of the FileLoader classes and/or implement a custom database interface to be used by the database filesystem driver.

Implementing your own DiskInterface requires more work but gives you more control over the virtual filesystem.

The file loader approach is simpler as the core disk interface handles all of the filesystem structure using a database table. There are various file loader classes that can be extended from simple loaders that only need to know about the data to a multi-threaded loader that is capable of background load/save operations with built in local caching of file data.

This document details both approaches to implementing a virtual filesystem.

The document also has sections detailing how to implement your own server configuration, authentication, virtual filesystem mapping classes, access control manager and access control rules, and quota manager.

1.1 Java Packages

The following table lists the JLAN Server packages :-

Package	Description
org.alfresco.jlan.app	Main JLANServer applications and XML server configuration implementation
org.alfresco.jlan.debug	Debug global class, plus console, file and JDK logging debug output implementations
org.alfresco.jlan.debug.cluster	Cluster based debug implementation
org.alfresco.jlan.ftp	FTP server
org.alfresco.jlan.locking	File locking classes/interfaces
org.alfresco.jlan.netbios	NetBIOS over TCP/IP
org.alfresco.jlan.netbios.server	NetBIOS name server
org.alfresco.jlan.netbios.win32	Win32 NetBIOS native code interface
org.alfresco.jlan.oncrpc	ONC/RPC protocol implementation
org.alfresco.jlan.oncprc.mount	Mount server
org.alfresco.jlan.oncrpc.nfs	NFS server
org.alfresco.jlan.oncrpc.portmap	Portmapper server
org.alfresco.jlan.sample	Various sample components
org.alfresco.jlan.server	Network server base classes/interfaces
org.alfresco.jlan.server.auth	Network server base classes/interfaces
org.alfresco.jlan.server.auth.acl	Access control manager classes/interfaces
org.alfresco.jlan.server.auth.asn	ASN.1 encoding/decoding
org.alfresco.jlan.server.auth.kerberos	Kerberos support
org.alfresco.jlan.server.auth.ntlm	NTLM support
org.alfresco.jlan.server.auth.passthru	Passthru authentication
org.alfresco.jlan.server.auth.spnego	SPNEGO
org.alfresco.jlan.server.config	Server configuration base classes/interfaces
org.alfresco.jlan.server.core	Core server classes/interfaces
org.alfresco.jlan.server.filesys	File server base classes/interfaces
org.alfresco.jlan.server.filesys.cache	Filesystem caching classes
org.alfresco.jlan.server.filesys.cache.cluster	Clustered state cache base classes
org.alfresco.jlan.server.filesys.cache.cluster.hazelcast	Clustered file state cache implementation based on Hazlecast
org.alfresco.jlan.server.filesys.db	Database filesystem classes and interfaces
org.alfresco.jlan.server.filesys.db.derby	Derby database interface
org.alfresco.jlan.server.filesys.db.mysql	MySQL database interface
org.alfresco.jlan.server.filesys.db.oracle	Oracle database interface
org.alfresco.jlan.server.filesys.loader	File loader base classes/interaces
org.alfresco.jlan.server.filesys.pseudo	Pseudo file classes/interface

Package	Description
org.alfresco.jlan.server.filesys.quota	Per share quota manager classes/interfaces
org.alfresco.jlan.server.locking	Server file locking classes/interfaces
org.alfresco.jlan.server.memory	Memory pool
org.alfresco.jlan.server.thread	Thread pool
org.alfresco.jlan.smb	SMB protocol classes
org.alfresco.jlan.smb.dcerpc	DCE/RPC classes used by client and server code
org.alfresco.jlan.smb.dcerpc.info	DCE/RPC information classes that may be serialized to/from a DCE/RPC buffer
org.alfresco.jlan.smb.dcerpc.server	DCE/RPC server classes
org.alfresco.jlan.smb.mailslot	Mailslot protocol classes/interfaces. Includes the HostAnnouncer for announcing the SMB server to Network Neighborhood
org.alfresco.jlan.smb.mailslot.win32	Host announcer implementation that uses the Win32 NetBIOS native interface
org.alfresco.jlan.smb.nt	NT specific SMB data structures
org.alfresco.jlan.smb.server	Main SMB server classes/interfaces
org.alfresco.jlan.smb.server.disk	Java.io.File virtual filesystem implementation
org.alfresco.jlan.smb.server.nio	NIO based CIFS channels/sessions
org.alfresco.jlan.smb.server.nio.win32	NIO baed Winsock NetBIOS channels/sessions
org.alfresco.jlan.smb.server.notify	SMB asynchronous change notification classes
org.alfresco.jlan.smb.server.ntfs	JLAN Server NTFS streams classes/interfaces
org.alfresco.jlan.smb.server.win32	Win32 NetBIOS native code session and packet handlers
org.alfresco.jlan.test.cluster	CIFS cluster test suite
org.alfresco.jlan.smb.util	Drive mapping classes
org.alfresco.jlan.util	Utility classes for data packing/unpacking, hex dump, IP address parsing, memory size parsing, name/value pairs and wildcard processing
org.alfresco.jlan.util.db	Database connection pool utility class
org.alfresco.jlan.util.win32	Win32 specific native code for mapping/unmapping network drives

2 The JLAN Server Applications

The Jar file supplied with the JLAN Server kit contains two fully functional server applications that use the SMB/CIFS, NetBIOS, FTP and NFS server components:-

- *org.alfresco.jlan.app.JLANServer*
  Allows the JLAN Server to be started as a console application, or as an NT service.
- *org.alfresco.jlan.app.JLANServerService*
  Allows the JLAN Server to be started as a console application, or NT service, or Linux/Unix daemon by using the ServiceWrapper from TanukiSoftware.

The server is configured using an XML configuration file. The application uses the DOM based parser built into the Java JDK/runtime to parse the XML configuration file.

The configuration file defaults to *jlanserver.xml* in the user home directory, under Windows this will be in the *Documents And Settings*\<*username*> *or Users*\<*username*> directory. The configuration file can also be specified on the command line.

2.1 org.alfresco.jlan.app.JLANServer

The JLANServer application can be run as a console application or as an NT service. The following command lines show various ways that the server application can be started:-

```
java -jar alfresco-jlan.jar
```

```
java -cp .\alfresco-jlan.jar org.alfresco.jlan.app.JLANServer jlanconfig.xml
```

A sample configuration file is included in the demo kit (*jlanserver.xml*), this defaults to using the Win32 NetBIOS interface.

The *runsrv.bat* batch file may also be used to start the server under Windows.

2.2 org.alfresco.jlan.app.JLANServerService

The JLANServerService application uses the ServiceWrapper from TanukiSoftware (http://wrapper.tanukisoftware.org/) to provide portability and resilience.

The ServiceWrapper is available for a wide range of platforms, including Windows, Linux, Mac OS X, Irix, HP-UX, Aix, FreeBSD and Solaris.

The JLAN Server kit contains the binaries for Windows, Linux, Solaris and Mac OS X support in the *service*\ sub-directory. A pre-configured ServerWrapper configuration file is included – jlansrv.conf. The main JLAN Server configuration file is expected to be in the user home directory, the jlansrv.conf file only provides the ServiceWrapper configuration of the JVM, application class, logging, JVM monitoring and NT service parameters.

To start the JLANServerService under Windows use the JLANServer.exe in the *wrapper**windows* sub-directory. To start as a console application use the following command line:-

```
jlanserver -c jlansrv.conf
```

The ServiceWrapper can also be used to run the JLAN Server as an NT service or daemon process. To install and start the JLAN Server as an NT service use the following commands:-

```
jlanserver -i jlansrv.conf

jlanserver -t jlansrv.conf
```

The *wrapper.ntservice.account* and *wrapper.ntservice.password* parameters in the jlansrv.conf will need to be modified before installing the JLAN Server as an NT service.

To start the JLANServerService under Linux, Solaris or FreeBSD use the jlanserver application in the appropriate *wrapper*\ sub-directory.

A script is provided in the *service**linux*\ sub-directory that can be used to start/stop/restart the JLAN Server as a daemon process under Linux.

For more information on configuring the ServiceWrapper and to download support for other operating systems visit the TanukiSoftware web site at http://wrapper.tanukisoftware.org/.

2.3 JLAN Server XML Configuration File

The JLAN Server is configured using a simple XML file. A DTD is available in the kit to validate the configuration (*jlanserver.dtd*).

The configuration is contained within the <*jlanserver*> section of the configuration file. The server is configured via the <*servers*>, <*global*>, <*SMB*>, <*FTP*>, <*NFS*> <*shares*>, <*security*>, <*shareMapper*>, <*DriveMappings*>, <*cluster*> and <*debug*> sub-sections.

See the *JLAN Server Installation Guide* for detailed information on the format of the XML configuration file.

3 Writing a DiskInterface Implementation

The various core protocol servers (SMB/CIFS, FTP and NFS) communicate with each virtual filesystem via a DiskInterface implementation (in the *org.alfresco.jlan.server.filesys* package).

Each virtual filesystem is represented by a DiskSharedDevice object. The DiskSharedDevice has a name for the device, a DiskInterface implementation and a DiskDeviceContext. The device context allows the same driver class instance to be used for multiple virtual filesystems, any per filesystem information should be stored in the context.

The DiskInterface has the following methods:-

- createContext
- treeOpened
- treeClosed
- closeFile
- createDirectory
- createFile
- deleteDirectory
- deleteFile
- fileExists
- flushFile
- getFileInformation
- isReadOnly
- openFile
- readFile
- renameFile
- seekFile
- setFileInformation
- startSearch
- truncateFile
- writeFile

The createContext, treeOpened and treeClosed methods are from the DeviceInterface interface that the DiskDeviceInterface extends.

To hold the details of an open file the core server uses the NetworkFile class. This is an abstract class that requires you to implement the I/O methods for your particular

implementation.

When a directory search is performed by the core server it may take several network exchanges with the client to return all of the file/directory listing information. To hold the details of an in progress directory search the core server uses the SearchContext class. This is an abstract class that requires you to implement the methods to determine if there are more files to return, to return the file name or file information for the next file/directory and to allow a search to be restarted at a particular restart point.

The DeviceInterface and DeviceContext classes are in the org.alfresco.jlan.server.core package. The DiskInterface, DiskDeviceContext, NetworkFile and SearchContext classes and interfaces are in the org.alfresco.jlan.server.filesys package.

Most of the DiskInterface methods have SrvSession and TreeConnection parameter objects. The SrvSession object contains details of the network session connected to the server. The TreeConnection object contains the details of the virtual filesystem that the client is accessing. The DiskInterface can access the device context using the TreeConnection.getContext() method.

There are several optional interfaces that the virtual filesystem driver can implement to add support for dynamic disk size information, volume information, enable support for NTFS streams, enable file locking support and enable custom I/O controls. The disk size and volume information may also be set statically in the DiskDeviceContext object.

3.1 DiskInterface Methods

Let's take a closer look at each of the DiskInterface methods.

3.1.1 createContext

```
DeviceContext createContext(NameValueList args)
    throws DeviceContextException;
```

The createContext method is called after the DiskInterface instance has been created. The method is responsible for creating the device specific context using the supplied name/value arguments.

A DiskInterface must return a DiskDeviceContext based class.

The *args* parameter contains named values which are either String or NameValueList values.

If the required configuration values have not been supplied or are invalid a DeviceContextException should be thrown.

Example

The *org.alfresco.jlan.smb.server.disk.JavaFileDiskDriver* maps the virtual filesystem to the local filesystem using the *java.io.File* class. The driver requires the local path to be used as the root path for the virtual filesystem, this must be specified using the *LocalPath* argument.

```
public DeviceContext createContext(NameValueList args)
  throws DeviceContextException {

  // Get the device name argument

  NameValue path = args.findItem("LocalPath");
  if ( path != null)
    return new DiskDeviceContext(path.getValue());
```

```
  //  Required parameters not specified

  throw new DeviceContextException("LocalPath parameter not specified");
}
```

3.1.2 openFile

```
NetworkFile openFile(SrvSession sess, TreeConnection tree,
                     FileOpenParams params)
    throws IOException;
```

The openFile method is called to open an existing file or directory on the virtual filesystem.

The FileOpenParams object contains the file path relative to the virtual filesystem root and various file open options. If the virtual filesystem supports NTFS streams the FileOpenParams may contain details of a stream to open.

The method should validate that the file/directory exists and create a NetworkFile derived class to represent the open file.

If the file/directory is not accessible the method should throw an AccessDeniedException, if the file/directory does not exist a FileNotFoundException should be thrown.

The openFile method should set various attributes of the NetworkFile object:-

- File name
  NetworkFile.setName()
- Full path/name
  NetworkFile.setFullName()
- File size
  NetworkFile.setFileSize()
- File dates
  NetworkFile.setCreationDate()
  NetworkFile.setModifyDate()
- Directory flag
  NetworkFile.setAttributes()
- Set the granted file access
  NetworkFile.setGrantedAccess()

3.1.3 createFile

```
NetworkFile createFile(SrvSession sess, TreeConnection tree,
                       FileOpenParams params)
    throws IOException;
```

The createFile method is called to create a new file on the virtual filesystem.

The FileOpenParams object contains the file path relative to the virtual filesystem root and various file open options. If the virtual filesystem supports NTFS streams the FileOpenParams may contain details of a stream to create.

The method should validate that the file does not exist and create a NetworkFile derived class to represent the new file.

If the file already exists the method should throw a FileExistsException, if the file cannot be created the method should throw an AccessDeniedException.

The createFile method should set various attributes of the NetworkFile object:-

- File name
  NetworkFile.setName()
- Full path/name
  NetworkFile.setFullName()
- File size
  NetworkFile.setFileSize()
- File dates
  NetworkFile.setCreationDate()
  NetworkFile.setModifyDate()
- Directory flag
  NetworkFile.setAttributes()
- Set the granted file access
  NetworkFile.setGrantedAccess()

3.1.4 createDirectory

```
void createDirectory(SrvSession sess, TreeConnection tree,
                     FileOpenParams params);
```

The createDirectory method is called to create a new directory on the virtual filesystem.

The FileOpenParams object contains the file path relative to the virtual filesystem root and various file open options.

If the user does not have the privilege to create the new directory an AccessDeniedException should be thrown, if the directory already exists a FileExistsException should be thrown.

3.1.5 fileExists

```
int fileExists(SrvSession sess, TreeConnection tree, String path);
```

The fileExists method is used to check if a path exists, the status returned should also indicate whether the path refers to a file or directory.

The method should return one of the constants:-

FileStatus.NotExist

FileStatus.FileExists

FileStatus.DirectoryExists

3.1.6 getFileInformation

```
FileInfo getFileInformation(SMBSrvSession sess, TreeConnection tree,
                            String path)
    throws IOException;
```

The getFileInformation method is used to return the details about a particular file or directory.

The returned file information object should have the following details set:-

- File name
  FileInfo.setShortName() or via constructor
- File size
  FileInfo.setSize() or via constructor
- File attributes
  FileInfo.setAttributes() using constants from the FileAttribute class.
- Modify date
  FileInfo.setModifyDate()
- File id
  FileInfo.setFileId()

If the path is not a valid file or directory the method should throw a FileNotFoundException.

3.1.7 isReadOnly

```
boolean isReadOnly(SrvSession sess, DeviceContext ctx)
    throws IOException;
```

The isReadOnly method is used to check if the virtual filesystem is writeable.

This method is currently called when the virtual filesystem device is created so the *sess* parameter will be null.

3.1.8 startSearch

```
SearchContext startSearch(SrvSession sess, TreeConnection tree, String
                          searchPath, int attrib)
    throws FileNotFoundException;
```

The startSearch method is used to search a directory for files and sub-directories that match the specified search path and file attributes.

The search path may contain wildcard characters such as '*' to match zero, one or more characters and '?' to match any single character.

The file attributes parameter specifies file attribute flags, from the FileAttribute class, to determine the files and directories to include in the search.

The *org.alfresco.jlan.util.WildCard* class can be used to filter file names according to a wildcard search path.

If the search path does not contain wildcard characters the search is for a single file or sub-directory. If the file or sub-directory does not exist a FileNotFoundException should be

thrown.

The SearchContext object that is returned must contain all of the context required to return file names or file information for the search. The list of files and directories may be returned over a number of network requests from the client.

The SearchContext should be optimized to return file names or information sequentially. Although a resume id is returned to the client it has not been observed to be used by any Windows or other SMB/CIFS clients.

3.1.9 deleteFile

```
void deleteFile(SrvSession sess, TreeConnection tree, String path)
    throws IOException;
```

The deleteFile method is used to permanently delete a file from the virtual filesystem.

If the user does not have the privilege to delete the file or the file cannot be deleted for another reason then an AccessDeniedException should be thrown. If the file to be deleted does not exist a FileNotFoundException should be thrown.

3.1.10 deleteDirectory

```
void deleteDirectory(SrvSession sess, TreeConnection tree, String path)
    throws IOException;
```

The deleteDirectory method is used to permanently delete a directory from the virtual filesystem.

If the user does not have the privilege to delete the directory or the directory is not empty an AccessDeniedException should be thrown.

3.1.11 renameFile

```
void renameFile(SrvSession sess, TreeConnection tree, String oldName,
                String newName)
    throws IOException;
```

The renameFile method is used to rename a file or directory within the virtual filesystem.

The *oldName* and *newName* parameters are both paths relative to the root of the virtual filesystem. A file may be renamed from one directory to another.

If the source file or directory does not exist a FileNotFoundException should be thrown. If the destination file or directory already exists a FileExistsException should be thrown.

3.1.12 setFileInformation

```
void setFileInformation(SrvSession sess, TreeConnection tree, String path,
                        FileInfo info)
    throws IOException;
```

The setFileInformation method is used to update file information for a file or directory within the virtual filesystem.

File/directory creation, access and/or modify date/times may be modified along with the file attributes. File attributes constants are defined in the FileAttribute class.

To determine which file information fields are valid and should be set the FileInfo object

contains a set of setter flags. The flags can be checked using the FileInfo.hasSetFlag(int) method. The available fields that can be set are detailed in the table below.

Setter Flag	Description	Value Returned By
SetFileSize	Set the file size.	getSize()
SetAllocationSize	Set the file allocation size.	getAllocationSize()
SetAttributes	Set the file attributes.	getFileAttribtues()
SetModifyDate	Set the file modification date/time.	getModifyDateTime()
SetCreationDate	Set the file creation date/time.	getCreationDateTime()
SetAccessDate	Set the file access date/time.	getAccessDateTime()
SetChangeDate	Set the file change date/time.	getChangeDateTime()
SetGid	Set the owner group id.	getGid()
SetUid	Set owner user id.	getUid()
SetMode	Set the file mode.	getMode()
SetDeleteOnClose	Set the delete on close flag.	hasDeleteOnClose()

If the file/directory path is invalid the method should throw a FileNotFoundException.

If the file information cannot be set the method should throw an AccessDeniedException.

3.1.13 readFile

```
int readFile(SrvSession sess, TreeConnection tree, NetworkFile file,
             byte[] buf, int bufPos, int siz, long filePos)
     throws IOException;
```

The readFile method is used to read a block of data from a file previously opened using the openFile or createFile method.

The file position to start the read is specified as a 64bit value to support large files.

The method returns the actual length of data read which should be less than or equal to the requested read size.

If the file cannot be read due to the file being unavailable the method can throw a FileOfflineException.

3.1.14 writeFile

```
int writeFile(SrvSession sess, TreeConnection tree, NetworkFile file,
              byte[] buf, int bufoff, int siz, long fileoff)
     throws IOException;
```

The writeFile method is used to write a block of data to a file previously opened using the openFile or createFile method.

The file position to start the write request is specified as a 64bit value to support large files.

Write requests may be issued that are passed the current end of file, in this case the file must be extended by padding if necessary.

The write method should call the NetworkFile.incrementWriteCount() method. The write count is used by the JDBCDiskDriver/ThreadedFileLoader to determine if a cached file has changed and requires saving.

The method returns the actual length of data written.

3.1.15 seekFile

```
long seekFile(SrvSession sess, TreeConnection tree, NetworkFile file,
              long pos, int typ)
    throws IOException;
```

The seekFile method is used to position the file pointer within a file.

This method does not tend to be used as the readFile and writeFile methods specify the file offset within the request parameters.

The file offset is specified as a 64bit file offset to support large files. The value may be a negative offset.

The typ parameter specifies where the seek is relative to:-

SeekType.StartOfFile

SeekType.CurrentPos

SeekType.EndOfFile

3.1.16 truncateFile

```
void truncateFile(SrvSession sess, TreeConnection tree, NetworkFile file,
                  long siz)
    throws IOException;
```

The truncateFile method is used to resize a file to the specified size.

The file size is specified as a 64bit value to support large files.

3.1.17 flushFile

```
void flushFile(SrvSession sess, TreeConnection tree, NetworkFile file)
    throws IOException;
```

The flushFile method is used to flush any buffered data for the specified file.

3.1.18 closeFile

```
void closeFile(SrvSession sess, TreeConnection tree, NetworkFile file)
    throws IOException;
```

The closeFile method is used to close the specified network file and release all resources.

3.2 DiskDeviceContext

The DiskDeviceContext class is used to hold per shared filesystem variables, the class is normally extended to add your own instance variables.

The extended DiskDeviceContext based class should be created via the DiskInterface.createContext() method.

The DiskDeviceContext contains a number of objects/values that are either required or most shared filesystem implementations will find useful. The following objects/values are available :-

- *Device name*
  The value can be set via the constructor or the setDeviceName() method. May be a physical device name, filesystem path or other value used by the filesystem driver.
- *Configuration parameters*
  The NameValueList configuration parameters passed to the DiskInterface.createContext() method are stored to allow the ServerConfiguration implementation to save the values, if required.
- *File State Cache*
  A file state cache can be used to cache file detail information to help speed up the filesystem driver, and also allow the JLAN Server to be run in a clustered configuration.
  The DBDiskDriver uses a file state cache to reduce the amount of requests to the database.
- *Notify Change Handler*
  If change notifications are enabled on the shared filesystem the notify change handler will be created.
- *Volume Information*
  Static volume information object.
- *Disk Information*
  Static disk information object.
- *Quota Manager*
  If per share quota management is enabled the quota manager object will be set.
  The DiskInterface implementation is responsible for setting the quota manager object in either the DiskInterface.createContext() method or DiskDeviceContext.startFilesystem() method.
- *Filesystem Attributes*
  Set of flags that enable various advanced filesystem features. The *org.alfresco.jlan.server.filesys.FileSystem* class defines the available values.
- *Device Attributes*
  Set of flags that specify the device type that the filesystem resides on. The *org.alfresco.jlan.server.filesys.DeviceAttribute* class defines the available values.

3.2.1 DiskDeviceContext Methods

The following section details the DiskDeviceContext methods.

3.2.1.1 Constructors

```
void DiskDeviceContext();

void DiskDeviceContext(String devName);
```

The device name value is not required to be set, the value is only for use by the filesystem driver implementation.

3.2.1.2 hasStateCache

```
boolean hasStateCache();
```

Returns *true* if the file state cache is enabled, else *false*.

3.2.1.3 getStateCache

```
FileStateCache getStateCache();
```

Return the file state cache object. The object is based on the *org.alfresco.jlan.server.filesys.cache.FileStateCache class*.

3.2.1.4 hasVolumeInformation

```
boolean hasVolumeInformation();
```

Return *true* if the static volume information is set, else *false*.

The filesystem driver may implement the DiskVolumeInterface optional interface to return the volume information.

3.2.1.5 getVolumeInformation

```
VolumeInfo getVolumeInformation();
```

Return the static volume information.

3.2.1.6 hasDiskInformation

```
boolean hasDiskInformation();
```

Return *true* if the static disk information is available, else *false*.

The filesystem driver may implement the DiskSizeInterface optional interface to return the disk information.

3.2.1.7 getDiskInformation

```
SrvDiskInfo getDiskInformation();
```

Return the static disk information.

3.2.1.8 setRequiresStateCache

```
void setRequiresStateCache(boolean req);
```

Indicates that this filesystem requires a state cache if *req* is true.

3.2.1.9 setStateCache

```
Void setStateCache(FileStateCache stateCache);
```

Sets, or clears, the file state cache implementation used by this filesystem.

3.2.1.10 enableChangeHandler

```
void enableChangeHandler(boolean ena);
```

Enables, and creates, the change notification handler that is used to send out asynchronous change notifications to SMB/CIFS clients.

3.2.1.11 removeExpiredFileStates

```
int removeExpiredFileStates();
```

Scan the file state cache for expired file states and return the count of file states that were expired.

3.2.1.12 CloseContext

```
void CloseContext();
```

Close the disk device context and release resources. This method should be overridden to release any filesystem specific resources.

If you override this method with your own then the method must call *super.CloseContext()*.

3.2.1.13 hasChangeHandler

```
boolean hasChangeHandler();
```

Return *true* if the change notification handler is enabled, else *false*.

3.2.1.14 getChangeHandler

```
NotifyChangeHandler getChangeHandler();
```

Return the change notification handler object.

3.2.1.15 addNotifyRequest

```
void addNotifyRequest(NotifyRequest req);
```

Adds a change notification request to the change notification handlers queue of clients.

3.2.1.16 removeNotifyRequest

```
void removeNotifyRequest(NotifyRequest req);
```

Remove a change notification request from the change notification handler.

3.2.1.17 setVolumeInformation

```
void setVolumeInformation(VolumeInfo info);
```

Set the static volume information.

3.2.1.18 setDiskInformation

```
void setDiskInformation(SrvDiskInfo info);
```

Set the static disk information.

3.2.1.19 hasQuotaManager

```
boolean hasQuotaManager();
```

Return *true* if the quota manager is enabled, else *false*.

3.2.1.20 getQuotaManager

```
QuotaManager getQuotaManager();
```

Return the configured quota manager for the shared filesystem, or null if quota management is not enabled.

3.2.1.21 setQuotaManager

```
void setQuotaManager(QuotaManager quotaMgr);
```

Set the quota manager for the shared filesystem. The quota manager must implement the *org.alfresco.jlan.server.filesys.quota.QuotaManager* interface.

3.2.1.22 startFilesystem

```
void startFilesystem(DiskSharedDevice share);
```

The *startFilesystem()* method is called after the shared filesystem driver has been initialized and the access control list has been set.

3.2.1.23 getDeviceAttributes

```
int getDeviceAttributes();
```

Return the device attributes.

3.2.1.24 setDeviceAttributes

```
void setDeviceAttributes(int attrib);
```

Set the device attributes.

The *org.alfresco.jlan.server.filesys.DeviceAttirbute* class contains the available values, and they are listed here :-

Device Attribute	Description
Removable	Device supports removable media

Device Attribute	Description
ReadOnly	Read only device
FloppyDisk	Floppy disk type device
WriteOnce	Device supports write-once media
Remote	Device is remote
Mounted	Filesystem is mounted on the device
Virtual	Volume is virtual

3.2.1.25 getFilesystemAttributes

```
int getFilesystemAttributes();
```

Return the filesystem attributes.

3.2.1.26 setFilesystemAttributes

```
void setFilesystemAttributes(int attrib);
```

Set the filesystem attributes.

The *org.alfresco.jlan.server.filesys.FileSystem* class contains the available values, and are listed here :-

Filesystem Attribute	Description
CaseSensitiveSearch	Supports case sensitive file names
CasePreservedNames	Fie system preserves the case of file names
UnicodeOnDisk	Supports Unicode characters in file names
PersistentACLs	File system preserves and enforces ACLs
FileCompression	Supports file based compression
VolumeQuotas	Supports disk quotas
SparseFiles	Supports spares files
ReparsePoints	Supports reparse points
RemoteStorage	
VolumeIsCompressed	Volume is compressed
ObjectIds	Supports object identifiers
Encryption	Supports the Encrypted File System (EFS)

3.3 Optional Interfaces

There are a number of optional interfaces that the virtual filesystem driver can implement to provide extended functionality or dynamic information:-

- DiskSizeInterface
- DiskVolumeInterface
- NTFSStreamsInterface
- FileLockingInterface
- **OpLockInterface**
- FileIdInterface
- IOCtlInterface
- SymbolicLinkInterface
- FTPSiteInterface
- **SecurityDescriptorInterface**
- TransactionalFilesystemInterface

3.3.1 DiskSizeInterface

The DiskDeviceContext class contains a disk size information object that can be set statically via the setDiskInformation() method, and disk volume information that can be set via the setVolumeInformation() method.

The DiskSizeInterface has a single method to implement:-

```
void getDiskInformation(DiskDeviceContext ctx, SrvDiskInfo diskInfo)
     throws IOException;
```

The diskInfo object should be filled in with the block size, blocks per allocation unit and disk total/free allocation units:-

- SrvDiskInfo.setBlockSize()
- SrvDiskInfo.setBlocksPerAllocationUnit()
- SrvDiskInfo.setTotalUnits()
- SrvDiskInfo.setFreeUnits()

The block size is usually set to 512 bytes.

3.3.2 DiskVolumeInterface

The DiskVolumeInterface has a single method to implement:-

```
VolumeInfo getVolumeInformation(DiskDeviceContext ctx);
```

The returned VolumeInfo should have a valid volume label String. The serial number and volume creation date values are optional.

3.3.3 NTFSStreamsInterface

The NTFSStreamsInterface has the following methods to implement:-

```
boolean hasStreamsEnabled(SrvSession sess, TreeConnection tree);
```

Determines whether NTFS streams support is enabled. Allows NTFS streams support to be disabled at runtime, and/or on a per virtual filesystem basis.

If NTFS streams support is enabled the NT protocol handler will return the virtual filesystem type as 'NTFS', if not enabled or the virtual filesystem does not implement the NTFSStreamsInterface the default filesystem type returned is 'FAT32'.

```
StreamInfo getStreamInformation(SrvSession sess, TreeConnection tree,
                  StreamInfo streamInfo);
```

Return NTFS stream information for the specified stream.

```
StreamInfoList getStreamList(SrvSession sess, TreeConnection tree,
                  String fileName);
```

Return a list of streams for the specified top level file.

```
void renameStream(SrvSession sess, TreeConnection tree, String oldName,
                  String newName, boolean overWrite);
```

Rename a stream.

3.3.4 FileLockingInterface

The FileLockingInterface has a single method to implement:-

```
LockManager getLockManager(SrvSession sess, TreeConnection tree);
```

The returned *org.alfresco.jlan.server.locking.LockManager* implementation is responsible for providing the actual file lock/unlock functionality.

A LockManager implementation is provided that uses the FileStateCache – *org.alfresco.jlan.smb.server.cache.FileStateLockManager*. The FileStateLockManager is used by the JDBCDiskDriver virtual filesystem to provide file locking support.

3.3.5 OpLockInterface

The OpLockInterface has the following methods to implement:-

```
OpLockManager getOpLockManager(SrvSession sess, TreeConnection tree);
```

The returned *org.alfresco.jlan.server.locking.OpLockManager* implementation is responsible for the providing the actual oplock functionality.

```
Boolean isOpLocksEnabled(SrvSession sess, TreeConnection tree);
```

Determines whether the oplocks support is enabled. Allows oplock support to be disabled at runtime, and/or on a per virtual filesystem basis.

3.3.6 FileIdInterface

The FileIdInterface has a single method to implement:-

```
String buildPathForFileId(SrvSession sess, TreeConnection tree, int dirId,
                          int fileId)
    throws FileNotFoundException;
```

This interface is currently only used by the NFS server code to convert a file id back to a filesystem relative path. The file id and directory id are encoded into the NFS file handle.

3.3.7 IOCtlInterface

The IOCtlInterface has a single method to implement :-

```
DataBuffer processIOControl(SrvSession sess, TreeConnection tree,
                            int ctrlCode, int fid, DataBuffer dataBuf,
                            boolean isFSCtrl, int filter)
          throws IOControlNotImplementedException, SMBException;
```

Process an I/O control request.

The *ctrlCode* parameter contains an encoded value with fields for the function code, device type, access type and method. The *org.alfresco.jlan.smb.nt.NTIOCtl* class contains the following static methods for extracting the various fields :-

- NTIOCtl.getFunctionCode(int)
- NTIOCtl.getDeviceType(int)
- NTIOCtl.getAccessType(int)
- NTIOCtl.getMethod(int)

The *dataBuf* parameter contains the raw data for the request. The DataBuffer object has methods for unpacking various data types such as int, long and String. The *isFSCtrl* parameter indicates whether the I/O control is a filesystem or device control.

The NTIOCtl class contains constants for device types (Device... values), filesystem function codes (FsCtl... values), access types (Access... values) and method types (Method... values).

The method may return a DataBuffer with the response data or null if there is no data to be returned to the client.

The device type value will usually have the value *NTIOCtl.DeviceFileSystem*.

In order to have a Windows client issue some of the predefined I/O control codes various filesystem attributes must be enabled via the *DiskDeviceContext.setFilesystemAttributes(int)* method. This should be called during the device context initialization.

The available filesystem attribute bits are defined in the *org.alfresco.jlan.server.filesys.FileSystem* class, the following values are defined :-

- FileSystem.CaseSensistiveSearch
- FileSystem.CasePreservedNames
- FileSystem.UnicodeOnDisk
- FileSystem.PersistentACLs
- FileSystem.FileCompression
  Must be enabled to receive the NTIOCtl.FsCtlGetCompression and

NTIOCtl.FsCtl.SetCompression I/O control requests.

This also enables the file compression checkbox on the Advanced Attributes dialog of Windows Explorer.

- FileSystem.VolumeQuotas
- FileSystem.SparseFiles
- FileSystem.ReparsePoints
- FileSystem.RemoteStorage
- FileSystem.VolumeIsCompressed
- FileSystem.ObjectIds
- FileSystem.Encryption
  Must be enabled to receive the NTIOCtl.FsCtlSetEncryption I/O control code.

  This also enables the encryption checkbox on the Advanced Attributes dialog of Windows Explorer.

3.3.8 SymbolicLinkInterface

The SymbolicLinkInterface has a single method to implement :-

```
String readSymbolicLink( SrvSession sess, TreeConnection tree, String path)
        throws AccessDeniedException;
```

Return the symbolic link data for the symbolic link at the specified path.

By implementing the *SymbolLinkInterface* a filesystem driver enables symbolic links within the NFS server.

Symbolic links are created by using the *DiskInterface createFile()* method. The *FileOpenParams* object has *isSymbolicLink()* and *getSymbolicLinkName()* methods.

Directory searches can return information for a symbolic link using the standard *FileInfo* object. The *FileInfo* class has been extended to add a file type field, with file types defined in the *org.alfresco.jlan.server.filesys.FileType* class, and listed here :-

- FileType.RegularFile
- FileType.Directory
- FileType.SymbolicLink
- FileType.HardLink
- FileType.Device

FileType.HardLink and FileType.Device are for future use.

The file type is returned in the search by using the *FileInfo.setFileType(int)* method, and can be checked using the *FileInfo.isFileType()* method.

The *DiskInterface.getFileInformation()* method can also return details of a symbolic link using the FileInfo object in the same way as directory searches.

3.3.9 FTPSiteInterface

The FTPSiteInterface has two methods to implement :-

```
void initializeSiteInterface(ServerConfiguration config,
                NameValueList params)
```

Initialize the FTP server site specific command interface.

```
void processFTPSiteCommand( FTPSrvSession sess, FTPRequest req)
     throws IOException;
```

Process an FTP SITE sub-command. The *req* parameter contains the sub-command name and any additional arguments.

The FTPSiteInterface allows site specific commands to be implemented that extend the standard FTP server commands. The FTP protocol has the SITE command which allows new commands to be implemented.

The FTP server will pass the processing of the SITE command to the FTPSiteInterface which must respond using one of the FTPSrvSession.sendFTPResponse() methods.

3.3.10 SecurityDescriptorInterface

The SecurityDescriptorInterface has the following methods to implement:-

```
int getSecurityDescriptorLength(SrvSession sess, TreeConnection tree,
                            NetworkFile file);
```

Return the security descriptor length, in bytes, for the specified file.

```
SecurityDescriptor loadSecurityDescriptor(SrvSession sess,
                                    TreeConnection tree, NetorkFile file);
```

Return the security descriptor for the specified file.

```
Void saveSecurityDescriptor(SrvSession sess,
                                    TreeConnection tree, NetworkFile file);
```

Set the security descriptor for the specified file.

The SecurityDescriptorInterface allows a filesystem to take control of the security descriptor associated with a file. If the filesystem does not implement this interface a default SecurityDescriptor that allows full access to the file to the Everyone group is returned by the CIFS server.

This interface is of limited use without the implementation of the LSA and SAMR DCE/RPC services as these are required for a client to be able to convert security ids to/from their names.

3.3.11 TransactionalFilesystemInterface

The TransactionalFilesystemInterface has the following methods to implement :-

```
void beginReadTransaction(SrvSession sess);
```

Start a read-only transaction.

```
void beginWriteTransaction(SrvSession sess);
```

Start a writable transaction.

```
void endTransaction(SrvSession sess, ThreadLocal<Object> tx);
```

The TransactionalFilesystemInterface allows multiple filesystem driver calls to be wrapped by a single transaction. The filesystem driver can begin a read-only or writable transaction and store the transaction state in the SrvSession object. When the protocol specific request has finished making calls to the filesystem driver the TransactionalFilesystemInterface stored in the session will be called to complete the transaction.

To store and clear transaction state objects on the SrvSession the following methods are available :-

```
void initializeTransactionObject();
```

Allocates the thread local object that is used to store the transaction state.

```
ThreadLocal<Object> getTransactionObject();
```

Returns the thread local object used to store the transaction state. Use the get() method on the ThreadLocal<Object> to retrieve the transaction state object.

```
void setTransaction(ThreadLocal<Object> tx,
                    TransactionalFilesystemInterface iface);

void setTransaction(TransactionalFilesystemInterface iface);
```

Stores the transaction interface, and transaction state, in the SrvSession.

```
void clearTransaction();
```

Clears the transaction state and interface that were stored in the SrvSession.

3.4 Useful Utility Classes

There are a number of utility classes that may be useful when writing a DiskInterface implementation:-

- WildCard
  Implements the SMB wildcard processing algorithm for the multiple character wildcard (*) and single character wildcard (?).
- FileName
  Provides methods for splitting the SMB/CIFS format paths that are used throughout the JLAN Server.

3.4.1 WildCard Class

The *WildCard* class is contained in the *org.alfresco.jlan.util* package.

A WildCard class contains a number of static methods that can be used to check for wildcard characters within a search string:-

- containsWildcards(String)
- containsUnicodeWildcard(String)
- convertUnicodeWildcardToDOS(String)
- convertToRegexp(String)

The core JLAN Server code converts Unicode wildcards to the standard DOS wildcards (*/?) before passing search strings to the DiskInterface implementation.

The WildCard class requires a search pattern containing wildcard characters and a flag to indicate if the search is case sensitive. The WildCard class can be initialized via the constructor *WildCard(String pattern, boolean caseSensitive),* or you can use the *setPattern(String pattern, boolean caseSensitive)* method after creating the WildCard object.

To check if a path matches the wildcard search pattern use the *boolean matchesPattern(String path)* method.

Other useful methods of the WildCard class:-

`int isType()`

Returns the wildcard type:-

WILDCARD_NONE	No wildcard characters in search pattern
WILDCARD_ALL	'*.*' or '*' pattern
WILDCARD_NAME	'*.ext' type pattern
WILDCARD_EXT	'name.*' type pattern
WILDCARD_COMPLEX	Complex wildcard pattern
WILDCARD_INVALID	

Invalid or uninitialized search pattern

`boolean isCaseSensitive()`

Returns *true* if the search is case sensitive, else *false*.

`String toString()`

Returns the wildcard details as a String in the format:-

[<pattern>,<type>,<matchPart>,Case|NoCase]

3.4.2 FileName Class

The *FileName* class is contained in the *org.alfresco.jlan.smb.server* package.

The FileName class contains a number of static methods for manipulating SMB/CIFS format virtual filesystem relative paths.

`String[] FileName.splitPath(String path)`

`String[] FileName.splitPath(String path, char seperator)`

Splits the specified path into an array of two Strings, the element [0] is the path and element [1] is the file name from the end of the path. The file name element may be null.

The *FileName.splitPath(String path)* method uses the default seperator of '\'.

`String[] FileName.splitAllPaths(String path)`

Splits the specified path into a variable length array of Strings. The path string is split using the default seperator of '\'.

`String removeFileName(String path)`

Remove the file name from the end of the path string if there is a path seperator within the *path*.

`String convertSeperators(String path, char sep)`

Convert the standard seperator characters in the path ('\') to the specified seperator character.

`String buildPath(String dev, String path, String filename, char sep)`

Build a path string using the specified component strings. The *dev*, *path* and /or *filename* components may be null.

`String mapPath(String base, String path)`

Maps a path to a real file/directory on the local filesystem. The mapping may require changing the case of various parts of the path.

4 Database Filesystem Driver

The database filesystem driver (*org.alfresco.jlan.server.filesys.db.DBDiskDriver*) is a filesystem driver that implements the base DiskInterface plus the optional DiskSizeInterface, DiskVolumeInterface, NTFSStreamsInterface, FileLockingInterface, and FileIdInterface interfaces.

The database filesystem driver requires a database interface implementation class to provide the interface to the actual database or repository to store the filesystem structure details, and a file loader implementation to load and save the file data.

All database access is handled through a number of interfaces, the implementation does not have to use JDBC, SQL or relational database access.

The *org.alfresco.jlan.server.filesys.db.DBInterface* interface provides the basic database interface to the database filesystem driver to store the filesystem structure in a database and perform searches of the filesystem. The base database interface also provides NTFS streams and data retention support.

There are a number of optional interfaces that the database interface class can implement to provide additional features which may be required by the file loader class, such as loading/saving file data to a database BLOB field or providing JLAN filesystem id to repository object id mappings.

The file loader implementation may provide direct access to the file data or may load/save the file data to a temporary area. A number of utility classes are available that provide background loading/saving of file data with queueing of load/save requests.

The following table lists the database interfaces provided in the standard JLAN Server kit.

Database	Class
MySQL	org.alfresco.jlan.server.filesys.db.mysql.MySQLDBInterface
Oracle	org.alfresco.jlan.server.filesys.db.oracle.OracleDBInterface
Derby	org.alfresco.jlan.server.filesys.db.derby.DerbyDBInterface

The following file loader implementations are provided in the JLAN Server kit.

Name	Description
SimpleFileLoader	Provides a simple file loader that loads/saves files to the local filesystem. The file loader maintains the directory structure. This loader does not have any database requirements.
DBFileLoader	Loads/saves file data to database BLOB fields using a thread pool of worker threads to load/save the file data in background. A queue of load/save requests is maintained in a database table for crash recovery. This loader requires that the database interface provide additional database interfaces for load/save request queueing and file data loading/saving.
ObjectIdFileLoader	Loads/saves file data to a repository by providing a mapping from the database filesystem file id to a repository object id.

Name	Description
	The loader provides background load/save support and caches file data locally. This is an abstract class, the loadFileData() and saveFileData() methods must be implemented.

4.1 Writing A Database Interface Implementation

The main database filesystem driver uses a seperate database interface class to allow different databases to be supported. The database interface class must implement the *org.alfresco.jlan.server.filesys.db.DBInterface* interface, plus it may also implement a number of optional interfaces.

The optional interfaces that are required depend on the file loader class that is being used. The database interface class has a set of feature flags which the file loader can check. If the database interface does not implement all of the required features an exception will be thrown during initialization of the filesystem.

The database interfaces hide all details of the database access, there are no JDBC objects or SQL strings passed between the database filesystem driver and database interface class. The database interface does not need to be JDBC or SQL based.

The following table lists the database interfaces that may be implemented. All interfaces are contained in the *org.alfresco.jlan.server.filesys.db* package.

Interface	Description
DBInterface	Main database interface that provides methods to store and retrieve the filesystem structure details. This interface must be implemented.
DBDataInterface	Provides methods for loading and saving file data to a database, or other repository. Required by the DBFileLoader file loader implementation.
DBQueueInterface	Provides methods for loading and saving file requests to a database. This is used by the background file loader where file data is loaded to a temporary caching area. Required by the DBFileLoader and ObjectIdFileLoader file loader implementations.
DBObjectIdInterface	Provides methods for storing filesystem file id to repository object ids to a database. Required by the ObjectIdFileLoader implementation.

4.1.1 Implementing The Main Database Interface

The *org.alfresco.jlan.filesys.server.db.DBInterface* interface provides the methods to add/change/delete file and folder records, return the file list for a particular folder, initialize and shutdown the database interface, calculate the used space and convert file/folder names to unique file ids.

DBInterface has the following methods :-

- createFileRecord
- createStreamRecord
- deleteFileRecord
- deleteStreamRecord
- deleteSymbolicLink
- fileExists
- getDBInterfaceName
- getFileId
- getFileInformation
- getFileRetentionDetails
- getStreamInformation
- getStreamsList
- getUsedFileSpace
- initializeDatabase
- readSymbolicLink
- renameFileRecord
- renameStreamRecord
- requestFeatures
- setFileInformation
- setStreamInformation
- shutdownDatabase
- startSearch
- supportsFeatures

The *org.alfresco.jlan.server.filesys.db.JdbcDBInterface* is an abstract class that implements the DBInterface interface and adds common functionality required by JDBC based database implementations, including a connection pool.

The *org.alfresco.jlan.server.filesys.SearchContext* abstract class must be extended to provide the context for a folder search. The *org.alfresco.jlan.server.filesys.db.DBSearchContext* abstract class can be used if the folder search returns a java.sql.ResultSet.

4.1.1.1 Filesystem Structure Details

The DBInterface database must store the details of files, folders and NTFS streams, and keep track of file retention details per file/folder, if enabled.

The minimum information that must be stored per file or folder is detailed in the following table along with the Java data type.

Field	Java Type	Description
FileId	int	Unique identifier for a file or folder that should start with file id one. File id zero is used as the root folder file id. The database interface implementation is responsible for generating the unique file id when a new file/folder record is created. This file id should be the primary key.
DirId	int	Parent directory id. Directory id zero is used as the root folder file id.
FileName	String	File name string, not including any path. The file name field should allow file names of up to 255 characters.
FileSize	long	File size in bytes.
CreateDate	long	File creation date/time as milliseconds since 1 Jan 1970 00:00:00 GMT.
ModifyDate	long	File modification date/time as milliseconds since 1 Jan 1970 00:00:00 GMT.
AccessDate	long	File access date/time as milliseconds since 1 Jan 1970 00:00:00 GMT.
ChangeDate	long	File change date/time as milliseconds since 1 Jan 1970 00:00:00 GMT.
ReadOnlyFile	int (bit mask)	Read only file attribute. Stored as a bit mask within the FileInfo class.
ArchivedFile	int (bit mask)	Archived file attribute. Stored as a bit mask within the FileInfo class.
DirectoryFile	int (bit mask)	Directory file attribute, indicating that this file record is for a folder. Stored as a bit mask within the FileInfo class.
SystemFile	int (bit mask)	System file attribute. Stored as a bit mask within the FileInfo class.
HiddenFile	int (bit mask)	Hidden file attribute. Stored as a bit mask within the FileInfo class.
OwnerUid	int	Owner user id, used by the NFS server.
OwnerGid	int	Owner group id, used by the NFS server.
FileMode	int	File mode flags, used by the NFS server.
IsDeleted	boolean	Flag to indicate if the file is deleted. The database filesystem driver can be configured to delete file records or mark them as deleted.

The FileId field should be the primary key. There should also be indexes on the following fields :-

- FileName and DirId
- DirId
- DirId and DirectoryFile
- FileName and DirId and DirectoryFile

NTFS streams are seperate parts of a file, like a file within a file, that are linked to the main file. Streams are often used to store metadata about the main file.

The following table details the NTFS streams information that must be stored by the database interface class.

Field	Java Type	Description
StreamId	int	Unique identifier for a stream that should start with stream id one. Stream id zero is used to indicate the main file stream. The database interface implementation is responsible for generating the unique stream id when a new stream record is created. The stream id should be the primary key.
FileId	int	File id that the stream belongs to
StreamName	String	Stream name string, not including any path. The stream name field should allow stream names of up to 255 characters.
StreamSize	long	Stream size in bytes.
CreateDate	long	Stream creation date/time as milliseconds since 1 Jan 1970 00:00:00 GMT.
ModifyDate	long	Stream modification date/time as milliseconds since 1 Jan 1970 00:00:00 GMT.
AccessDate	long	Stream access date/time as milliseconds since 1 Jan 1970 00:00:00 GMT.

The StreamId field should be the primary key. There should also be an index on the FileId field.

When the top level file is deleted all associated stream records should also be deleted.

If file retention is enabled a file retention record should be created when a new file or folder is created. The retention expiry date/time can be stored as part of the main file/folder record or using a seperate table linked via the file id.

The following table details the retention information that must be stored by the database interface class.

Field	Java Type	Description
FileId	int	File id of the file that is being retained. The file id field should be the primary key.
StartDate	long	Date/time that file retention starts, as milliseconds since 1 Jan 1970 00:00:00 GMT.

Field	Java Type	Description
EndDate	long	Date/time that file retention ends, as milliseconds since 1 Jan 1970 00:00:00 GMT.

4.1.1.2 DBInterface Methods

This section details the main database interface methods.

4.1.1.2.1 createFileRecord

```
int createFileRecord(String fname, int dirId, FileOpenParams params,
                     boolean retain)
          throws DBException, FileExistsException;
```

Create a new record for a file or folder and return the allocated unique file id.

The *fname* parameter contains the file/folder name not including any path. The *dirId* parameter contains the file id of the parent directory that the new file/folder is linked to. The *params* parameter contains the full file open request parameters. The *retain* parameter indicates whether a retention record should be created for the new file/folder.

4.1.1.2.2 createStreamRecord

```
int createStreamRecord(String sname, int fileId)
          throws DBException;
```

Create a new record for an NTFS stream and return the allocated unique stream id.

The *sname* parameter specifies the name of the new NTFS stream. The *fileId* parameter specifies the file that the new stream belongs to.

4.1.1.2.3 deleteFileRecord

```
void deleteFileRecord(int dirId, int fileId, boolean markOnly)
          throws DBException, DirectoryNotEmptyException;
```

Delete a file record from the database, or if the *markOnly* parameter is true mark the file/folder as deleted so it does not appear in directory searches.

The *dirId* parameter specifies the parent directory id, the *fileId* parameter specifies the file record to be deleted. The *markOnly* parameter specifies that the file record should be marked as deleted if true or the record should be deleted if false.

4.1.1.2.4 deleteStreamRecord

```
void deleteStreamRecord(int fileId, int streamId, boolean markOnly)
          throws DBException;
```

Delete an NTFS stream from the database, or if the *markOnly* parameter is true mark the stream as deleted.

The *fileId* parameter specifies the owner file, the *streamId* parameter specifies the unique id of the stream to be deleted. The *markOnly* parameter specifies that the stream record should be marked as deleted if true or the record should be deleted if false.

4.1.1.2.5 deleteSymbolicLink

```
void deleteSymbolicLink(int dirId, int fileId)
         throws DBException;
```

Delete a symbolic link record.

The *dirId* parameter specifies the parent directory id, the *fileId* parameter specifies the unique file id of the record to be deleted.

4.1.1.2.6 fileExists

```
int fileExists(int dirId, String fname)
          throws DBException;
```

Determine if the specified file or folder exists within the parent directory. Return a status to indicate if a file exists, folder exists or the file/folder does not exist.

The *dirId* parameter specifies the parent directory that the file or folder should be contained within. The *fname* parameter specifies the file/folder name to check for.

The returned status code values are contained in the *org.alfresco.jlan.server.filesys.FileStatus* class. The available values are :-

- FileStatus.NotExist
- FileStatus.FileExists
- FileStatus.DirectoryExists

4.1.1.2.7 getDBInterfaceName

```
String getDBInterfaceName();
```

Return the database interface name.

4.1.1.2.8 getFileId

```
int getFileId(int dirId, String fname, boolean dirOnly, boolean caseLess)
          throws DBException;
```

Lookup a file/folder name and return the unique file id, or -1 if the file/folder does not exist.

The *dirId* parameter specifies the parent directory to search for the file/folder. The *fname* parameter specifies the file/folder name to search for. The *dirOnly* flag specifies that the search is for a directory. The *caseLess* parameter specifies that a case sensitive search be used if false, and a case insensitive search be used if true.

4.1.1.2.9 getFileInformation

```
DBFileInfo getFileInformation(int dirId, int fileId, int infoLevel)
          throws DBException;
```

Return file information for the specified file.

The *dirId* parameter specifies the parent directory id, the *fileId* parameter specifies the unique file id of the file/folder to return information for. The *infoLevel* parameter specifies the level of detail to be returned, the possible values are :-

- DBInterface.FileNameOnly
  Return the file name.
- DBInterface.FileIds
  Return the file name, file id and parent directory id.

- DBInterface.FileAll
  Return all file details.

4.1.1.2.10 getFileRetentionDetails

```
RetentionDetails getFileRetentionDetails(int dirId, int fileId)
        throws DBException;
```

Return the file retention details for the specified file/folder, or null if the file has no retention record.

The *dirId* parameter specifies the parent directory id, the *fileId* parameter specifies the unique id of the file/folder to return retention details for.

4.1.1.2.11 getStreamInformation

```
StreamInfo getStreamInformation(int fileId, int streamId, int infoLevel)
        throws DBException;
```

Return information for an NTFS stream.

The *fileId* parameter specifies the unique id of the file that owns the NTFS stream. The *streamId* parameter specifies the unique id of the NTFS stream to return information for. The *infoLevel* parameter specifies the details to be returned, the possible values are:-

- DBInterface.StreamNameOnly
  Stream name only.
- DBInterface.StreamIds
  Stream name, file id and stream id.
- DBInterface.StreamAll
  Return all stream details.

4.1.1.2.12 getStreamsList

```
StreamInfoList getStreamsList(int fileId, int infoLevel)
        throws DBException;
```

Return the list of available NTFS streams for the specified file.

The *fileId* parameter specifies the unique id of the file to return the NTFS streams list for. The *infoLevel* parameter specifies the details to be returned, the possible values are the same as the information levels for the *getStreamInformation()* method.

4.1.1.2.13 getUsedFileSpace

```
long getUsedFileSpace();
```

Return the total space used, in bytes, by all files within the database by totalling up the file sizes.

4.1.1.2.14 initializeDatabase

```
void initializeDatabase(DBDeviceContext dbCtx, ConfigElement params)
        throws InvalidConfigurationException;
```

Initalize the database interface. Initialization may involve creating the appropriate database tables, indexes etc.

The *dbCtx* parameter specifies the database filesystem device context. The *params*

parameter specifies the database interface configuration parameters.

4.1.1.2.15 readSymbolicLink

```
String readsymbolicLink(int dirId, int fid)
         throws DBException;
```

Return the symbolic link contents.

The *dirId* parameter specifies the parent directory id, the *fid* parameter specifies the unique file id of the file.

4.1.1.2.16 renameFileRecord

```
void renameFileRecord(int dirId, int fileId, String newName, int newDir)
          throws DBException;
```

Rename a file/folder and optionally move it to another parent directory.

The *dirId* parameter specifies the parent directory id, the *fileId* parameter specifies the unique file id of the file/folder to be renamed. The *newName* parameter specifies the new name for the file/folder. The *newDir* parameter specifies the new parent directory for the file/folder.

4.1.1.2.17 renameStreamRecord

```
void renameStreamRecord(int dirId, int fileId, int streamId, String newName)
          throws DBException;
```

Rename an NTFS stream.

The *dirId* parameter specifies the parent directory id, the *fileId* parameter specifies the unique file id of the file that owns the NTFS stream. The *streamId* parameter specifies the unique stream id of the stream to be renamed. The *newName* parameter specifies the new stream name.

4.1.1.2.18 requestFeatures

```
void requestFeatures(int mask)
          throws DBException;
```

Called to request that the specified optional database features are enabled, and that the database interface supports the requested features.

The *mask* parameter specifies the feature bit mask, possible values are combinations of the following values :-

- DBInterface.FeatureNTFS
  Enable NTFS streams support.
- DBInterface.FeatureRetention
  Enable file/folder retention support.
- DBInterface.FeatureQueue
  File loader save/load queue.
- DBInterface.FeatureData
  Load/save file data to database fields.
- DBInterface.FeatureJarData

Load/save multiple file data to Jar files.

- DBInterface.FeatureObjectId
  File id to external object id mapping.
- DBInterface.FeatureSymLinks
  Symbolic links supported.

4.1.1.2.19 setFileInformation

```
void setFileInformation(int dirId, int fileId, FileInfo info)
          throws DBException;
```

Set file/folder details such as access/modification date/times, file size, file attributes.

The *dirId* parameter specifies the parent directory id, the *fileId* parameter specifies the unique id of the file/folder to be modified. The *info* parameter contains the new file/folder details.

The FileInfo object has a mask of the fields that are valid that can be checked by using the hasSetFlag(int) method. The possible FileInfo set flags are :-

- FileInfo.SetFileSize
- FileInfo.SetAllocationSize
- FileInfo.SetAttributes
- FileInfo.SetModifyDate
- FileInfo.SetCreationDate
- FileInfo.SetAccessDate
- FileInfo.SetChangeDate
- FileInfo.SetGid
- FileInfo.SetUid
- FileInfo.SetMode

4.1.1.2.20 setStreamInformation

```
void setStreamInformation(int dirId, int fileId, int streamId,
                          StreamInfo info)
          throws DBException;
```

Set NTFS stream information such as stream size, modification date/time.

The *dirId* parameter specifies the parent directory id, the *fileId* parameter specifies the unique id of the file that owns the stream. The *streamId* parameter specifies the unique id of the NTFS stream to be updated. The *info* parameter specifies the stream details to be updated.

The StreamInfo object has a mask of the fields that are valid that can be checked using the hasSetFlag(int) method. The possible StreamInfo set flags are :-

- StreamInfo.SetStreamSize
- StreamInfo.SetAllocationSize

- StreamInfo.SetModifyDate
- StreamInfo.SetCreationDate
- StreamInfo.SetAccessDate

4.1.1.2.21 shutdownDatabase

```
void shutdownDatabase(DBDeviceContext dbCtx)
```

Shutdown the database interface and release any resources.

4.1.1.2.22 startSearch

```
SearchContext startSearch(int dirId, String searchPath, int attrib,
                          int infoLevel, int maxRecords)
        throws DBException;
```

Start a search of a folder for a specified file/folder or a wildcard search for a number of files/folders.

The *dirId* parameter specifies the folder to be searched. The *searchPath* specifies the file(s)/folder(s) to search for and may contain wildcard characters. The *attrib* parameter specifies the types of files/folders to be returned. The *infoLevel* parameter specifies the level of detail to return for each file/folder, the possible values are :-

- DBInterface.FileNameOnly
  Return the file name only.
- DBInterface.FileIds
  Return the file name, file id and parent directory id.
- DBInterface.FileAll
  Return all file details.

The *maxRecords* parameter specifies the maximum number of records to return, or -1 if there is no limit.

4.1.1.2.23 supportsFeatures

```
boolean supportsFeatures(int feature);
```

Check if the database interface supports the specified feature.

The feature parameter specifies the database feature to check for, possible values are :-

- DBInterface.FeatureNTFS
  Supports NTFS streams.
- DBInterface.FeatureRetention
  Supports file/folder retention.
- DBInterface.FeatureQueue
  Supports background load/save queue, implements the *DBQueueInterface*.
- DBInterface.FeatureData
  Supports storing file data in database fields, implements the *DBDataInterface*.
- DBInterface.FeatureJarData
  Supports storing file data in the database by packaging collections of small

files into Jar files, implements the *DBDataInterface* Jar methods.

- DBInterface.FeatureObjectId
  Supports the file id to object id mapping database, implements the *DBObjectIdInterface*.
- DBInterface.FeatureSymLinks
  Supports symbolic links.

The file loader will check the features that the database interface has to determine if all the required features are available, if not then an exception will be thrown during filesystem initialization.

4.1.2 Implementing The Data Interface

The *org.alfresco.jlan.server.filesys.db.DBDataInterface* is an optional interface that may be implemented to provide the ability to load and save file data to database BLOB fields or repository objects.

This is interface is usually used in conjunction with the DBQueueInterface that provides the queueing of load and save requests used by the background load/save component.

The DBDataInterface has the following methods :-

- deleteFileData
- deleteJarData
- getFileDataDetails
- getMaximumFragmentSize
- loadFileData
- loadJarData
- saveFileData
- saveJarData

4.1.2.1 Data Storage

This section details how the various relational database interfaces that are included in the JLAN Server kit implement the data storage interface.

The database interface implementations included in the JLAN Server kit split the data storage into two database tables, one holds the file data in BLOB fields and the other holds the Jar file data in BLOB fields. The main data table has the following fields :-

Field	Java Type	Description
FileId	int	File id from the main filesystem structure table.
StreamId	int	Stream id from the main filesystem structure table. A stream id of zero indicates that this is the main file data.
FragNo	int	File data fragment number. The file data is written to the database in manageable chunks of data.
FragLen	int	Length of this file data fragment.
Data	byte[]	BLOB field holding the file data fragment.

Field	Java Type	Description
JarFile	boolean	Flag to indicate if the file data is contained in a Jar file. If the data is stored in a Jar file with other files the Data field will be empty.
JarId	int	Unique id of the Jar file record that holds the Jar data for this file.

The data interface provides methods for loading and saving Jar files that may contain many small files. The Jar data table has the following fields :-

Field	Java Type	Description
JarId	int	Unique identifier for the Jar record. The data interface is responsible for generating the unique Jar id. The jar id is the primary key.
Data	byte[]	BLOB field holding the Jar file data.

4.1.2.2 DBDataInterface Methods

This section details the data interface methods.

4.1.2.2.1 deleteFileData

```
void deleteFileData(int fileId, int streamId)
          throws DBException, IOException;
```

Delete the file data for the specified file or NTFS stream. If the streamId value is zero it indicates that the main file data should be deleted.

4.1.2.2.2 deleteJarData

```
void deleteJarData(int jarId)
          throws DBException, IOEXception;
```

Delete the Jar file data for the specified Jar.

4.1.2.2.3 getFileDataDetails

```
DBDataDetails getFileDataDetails(int fileId, int streamId)
          throws DBException;
```

Return details of the file data for the specified file or NTFS stream.

The returned DBDataDetails object indicates whether the file/stream data is stored in the main data table or if it is contained in a Jar file.

4.1.2.2.4 getMaximumFragmentSize

```
long getMaximumFragmentSize();
```

Return the maximum data fragment size that the data interface can handle, in bytes.

4.1.2.2.5 loadFileData

```
void loadFileData(int fileId, int streamId, FileSegment fileSeg)
          throws DBException, IOException;
```

Load file data for the specified file or NTFS stream. The *fileSeg* parameter contains the details of the local temporary file that the file data should be written to.

4.1.2.2.6 loadJarData

```
void loadJarData(int jarId, FileSegment fileSeg)
          throws DBException, IOException;
```

Load file data for the specified Jar file. The *fileSeg* parameter contains the details of the local temporary file that the Jar file data should be written to.

4.1.2.2.7 saveFileData

```
int saveFileData(int fileId, int streamId, FileSegment fileSeg)
          throws DBException, IOException;
```

Save file data for the specified file or NTFS stream to the data table. The *fileSeg* parameter contains the details of the temporary file that holds the data to be written to the database.

The method returns the number of data fragments written to the data table.

4.1.2.2.8 saveJarData

```
int saveJarData(String jarFile, DBDataDetailsList fileList)
          throws DBException, IOException;
```

Save Jar file data to the Jar data table.

The *jarFile* parameter contains the path to the Jar file to be stored. The *fileList* parameter contains the list of files that are contained within the Jar file. The *fileList* is used to update the main data records to point to the newly created Jar data record.

The method returns the unique Jar id that was allocated.

4.1.3 Implementing the Queueing Interface

The *org.alfresco.jlan.server.filesys.db.DBQueueInterface* is an optional interface that maybe implemented to provide the ability to queue file load and save requests to a pool of background worker threads.

By writing the file request queue to the database the background load/save component can handle thousands of requests and provides a mechanism for crash recovery.

This interface is usually used in conjunction with the DBDataInterface to load/save the file data using a background worker thread pool.

The DBQueueInterface has the following methods :-

- deleteFileRequest
- loadFileRequests
- loadTransactionRequest
- performQueueCleanup

- queueFileRequest

There are currently three types of file request that may be submitted to the queue – file load request, file save request and a transaction save request. The transaction save request is used to pack multiple small files into a Jar file which is then saved as a single file.

4.1.3.1 File Request Queue Storage

This section details how the various relational database interfaces that are included in the JLAN Server kit implement the file request queue storage interface.

The main file request queue table holds the single file load/save requests, with one record per request. The database table and has the following fields :-

Field	Java Type	Description
FileId	int	File id of the file to be loaded or saved.
StreamId	int	Stream id of the file/stream to be loaded or saved.
ReqType	int	Request type. 0 for a file load, 1 for a file save.
SeqNo	int	Unique sequence number for this request.
TempFile	String	Path to the temporary file.
VirtualPath	String	Virtual path relative to the virtual filesystem root.
QueuedAt	long	Date/time the request was queued.

The transaction request queue table holds multiple records per save request, and has the following fields :-

Field	Java Type	Description
FileId	int	File id of the file to be saved.
StreamId	int	Stream id of the file/stream to be saved.
TranId	int	Transaction id. All files within the same transaction request have the same transaction id.
ReqType	int	Request type. 2 for transaction save.
TempFile	String	Path to the temporary file.
VirtualPath	String	Virtual path relative to the virtual filesystem root.
QueuedAt	long	Date/time the request was queued.

4.1.3.2 DBQueueInterface Methods

This section details the DBQueueInterface methods.

4.1.3.2.1 deleteFileRequest

```
void deleteFileRequest(FileRequest fileReq)
          throws DBException;
```

Delete a file request from the queue database. The fileReq object contains the unique sequence number for the file reqeust record to be deleted.

4.1.3.2.2 loadFileRequest

```
int loadFileRequests(int seqNo, int reqType, FileRequestQueue queue,
                                int recLimit)
          throws DBException;
```

Load a number of file request records and add them to the specified request queue.

The *seqNo* parameter specifies the sequence number of the last record loaded from the database. The *reqType* parameter specifies the type of file request records to load from the database. The *queue* parameter specifies the file request queue to add the FileRequest objects to. The *recLimit* parameter specifies the maximum number of records to load.

4.1.3.2.3 loadTransactionRequest

```
MultipleFileRequest loadTransactionRequest(MultipleFileRequest fileReq)
          throws DBException;
```

Load a transaction request containing multiple files to be saved.

The *fileReq* parameter contains the transaction id of the transaction to be loaded.

4.1.3.2.4 performQueueCleanup

```
int performQueueCleanup(File tempDir, String tempDirPrefix,
                        String tempFilePrefix, String jarFilePrefix)
          throws DBException;
```

Perform queue cleanup during startup by matching files in the temporary caching area with file save requests in the single file and transaction queue database tables. File load requests can be removed from the queue as the server is starting so there will be no active sessions.

The *tempDir* parameter specifies the location of the temporary caching area on the local filesystem. The *tempDirPrefix* parameter specifies the prefix used to create temporary sub-directories within the temporary by the associated file loader. The *tempFilePrefix* parameter specifies the temporary file prefix used by the associated file loader. The *jarFilePrefix* specifies the Jar file prefix used by the associated loader.

The method returns the number of files recovered from the temporary area that have matching save requests.

4.1.3.2.5 queueFileRequest

```
void queueFileRequest(FileRequest fileReq)
          throws DBException;
```

Queue a file load, save or transaction request.

The *fileReq* parameter specifies the file request details. To check if the file request is part of a transaction request use the isTransaction() method.

4.1.4 Implementing The Object Id Interface

The org.alfresco.jlan.server.filesys.db.DBObjectIdInterface is an optional interface that may be implemented to provide mapping between the unique file id used by the database

filesystem and object ids used in an external repository.

This interface is used by the ObjectIdFileLoader abstract class. To implement a file loader with temporary file caching support and a background load/save thread pool only requires implementing two methods – loadFileData() and saveFileData().

The DBObjectIdInterface has the following methods :-

- deleteObjectId
- loadObjectId
- saveObjectId

4.1.4.1 Object Id Storage

This section details how the various relational database interfaces that are included in the JLAN Server kit implement the object id mapping interface.

The object id mapping table has the following fields :-

Field	Java Type	Description
FileId	int	File id within the database filesystem
StreamId	int	Stream id within the database filesystem
ObjectId	String	Object id within the external repository

4.1.4.2 DBObjectIdInterface Methods

This section details the DBObjectIdInterface methods.

4.1.4.2.1 deleteObjectId

```
void deleteObjectId(int fileId, int streamId, String objectId)
          throws DBException;
```

Delete an object id from the mapping database.

4.1.4.2.2 loadObjectId

```
String loadObjectId(int fileId, int streamId)
          throws DBException;
```

Load an object id mapping for the specified database filesystem file/stream.

4.1.4.2.3 saveObjectId

```
void saveObjectId(int fileId, int streamId, String objectId)
          throws DBException;
```

Save, or update, an object id mapping for the specified database filesystem file/stream.

4.2 Writing A File Loader Implementation

The main database filesystem driver uses a seperate file loader interface class to provide file data loading and saving. The file loader design is flexible so that it can handle direct access to the file data or can use a thread pool to load/save the file data in background.

There are a number of file loader implementations included in the JLAN Server kit that can either be extended or are ready to use.

The following table lists the various file loader interfaces and implementations that are available in the JLAN Server kit :-

Name	Type	Description
FileLoader	Interface	The base interface that all file loaders must implement.
NameFileLoader	Interface	Extends the FileLoader interface to provide methods to recreate the directory structure.
SimpleFileLoader	Class	NamedFileLoader implementation that stores file data using the local filesystem, recreating the directory structure and file names. This is a complete file loader implemention that can be used as is. The database interface only needs to implement the base DBInterface interface.
DBFileLoader	Class	FileLoader implementation that loads/saves the file data to database BLOB fields. The loader caches the file data in a temporary directory using a thread pool of background worker threads to load/save the file data as files are opened/created. This is a complete file loader that can be used as is. The database interface must implement the DBQueueInterface and DBDataInterface interfaces.
ObjectIdFileLoader	Abstract class	FileLoader implementation that loads/saves the file data to a repository. The loader caches the file data in a temporary directory using a thread pool of background worker threads to loads/save the file data as files are opened/created. The database filesystem file ids are mapped to repository object ids. The loadFileData() and saveFileData() methods must be implemented to load/save the file data to/from the repository.

4.2.1 Implementing The File Loader Interface

A file loader must implement the *org.alfresco.jlan.server.filesys.loader.FileLoader* interface, which has the following methods :-

- addFileProcessor
- closeFile
- deleteFile
- getRequiredDBFeatures

- initializeLoader
- openFile
- queueFileRequest
- shutdownLoader
- supportsStreams

The basic file loader interface only receives requests for files, folder requests are handled by the main database filesystem driver. If the file loader requires details of the directory structure the *org.alfresco.jlan.server.filesys.loader.NamedFileLoader* interface should be implemented. The NamedFileLoader interface extends the FileLoader interface to add methods for creating/deleting/renaming folders and for setting file/folder information, it has the following methods :-

- createDirectory
- deleteDirectory
- renameFileDirectory
- setFileInformation

The *org.alfresco.jlan.server.filesys.loader.SimpleFileLoader* is an implementation of a NamedFileLoader. The SimpleFileLoader creates the directory structure on the local filesystem.

4.2.1.1 FileLoader Methods

This section details the file loader methods.

4.2.1.1.1 addFileProcessor

```
void addFileProcessor(FileProcessor proc)
          throws FileLoaderException;
```

Add a file processor to the list of file load/save processors. File processors implement the *org.alfresco.jlan.server.filesys.loader.FileProcessor* interface, they can be used to process the file data just after file loading or just before file saving.

The *org.alfresco.jlan.jlansrv.Md5SumFileProcessor* is a sample FileProcessor that produces an MD5 sum of the file data that is compatible with the md5sum command available on various operating systems.

If file processors are not supported by the file loader an exception should be thrown.

4.2.1.1.2 closeFile

```
void closeFile(NetworkFile file)
          throws IOException;
```

Close a file previously opened via the openFile() file loader method.

The *org.alfresco.jlan.server.filesys.NetworkFile* class is an abstract class that provides methods to access the file data and information.

4.2.1.1.3 deleteFile

```
void deleteFile(String fname, int fileId, int streamId)
```

```
        throws IOException;
```

Delete the file data associated with the specified file.

4.2.1.1.4 getRequiredDBFeatures

```
int getRequiredDBFeatures();
```

Return a bit-mask of the database features that the loader requires from the database interface implementation. The returned database feature bit-mask can be a combination fo the following values, or zero if the file loader does not require and database features :-

- DBInterface.FeatureNTFS
  Requires NTFS streams support.
- DBInterface.FeatureRetention
  Requires data retention support.
- DBInterface.FeatureData
  Requires the database interface support loading/saving file data to the database, using BLOB fields or other persistence mechanism.
- DBInterface.FeatureJarData
  Requires the database interface support the loading/saving of Jar files that contain multiple files from the database filesystem.
- DBInterface.FeatureQueue
  Requires file request queue support.
- DBInterface.FeatureObjectId
  Requires database filesystem file id to repository object id mapping support.
- DBInterface.FeatureSymLinks
  Requires symbolic links support.

4.2.1.1.5 initalizeLoader

```
void initializeLoader(ConfigElement params, DeviceContext ctx)
        throws FileLoaderException, IOException;
```

Initalize the file loader.

The *params* parameter contains the configuration values for the file loader. The *ctx* parameter contains the associated device context, this will be a DBDeviceContext object for the database filesystem.

4.2.1.1.6 openFile

```
NetworkFile openFile(FileOpenParams params, int fileId, int streamId,
                     int dirId, boolean create, boolean dir)
        throws IOException, FileNotFoundException;
```

Open a file and return a NetworkFile class that will be used to read/write file data.

The *params* parameter contains the full file open/create parameters from the databse filesystem open/create request. The *fileId* parameter contains the unique id of the file. The *streamId* parameter contains the unique NTFS stream id or zero for the main file. The *dirId* parameter specifies the unique file id of the parent directory. The *create* parameter indicates whethe the file should be created if it does not exist. The *dir* parameter indicates that the file open/create is for a directory if true or a file if false.

4.2.1.1.7 queueFileRequest

```
void queueFileRequest(FileRequest fileReq);
```

Queue a file load/save request.

File load/save requests are used by file loaders that use the background load/save thread pool component.

4.2.1.1.8 shutdownLoader

```
void shutdownLoader(boolean immediate);
```

Shutdown the file loader and release any resources.

The *immediate* parameter indicates if the file loader should shutdown gracefully, if false, or immediately, if true.

4.2.1.1.9 supportsStreams

```
boolean supportSStreams();
```

Determine if the file loader implementation supports NTFS streams.

The SimpleFileLoader does not support NTFS streams as it uses the java.io.File class to access the file data. The java.io.File class does not support NTFS streams.

4.2.1.2 NamedFileLoader Methods

This section details the named file loader methods.

4.2.1.2.1 createDirectory

```
void createDirectory(String dir, int fileId)
          throws IOException;
```

Create a directory.

The *dir* parameter specifies the relative path for the new directory. The *fileId* parameter specifies the unique file id of the new directory.

4.2.1.2.2 deleteDirectory

```
void deleteDirectory(String dir, int fileId)
          throws IOException;
```

Delete a directory.

The *dir* parameter specifies the relative path for the directory to be deleted. The *fileId* parameter specifies the unique file id of the directory.

An IOException should be thrown if the directory is not empty.

4.2.1.2.3 renameFileDirectory

```
void renameFileDirectory(String curName, int fileId, String newName,
                         boolean isDir)
          throws IOException;
```

Rename a file or folder.

The *curName* parameter specifies the relative path to the current file or folder. The *fileId* parameter specifies the unique file id of the file or folder. The *newName* parameter specifies the new relative path to the file or folder, the file/folder may be moved to a new parent folder. The *isDir* parameter indicates if the path is for a file or folder.

4.2.1.2.4 setFileInformation

```
void setFileInformation(String path, int fileId, FileInfo info)
          throws IOException;
```

Set file information for a file or folder.

The *path* parameter specifies the relative path to the file or folder to set information for. The *fileId* parameter specifies the unique id of the file or folder. The *info* parameter specifies the new file/folder details.

The FileInfo object has a mask of the fields that are valid that can be checked by using the hasSetFlag(int) method. The possible FileInfo set flags are :-

- FileInfo.SetFileSize
- FileInfo.SetAllocationSize
- FileInfo.SetAttributes
- FileInfo.SetModifyDate
- FileInfo.SetCreationDate
- FileInfo.SetAccessDate
- FileInfo.SetChangeDate
- FileInfo.SetGid
- FileInfo.SetUid
- FileInfo.SetMode

4.2.2 Extending The ObjectIdFileLoader

The *org.alfresco.jlan.server.filesys.db.ObjectIdFileLoader* class is a file loader implementation that includes caching of file data to a temporary directory on the local filesystem, mapping of database filesystem ids to repository object ids and background loading/saving of file data to/from the repository.

The ObjectIdFileLoader requires a database interface that implements the DBObjectId Interface to provide the persistence for the file id to object id mappings and the DBQueueInterface to store the file load/save request queue.

The ObjectIdFileLoader is an abstract class that requires you to implement the file data load and save methods. The load/save methods are called from a thread pool of worker threads that load/save the file data to/from the repository.

4.2.2.1 ObjectIdFileLoader Methods To Implement

This section details the methods that must be implemented when extending the ObjectIdFileLoader abstract class.

The *initializeLoader()* and *shutdownLoader()* methods can also be overridden but must call the ObjectIdFileLoader methods.

4.2.2.1.1 loadFileData

```
void loadFileData(int fileId, int streamId, String objectId,
                  FileSegment fileSeg)
          throws IOException;
```

Load the file data for the specified database filesystem file id with the corresponding repository object id to a temporary file on the local filesystem.

The *fileId* parameter specifies the unique id of the file in the database filesystem. The *streamId* parameter specifies the unique NTFS stream id or zero for the main file data stream. The *objectId* parameter specifies the id of the object within the repository. The *fileSeg* parameter contains the details of the temporary file to load the file data into.

During file data loading the method should call the FileSegment *setReadableLength(long)* and *signalDataAvailable()* methods. Any session threads that are trying to read the file data may use the FileSegment to wait for data to be loaded. The *signalDataAvailable()* method is used to wakeup any waiting threads and the readable length indicates the amount of valid data that can be read from the file.

4.2.2.1.2 saveFileData

```
String saveFileData(int fileId, int streamId, FileSegment fileSeg)
          throws IOException;
```

Save the file data for the specified database filesystem file id and stream id to the repository.

The *fileId* parameter specifies the unique id of the file in the database filesystem. The *streamId* specifies the unique NTFS stream id or zero for the main file data stream. The *fileSeg* parameter contains the details of the temporary file to save the data from.

The method must return the object id allocated to the file data by the repository. The object id string can be up to 255 characters.

5 Server Configuration

The ServerConfiguration class is a container for the configuration sections for the various protocols and other components within the JLAN Server.

The base ServerConfiguration class does not implement any load or save functionality so must be extended in order to provide this.

A listener mechanism is included in the ServerConfiguration class to inform registered listeners when a configuration parameter is modified. The configuration listener can return a status to indicate if the change was ignored or accepted, and whether the change will apply to new sessions only or a restart is required.

All of the main JLAN Server components register with the ServerConfiguration so that they receive the configuration change events. The aim is to eventually have a server that can be configured dynamically without the need to restart.

The sample JLANServer application uses an XML based ServerConfiguration implementation that uses the JDOM API to load and save the configuration.

A ServerConfiguration implementation must override the following methods:-

- *loadConfiguration*
- *saveConfiguration*

5.1 loadConfiguration

```
void loadConfiguration(String location)
     throws IOException, InvalidConfigurationException;
```

The *loadConfiguration()* method is responsible for loading and validating the server configuration values.

The *location* parameter is implementation specific, the *XMLServerConfiguration* class uses the location parameter as the file name to load the configuration from.

The default implementation in the ServerConfiguration base class throws an IOException to indicate that *loadConfiguration()* has not been implemented.

5.2 saveConfiguration

```
void saveConfiguration(String location)
     throws IOException;
```

The *saveConfiguration()* method is responsible for saving the server configuration values.

The *location* parameter is implementation specific, the *XMLServerConfiguration* class uses the location parameter as the file name to save the configuration to.

The default implementation in the ServerConfiguration base class throws an IOException to indicate that save*Configuration()* has not been implemented.

6 Cifs Authentication

The JLAN Server CIFS authentication is handled by the *CifsAuthenticator* class.

The latest implementations of the *CifsAuthenticator* class now integrate more tightly into the CIFS server. In order to implement the latest CIFS authentication mechanisms the *CifsAuthenticator* must control the negotiate and session setup phases of the CIFS server. Rather than implement your own authenticator class you can just provide the user account information for the authenticator by implementing the *UsersInterface* interface.

6.1 UsersInterface

The UsersInterface interface provides user account information to the CifsAuthenticator during the initialization of a new session. The user account information provides the user name and password, plus optional information such as whether the account is an administrator or guest, and the home directory for the user.

The user account information may specify the password in plaintext or as an MD4 hashed value. The MD4 hashed value is the preferred way to store the password as it is more secure. MD4 allows the password to be stored within a database or within the XML configuration file, for example. The plaintext password may be specified but will only be used if the MD4 version of the password is not available.

The UsersInterface inteface has the following methods to implement :-

- *initializeUsers*
- *getUserAccount*

The *initializeUsers(ServerConfiguration, NameValueList)* initializes the users interface.

The *getUserAccount(String)* method returns a *UserAccount* object with the details of the specified user including the password. If the user name is not valid the method should return null.

The *DefaultUsersInterface* implementation provides a user list from the XML server configuration file. This provides backwards compatibility with earlier versions of the JLAN Server.

6.1.1 UsersInterface Methods

The following section details the UsersInterface methods.

6.1.1.1 initializeUsers

```
void initializeUsers(ServerConfiguration config, NameValueList params)
    throws InvalidConfigurationException;
```

Initializes the UsersInterface using the specified parameter list.

6.1.1.2 getUserAccount

```
UserAccount getUserAccount(String userName);
```

Return the user account details for the specified user, or null if the user name is not valid.

The returned UserAccount object can contain the MD4 hashed password and/or plaintext password, the users real name, a comment, a list of share names that the user is allowed

to connect to, flags to indicate if the account is an administrator or guest account, and a home directory path. The user name and either the MD4 or plaintext password fields should be provided, all other fields are optional.

6.2 CifsAuthenticator

The *CifsAuthenticator* base class provides the LanMan and NTLM encryption algorithms required by the SMB server.

There are several abstract methods in the CifsAuthenticator class:-

- *getAuthContext*
- *authenticateUser*
- *authenticateShareConnect*

There is also an *initialize(NameValueList args)* method that is called from the ServerConfiguration implementation when the CifsAuthenticator class is instantiated.

The *authenticateUser* method is used to authenticate a user when a new session is established to the SMB server. The client sends a hashed version of the password using a challenge key returned in the initial negotiate response SMB.

The LanMan or NTLM encryption method is used to generate the hashed password depending upon the SMB dialect that has been negotiated with the client and the client operating system.

The initial SMB session setup phase has the following steps:-

- Client sends negotiate SMB with a list of SMB dialects that it supports.
- Server responds with the highest available SMB dialect from the client list and enabled list of server SMB dialects. The negotiate response also contains the 8 byte challenge response.
- Client sends session setup SMB with username and hashed password details.
- Server responds with success status or logon failure type status.

6.2.1 CifsAuthenticator Methods

This section details the CifsAuthenticator methods that must be implemented.

6.2.1.1 getAuthContext

```
AuthContext getAuthContext(SMBSrvSession sess);
```

The *getAuthContext()* method is called to generate the challenge key, or other context, required by the negotiate response. The default implementation generates an NTLanManAuthContext object that contains a random challenge key, which is then stored in the *SMBSrvSession* so that it can be retrieved when the session setup request is received.

6.2.1.2 authenticateUser

```
int authenticateUser(ClientInfo client, SMBSrvSession sess, int alg);
```

The authenticateUser() method is called when the session setup request is received with the username and hashed password.

The ClientInfo object contains the username, hashed passwords, domain, operating system and remote TCP/IP address of the client.

The client may specify two hashed passwords, one using the ASCII password and one using the Unicode password.

The authenticateUser() method should return one of the following values (declared in the Authenticator base class):-

- AUTH_ALLOW
  User logged on successfully
- AUTH_GUEST
  User logged on with guest access
- AUTH_DISALLOW
  User not allowed to logon
- AUTH_BADPASSWORD
  Invalid password
- AUTH_BADUSER
  Invalid user

An error status (AUTH_DISALLOW, AUTH_BADPASSWORD and AUTH_BADUSER) will result in a logon failure/access denied status being returned to the client.

To extend the ClientInfo object you can implement your own ClientInfoFactory to create the extended objects.

Use the ClientInfo.setFactory(ClientInfoFactory) method to override the default object factory.

6.2.1.3 authenticateShareConnect

```
int authenticateShareConnect(ClientInfo info, SharedDevice share,
                             String shrPwd,
                    SMBSrvSession sess);
```

The *authenticateShareConnect()* method is called when the SMB server receives a tree connect request to connect to a particular share.

The client connection will have already been allowed to access the server by the *authenticateUser()* method. The share connect method can be used to allow/disallow access to particular shares depending upon the logged in user.

If the user is allowed to access the specified share the *authenticateShareConnect()* method should return a file permission to indicate if the user has read or write access to the share:-

- FilePermission.ReadOnly

- FilePermission.Writeable

If the user is not allowed to access the share the *authenticateShareConnect()* method should return one of the following status values (declared in the Authenticator base class):-

- AUTH_DISALLOW
- AUTH_BADPASSWORD
- AUTH_BADUSER

The SMB server component will return an AccessDenied error status to the client if the user is not allowed to access the share.

If the *authenticateUser()* method handles all of the access control in your Authenticator implementation the *authenticateShareConnect()* method should return a value of *FilePermission.Writeable*.

7 Share Mapper

The ShareMapper interface is used by the core JLAN Server to locate a shared device when a client makes a connection to a particular virtual filesystem. The share device may be returned from a static list of available virtual filesystems or created on demand.

The ShareMapper interface can also be used to implement virtual hosts. The SMB server component can announce itself so that it appears as a number of hosts under Network Neighborhood when the NetBIOS protocol is enabled. The host that the client is connecting to is passed to the ShareMapper in the findShare() method.

The ShareMapper has the following methods:-

- initializeMapper()
- getShareList()
- findShare()
- deleteShares()
- closeMapper()

7.1 ShareMapper Methods

This section details the ShareMapper methods that must be implemented, or may be overridden.

7.1.1 initializeMapper

```
void initializeMapper(ServerConfiguration config, NameValueList params)
    throws InvalidConfigurationException;
```

The *initializeMapper* method is called after the ShareMapper is instantiated. The *params* list contains the configuration parameters for the ShareMapper.

7.1.2 getShareList

```
SharedDeviceList getShareList(String host, SrvSession sess);
```

The *getShareList* method returns a list of shared devices from the static list of shares plus any dynamic shares that have been defined for the specified session.

7.1.3 findShare

```
SharedDevice findShare(String tohost, String name, int typ, SrvSession sess,
                       boolean create)
    throws Exception;
```

The *findShare* method is used to search the available list of shared devices for a share that matches the required name and share type.

The available share types are defined in the ShareType class:-

- DISK
- PRINTER

- NAMEDPIPE
- ADMINPIPE
- UNKNOWN

In the current JLAN Server the findShare method will only receive requests for shares of type *DISK* or *UNKNOWN*. The *UNKNOWN* share type indicates that the client used the wildcard share type in the connection request.

The findShare method may be called when a client is connecting to a particular virtual filesystem or when a share information request has been received from the client. The *create* flag indicates whether the request is a client connecting to a virtual filesystem (true) or the client is requesting information on a virtual filesystem (false)

7.1.4 deleteShares

```
void deleteShares(SrvSession sess);
```

The *deleteShares* method is called when a session is closing down to delete any dynamic shares that may have been created for the specified session.

7.1.5 closeMapper

```
void closeMapper();
```

The *closeMapper* method is called when the JLAN Server is closing down to allow the ShareMapper to release any allocated resources.

8 Access Control Manager

The access control manager and associated rule implementations provide the per share access control mechanism.

Access controls may be configured per share or globally to be applied to all shared filesystems that do not have any access controls defined.

The *DefaultAccessControlManager* class is an implementation of the *org.alfresco.jlan.server.auth.acl.AccessControlManager* interface. If the access control manager class is not configured via the ServerConfiguration class then the DefaultAccessControlManager will be used.

The DefaultAccessControlManager includes a number of access control rule types to control access to a shared filesystem based upon username, protocol type, domain name or TCP/IP address.

New rule types may be added to the default set of rule types or the default rule types can be overridden via the server configuration.

8.1 AccessControlManager Interface

This section details the AccessControlManager methods.

8.1.1 initialize

```
void initialize(ServerConfiguration config, NameValueList params)
    throws InvalidConfigurationException;
```

The *initialize* method is called after the AccessControlManager is instantiated.

The *params* list contains the configuration parameters for the AccessControlManager.

8.1.2 checkAccessControl

```
int checkAccessControl(SrvSession sess, SharedDevice device);
```

The *checkAccessControl* method is called by the various protocol handlers to check if the client has access to the shared filesystem, and determine the access type.

The method should return one of the following constants from the *AccessControl* class :-

- AccessControl.None
  Client is not allowed to access the shared filesystem.
- AccessControl.ReadOnly
  Client has read-only access to the shared filesystem.
- AccessControl.ReadWrite
  Client has full access to the shared filesystem.

8.1.3 filterShareList

```
SharedDeviceList filterShareList(SrvSession sess, SharedDeviceList shares);
```

The *filterShareList* method is called by the various protocol handlers to generate a list of shared filesystems that are visible to the client.

The shared filesystem list is filtered by removing filesystems that return a status of *AccessControl.None*.

When a SMB/CIFS user browses to the JLAN Server shared filesystems that return an access status of AccessControl.None will not be listed under the host.

If an NFS client attempts to mount a shared filesystem which it does not have access to a status will be returned to indicate that the mount does not exist.

8.1.4 addAccessControlType

```
void addAccessControlType(AccessControlParser parser);
```

Adds a new access control rule type to the list of available rule types, or can also be used to override the default rule types.

8.1.5 createAccessControl

```
AccessControl createAccessControl(String type, NameValueList params)
    throws ACLParseException, InvalidACLTypeException;
```

Creates a new access control rule instance of the type *type* using the parameter list *params*.

An *ACLParseException* should be thrown if the parameter list is invalid.

An *InvalidACLTypeException* should be thrown if the type name is not a valid access control rule type.

8.2 Access Control Rules

To create a new access control rule requires two classes, an *AccessControlParser* class to parse and validate the input parameters and an *AccessControl* class that contains the value(s) to be checked and the rule checking logic.

The *AccessControlParser* class provides the access control type name to the *AccessControlManager* implementation and creates an instance of the *AccessControl* after parsing the parameter list.

The access control rule must be registered with the access control manager via the *addAccessControlType()* method.

8.2.1 AccessControlParser Class

The AccessControlParser class is an abstract class, the following methods must be implemented.

8.2.1.1 getType

```
String getType();
```

Returns the unique access control rule type name.

The following table details the default rule types used by the DefaultAccessControlManager. Using these names will override the default rule implementation with your own implementation.

Type	Rule Class
address	IpAddressAccessControl
domain	DomainAccessControl
protocol	ProtocolAccessControl
user	UserAccessControl

8.2.1.2 createAccessControl

```
AccessControl createAccessControl(NameValueList params)
    throws ACLParseException;
```

Parse and validate the parameters and create a new *AccessControl* instance.

If the parameter list is invalid an *ACLParseException* should be thrown.

8.2.2 AccessControl Class

The AccessControl class is an abstract class, the following method must be implemented.

8.2.2.1 allowsAccess

```
int allowsAccess(SrvSession sess, SharedDevice share,
                 AccessControlManager mgr);
```

Runs the access control logic and returns one of the following status values to indicate the access level allowed :-

- AccessControl.None
  Client does not have access to the shared filesystem.
- AccessControl.ReadOnly
  Client has read-only access to the shared filesystem.
- AccessControl.ReadWrite
  Client has full access to the shared filesystem.

The access control logic can use values from the SrvSession to determine if the client has access, such as the username for a CIFS session.

The SrvSession type can be checked if the rule only applies to a particular protocol by using the *instanceof* operator.

9 Quota Manager

The *QuotaManager* interface is designed to provide per shared filesystem disk quota management. The interface is designed to allow either a global quota for the filesystem or per user quotas.

The quota manager is attached to the DiskDeviceContext. The quota manager class used is controlled by the DiskInterface implementation as it is closely tied to the filesystem implementation.

The *org.alfresco.jlan.smb.server.disk.jdbc.JDBCQuotaManager* is a reference implementation of a global quota manager.

9.1 QuotaManager Interface

The *org.alfresco.jlan.server.filesys.quota.QuotaManager* interface provides the quota management interface.

9.1.1 startManager

```
void startManager(DiskInterface disk, DiskDeviceContext ctx)
    throws QuotaManagerException;
```

Called to start the quota manager during the initialization of the filesystem.

9.1.2 stopManager

```
void stopManager(DiskInterface disk, DiskDeviceContext ctx)
    throws QuotaManagerException;
```

Called to stop the quota manager during the closing of the filesystem.

9.1.3 allocateSpace

```
long allocateSpace(SrvSession sess, TreeConnection tree, NetworkFile file,
                   long alloc)
    throws IOException;
```

Allocate a block of space to the specified file.

The returned value can be the actual allocated block size or an encoded value that identifies the block.

9.1.4 releaseSpace

```
void releaseSpace(SrvSession sess, TreeConnection tree, int fid, String
path,                       long alloc)
    throws IOException;
```

Release space back to the available free space on the filesystem.

9.1.5 getAvailableFreeSpace

```
long getAvailableFreeSpace();
```

Return the total free space available on the filesystem, in bytes.

9.1.6 getUserFreeSpace

```
long getUserFreeSpace(SrvSession sess, TreeConnection tree);
```

Return the free space available to the user/session on the filesystem.

If the quota manager does not implement per user/session quotas the total available free space value should be returned.

10 ONC/RPC Authenticator

The ONC/RPC authentication is provided by a class that implements the *RpcAuthenticator* interface (in the *org.alfresco.jlan.oncrpc package*).

There is a default RpcAuthenticator implementation, *DefaultRpcAuthenticator*, that is used if an authenticator is not specified in the server configuration. The default RPC authenticator supports the null and Unix RPC authentication mechanisms allowing any group/user to access the JLAN Server.

The RpcAuthenticator interface has the following methods :-

- initialize
- getRpcAuthenticationTypes
- authenticateRpcClient
- getRpcClientInformation

10.1 RpcAuthenticator Methods

This section details the RpcAuthenticator methods.

10.1.1 initialize

```
void initialize(ServerConfiguration config, NameValueList params)
     throws InvalidConfigurationException;
```

The initialize method is called from the ServerConfiguration implementation when the RpcAuthenticator class is instantiated.

The params argument contains configuration parameters for the RpcAuthenticator implementation.

10.1.2 getRpcAuthenticationTypes

```
int[] getRpcAuthenticationTypes();
```

Return an array of authentication types that the RPC authenticator supports.

The available types are in the org.alfresco.jlan.oncrpc.AuthType class, and are listed below :-

- Null
- Unix
- Short
- DES

10.1.3 authenticateRpcClient

```
Object authenticateRpcClient(int authType, RpcPacket rpc)
    throws RpcAuthenticationException;
```

The *authenticateRpcClient()* method is called to authenticate every RPC request made to the various RPC servers (such as mount and NFS).

The *authType* parameter specifies the authentication type from the RPC.

The *rpc* parameter contains the RPC request packet. The RpcPacket class has methods for accessing the credentials area of the RPC plus methods for unpacking various data types from the request.

The following table lists the RpcPacket methods that may be useful.

Method	Description
getCredentialsType()	Return the credentials block type.
getCredentialsLength()	Return the credentials block length.
positionAtCredentialsData()	Position the buffer pointer at the credentials data (after the type and length fields).
unpackInt()	Unpack an integer value from the current buffer position and update the position.
unpackByteArray(byte[])	Unpack a byte array at the current buffer position and update the position. Use the byte[] length to determine the number of bytes to unpack.
unpackByteArrayWithLength(byte[])	Unpack a byte array at the current buffer position that has a length value and update the position.
unpackIntArray(int[])	Unpack an array of integer values at the current buffer position and update the position. Use the int[] length to determine the number of integers to unpack.
unpackLong()	Unpack a long (64 bit) value from the current buffer position and update the position.
unpackString()	Unpack a string that has a length value, and update the current position.

The method must return an Object which will be used by the RPC server to identify the associated session object.

The DefaultRpcAuthenticator implementation returns an Integer object that uses the client TCP/IP address as session key object for null authentication and a Long object that uses the client TCP/IP address, group id and user id as the session key object for Unix authentication.

10.1.4 getRpcClientInformation

```
ClientInfo getRpcClientInformation(Object sessKey, RpcPacket rpc);
```

The *getRpcClientInformation()* method is called to create the client information object that is associated with the session. The method is called when an associated session is not found for the session key returned by the *authenticateRpcClient()* method.

The *sessKey* parameter is the Object that was previously returned by the *authenticateRpcClient()* method.

The *rpc* parameter is the RPC request packet for the current request.

The DefaultRpcAuthenticator implementation returns a ClientInfo object with the client TCP/IP address set for the null authentication type, and returns a ClientInfo object with the client name, group id, user id and additional groups list for the Unix authentication type.

11 FTP Authenticator

The FTP Server authentication is provided by a class that implements the FTPAuthenticator interface (in the *org.alfresco.jlan.ftp* package).

There is a default FTPAuthenticator implementation, *LocalAuthenticator*, that is used if an authenticator is not specified in the server configuration. The LocalAuthenticator uses the UsersInterface to get the user account details.

The FTPAuthenticator interface has the following methods :-

- initialize
- authenticateUser

11.1 FTPAuthenticator Methods

This section details the FTPAuthenticator methods.

11.1.1 initialize

```
void initialize( ServerConfiguration config, ConfigElement params)
                  throws InvalidConfigurationException;
```

The initialize method is called from the ServerConfiguration implementation when the FTPAuthenticator class is instantiated.

The params argument contains configuration parameters for the FTPAuthenticator implementation.

11.1.2 authenticateUser

```
boolean authenticateUser( ClientInfo cInfo, FTPSrvSession sess);
```

Authenticate the user, return *true* if the logon is successful, else *false* to prevent the user logging on.

The *cInfo* parameter contains the details of the user that is logging on, including the user name and plaintext password.

The *sess* parameter contains the FTP session, that has details of the socket connection.

12 File State Cache and Clustering

The file state cache is a component that can be used by any filesystem to increase performance by caching various bits of file information, and can also provide additional features such as byte range locking and oplock support which help with client compatibility.

The file state cache can also allow the Alfresco JLAN Server to be run in a clustered configuration where multiple hosts co-ordinate access to the same filesystem(s).

There are two file state cache implementations included in the JLAN Server kit, standalone and clustered. You can also implement your own file state cache if required by extending the FileStateCache abstract class (in the *org.alfresco.jlan.server.filesys.cache* package).

The clustered state cache implementation uses the Hazelcast library to provide the underlying clustering support as it includes classes for a distributed hash map and messaging between nodes, as well as cluster initialization and failover.

12.1 File State Cache Basics

The file state cache implementations included in the Alfresco JLAN Server kit store file state information using a normalized version of the filesystem relative path as the key with a FileState based class to hold the file details.

The state cache holds a file state for all open files, plus files that were recently open, and may optionally have states for files that do not exist. Caching states for files that do not exist can be useful when handling requests from CIFS clients, for example the desktop.ini file is requested regularly for each folder that a user accesses.

The base file state class holds details of a particular file or folder on the filesystem, there will only be one file state cache entry for a given path. The file state contains the shared details about the file or folder, such as the file status (whether it exists, if it is a file or folder), the current open count, shared access mode, active byte range locks, oplock details, current file size and allocation size, latest creation/modification/change timestamps, retention expiry date/time, and the expiry time for the file state.

A state cache implementation will usually extend the base FileState class. A filesystem may attach named attributes to a file state. For example, the database filesystem uses this mechanism to cache the file information object for a file/folder to avoid excessive database requests. Attributes are implemented in the base FileState class so all implementations will support this feature.

A file state cache reaper thread is responsible for periodically calling the state cache expiry checker method for each filesystem that has registered a state cache. File states are removed from the cache when the file open count is zero and the expiry time has passed. A FileStateListener may be registered with a state cache to allow additional expiry logic to be implemented.

12.2 File Access Tokens

Using the file state cache requires the filesystem to use file access tokens. A file access token is granted by the file state cache when a file/folder is opened or created. The file access token should be granted before the NetworkFile based object is created. When the file/folder is closed, or the open/create fails, the access token must be released.

The file access token should be stored on the associated NetworkFile based class, via the setAccessToken() method.

12.2.1 Grant File Access

```
FileAccessToken grantFileAccess(FileOpenParams params, FileState fstate,
                                int fileSts)
    throws FileSharingException, AccessDeniedException;
```

The file state cache grantFileAccess() method is responsible for performing access mode checks, such as checking for exclusive access when the file is already open by another client, and may perform oplock checks. If a file access token is granted the current file open count will be incremented.

In the clustered file state cache the oplock check is done during the grantFileAccess() call as this saves several remote method calls.

12.2.2 Release Access Token

```
int releaseFileAccess(FileState fstate, FileAccessToken token);
```

The file state cache releaseFileAccess() method is responsible for decrementing the open file count for the file/folder. If the open count is now zero then the access mode and owner process id will be reset to the default values.

12.3 Using The File State Cache

Using the file state cache within your filesystem driver can improve the overall performance considerably, and can also provide additional functionality such as byte range locking and oplock support which help with application compatibility.

As with any caching mechanism one of the most important requirements is that of cache consistency. There are also a number of other requirements when using the file state cache in order to be compatible with the clustered state cache implementation. The following sections outline the requirements for the various filesystem classes.

12.3.1 DiskInterface Requirements

This section details the file state cache requirements for the main filesystem implementation that extends the DiskInterface interface.

12.3.1.1 fileExists

The fileExists method should use the cached status from the file state cache when a file state is available for the specified path, via the FileState.getFileStatus() method.

If a file state does not exist for the path then the file status should be determined and a state cache entry created for the path. The file status value indicates if a file exists, folder exists or the path does not exist. The FileStatus class contains the available values.

It can be useful to cache not exists status values for paths as Windows CIFS clients in particular can poll for the same file many times, such as the desktop.ini file. It is not necessary to cache the not exists states though.

12.3.1.2 getFileInformation

The getFileInformation method can optionally use the file state cache by attaching the FileInfo object to the file state as an attribute, using the FileState.addAttribute(String,Object) method.

If a file state exists for the path and the file information attribute is attached then the cached value can be returned.

If the file information is cached in this way it is important to update the cached values when there are changes to the file, such as file writes, truncate/resize, set file information, timestamp updates.

12.3.1.3 openFile

In order to perform access mode and oplock checks before a file is opened the file state cache grantFileAccess(...) method must be called to allocate a file access token. In a clustered environment the access checks and oplocks will be enforced across all nodes within the cluster.

The file access token must be attached to the associated NetworkFile based class via the setAccessToken() method.

If the file open fails the file access token must be released via the file state cache releaseAccessToken() method.

12.3.1.4 createFile/createDirectory

In order to perform access mode and oplock checks before a file/folder is created the file

state cache grantFileAccess(...) method must be called to allocate a file access token. The initial file status should be set to not exist.

The file access token must be attached to the associated NetworkFile based class via the setAccessToken() method.

Once the file/folder has been created the file state status should be changed to indicate that the file/folder now exists, via the file state setFileStatus() method. A reason code of file created or folder created should also be specified.

If the file/folder create fails the access token must be released via the file state cache releaseAccessToken() method.

12.3.1.5 deleteFile/deleteDirectory

When a file/folder is deleted the file state status should be set to not exists with a reason code of file deleted/folder deleted, via the file state setFileStatus() method.

If file state file ids are being used the file id should be reset to the unknown value, via the file state setFileId() method using either a value of FileState.UnknownFileId.

Any attributes that have been attached to the file state should be removed, such as file information.

The deleteFile() and deleteDirectory() methods work on paths rather than open files/folders. An open file/folder may be marked for delete on close, in this case the closeFile() method should have released the file access token before the file/folder is deleted.

12.3.1.6 closeFile

When a file/folder is closed the associated file access token must be released by calling the file state cache releaseFileAccess() method.

12.3.1.7 renameFile

When a file/folder is renamed the file state cache renameFileState() method must be called. If a folder is renamed all file states that are on paths below the renamed folder will need to be renamed or removed from the file state cache. The renameFileState() method will take care of updating any associated file states.

12.3.1.8 setFileInformation

If the file information is being cached via a file state attribute the cached information should be updated to make sure it is in sync, or removed so that it gets cached with the latest version.

12.3.1.9 startSearch

For non-wildcard searches where the details of a single file or folder will be returned it is possible to use the cached file information for the search by using the CachedSearchContext class.

12.3.1.10 writeFile/truncateFile

If the write or truncate call changes the file size then any cached values in the file state or a cached file information object should also be updated.

The file state change date time and allocation size should also be updated.

Write requests can truncate a file by using a zero length write. Truncate requests can extend or truncate the size of the file.

12.3.1.11 readFile/flushFile/seekFile

These methods do not need to use the file state cache.

12.3.1.12 getLockManager/getOplockManager

The file state based lock/oplock manager that was created during filesystem initialization should be returned by these methods. The lock/oplock manager is stored as part of the disk device context that is associated with the filesystem driver.

The disk device context can be accessed via the TreeConnection parameter :-

```
    DiskDeviceContext diskCtx = (DiskDeviceContext) tree.getContext();
    return diskCtx.getLockManager();
or
    return diskCtx.getOplockManager();
```

12.3.2 DiskDeviceContext Requirements

This section details the file state cache requirements for the disk device context that is associated with a filesystem driver.

The DiskDeviceContext based class should implement the FileStateCacheListener interface so it receives callbacks as the file state cache initializes and shuts down. The clustered file state cache can take some time to initialize during which time the filesystem should not be accessible, as it will not be able to find/add/update file states until the file state cache is running.

The FileStateCacheListener interface has three methods :-

```
    public void stateCacheInitializing();

    public void stateCacheRunning();

    public void stateCacheShuttingDown();
```

12.3.2.1 Initialization

During initialization of the disk device context the requires state cache flag should be set to true via the setRequiresStateCache(true) method. This indicates to the startup code that this filesystem requires a file state cache, if a file state cache has not been configured an error will be output and the server will not start.

12.3.2.2 startFilesystem

If the file state based lock/oplock manager is to be used by the filesystem it should be created in the startFilesystem() method.

The file state lock/oplock manager can either create a seperate thread for lock timeout handling or it can use the global thread pool that other parts of the file server use.

See the DBDeviceContext.startFilesystem() method for sample code.

12.3.2.3 stateCacheInitializing

Whilst the file state cache is initializing the filesystem should be marked as not available by calling the DiskDeviceContext.setAvailable(false).

12.3.2.4 stateCacheRunning

Once the file state cache signals that it is running the filesystem can be made available, via the DiskDeviceContext.setAvailable(true) method.

12.3.2.5 stateCacheShuttingDown

Indicates that the file state cache is shutting down so the filesystem must be marked as unavailable, via the DiskDeviceContext.setAvailable(false) method.