ChatGPT

Umbau-Spezifikation: Einheitliche FileService-Architektur für FTP und lokale Dateien

Zielstruktur und Architekturprinzipien

Die aktuelle Implementierung arbeitet direkt mit `FTPClient` aus Apache Commons Net und verwaltet den Verbindungszustand manuell (z. B. via `CWD`). Dies führt zu Problemen bei Lesezeichen (Bookmarks) und parallelen Dateioperationen. Ziel ist eine **vereinheitlichte FileService-Architektur**, die lokale Dateisysteme (Windows/Linux) und FTP/FTPS-Server über eine gemeinsame Schnittstelle abstrahiert.

Apache Commons VFS (Virtual File System) wird als Kern der neuen Architektur eingesetzt, da es einen einheitlichen API-Zugriff auf unterschiedliche Dateisysteme bietet [1]. Dadurch kann derselbe Code für lokale Dateien, Samba-Shares, FTP/FTPS und weitere Protokolle verwendet werden [1]. Insbesondere unterstützt VFS von Haus aus FTP und FTPS über ein einheitliches URI-Schema (`ftp://…`, `ftps://…`) [2]. Die Nutzung von VFS gewährleistet konsistente Dateioperationen (Auflisten, Lesen, Schreiben etc.) unabhängig vom physikalischen Speicherort.

Jede Operation der FileService-Schnittstelle soll **zustandslos** sein: Es gibt kein persistentes *Working Directory* pro Session. Stattdessen wird jede Methode mit einem vollständigen (absoluten) Pfad aufgerufen – die Anfrage startet logisch immer am Wurzelpfad des jeweiligen Dateisystems. Im Hintergrund wird jedoch die **FTP-Verbindung offen gehalten**, um Performanceprobleme durch ständiges Neuverbinden zu vermeiden. Dies wird erreicht, indem ein gemeinsamer `FileSystemManager` (von VFS) wiederverwendet wird. So können offene Verbindungen durch VFS-Caching erneut genutzt werden, anstatt für jede Operation neu aufgebaut zu werden [3]. Der `FileSystemManager` wird typischerweise nur einmal initialisiert (Singleton bzw. zentral verwaltet) und nicht nach jeder Aktion geschlossen, damit bestehende Verbindungen nicht unnötig getrennt werden [3].

Die **Schichtenarchitektur** des Designs orientiert sich an SOLID-Prinzipien und sauberer Trennung der Verantwortlichkeiten:

- **Präsentation/UI-Schicht:** Bestehende UI-Klassen (z. B. `MainFrame`, `ConnectMenuCommand`) interagieren nur noch mit einer abstrakten `FileService`-Schnittstelle, anstatt direkt FTP-spezifische Logik aufzurufen. Die UI bleibt somit unabhängig von der konkreten Implementierung des Dateizugriffs.
- **Service-Schicht:** Die `FileService`-Implementierung kapselt sämtliche Datei-Operationen (lokal und remote). Sie nutzt intern Apache Commons VFS, um Dateisystemzugriffe durchzuführen. Die Service-Methoden sind *stateless* im Hinblick auf Pfade (keine Session-Zustände). Verbindungsdetails (z. B. FTP-Credentials) werden entweder beim Initialisieren des Service oder pro Anfrage konfiguriert, ohne dass die UI sich darum kümmern muss.
- **Integrations-/Infrastruktur-Schicht:** Apache Commons VFS fungiert als einheitlicher Integrationslayer zu verschiedenen Dateisystemen. VFS übernimmt die Protokoll-spezifische Kommunikation (FTP, FTPS, lokale Dateisystem-APIs etc.) und abstrahiert diese über das `FileObject`-Interface. Durch die Verwendung von VFS entfällt die direkte Abhängigkeit von Apache Commons Net im Applikationscode; stattdessen nutzt der VFS-FTP-Provider intern

weiterhin Commons Net (und kann z. B. MVS-spezifische Verzeichnislisten bereits interpretieren [4]).

- **Domain-Schicht:** Datenklassen wie `FileNode` (für Dateimetadaten) und `FilePayload` (für Datei-Inhalte) repräsentieren Dateiobjekte unabhängig vom tatsächlichen Dateisystem. Diese Klassen erlauben eine lose Kopplung, indem sie nur die notwendigen Informationen (Name, Pfad, Typ, Größe etc.) enthalten, ohne UI-Logik oder Protokolldetails. Sie folgen dem **Single-Responsibility-Prinzip** (eine Klasse für Metadaten, eine für Inhalte). Dadurch können sie leicht erweitert werden (Open-Closed-Prinzip), z. B. um weitere Attribute aufzunehmen, ohne existierende Logik zu brechen.

SOLID-Konformität:
Die geplante Struktur beachtet die SOLID-Prinzipien: - *Single Responsibility:* Jede Klasse hat eine klar umrissene Verantwortung. Beispielsweise kümmert sich `FileService` nur um Dateioperationen, während `FileNode` nur Daten trägt. Dadurch sind Änderungen (z. B. andere Art der Dateibeschaffung) auf einzelne Klassen begrenzt. - *Open/Closed:* Das System ist erweiterbar, ohne bestehende Klassen anzupassen. Neue Dateisystemtypen (etwa SFTP oder spezielle MVS-Handler) können durch neue `FileService`-Implementierungen oder VFS-Provider hinzugefügt werden, ohne die UI oder Kernlogik zu ändern. Die `FileService`-Schnittstelle bleibt dabei stabil. - *Liskov Substitution:* Die `FileService`-Implementierungen (für lokale Dateien, FTP usw.) sind austauschbar und erfüllen dieselben Verträge, sodass die UI nicht unterscheiden muss, mit welchem Dateisystem sie gerade arbeitet. - *Interface Segregation:* Die Schnittstellen werden so zugeschnitten, dass UI-Klassen nur die Methoden sehen, die sie benötigen. Beispielsweise könnte die `FileService` ggf. in kleinere Interfaces aufgeteilt werden (z. B. Lesbar, Schreibbar), falls das hilfreich ist – hier reicht jedoch ein zentrales Interface mit den benötigten Methoden. - *Dependency Inversion:* Die UI-Schicht hängt von Abstraktionen (FileService-Interface) ab, nicht von konkreten Implementierungen. Die konkrete Implementierung (`VfsFileService` o. ä.) wird z. B. via Factory oder Dependency Injection bereitgestellt. Dadurch ist die Oberflächenschicht unabhängig vom Infrastrukturdetail und leicht austauschbar oder mockbar für Tests.

Erweiterbarkeit (MVS z/OS): Besonderheiten von Mainframe-Dateisystemen (MVS/zOS), etwa **Record-basierte Dateien** oder **PDS-Mitgliedsnavigation**, werden durch die Architektur berücksichtigt. Apache Commons Net (das von VFS-FTP intern verwendet wird) enthält bereits Parser für z/OS-Verzeichnislisten [4], sodass grundlegende List- und Dateiinfos von MVS-Systemen korrekt ausgelesen werden können. Sollte erweiterte Funktionalität nötig werden – z. B. das Navigieren in PDS/Partitioned Datasets als "virtuelle Verzeichnisse" oder das spezielle Handling von Fixed/Variable Records – kann dies durch **dedizierte Provider-Klassen oder Adapter** erreicht werden. Denkbar ist etwa ein `MvsFileProvider`, der auf VFS aufsetzt und MVS-spezifische Kommandos (z. B. DSINFO, SITE-Befehle) ausführt, oder ein Adapter in der `FileService`-Implementierung, der die von VFS gelieferten Daten aufbereitet. Diese Erweiterungen können als zusätzliche Module implementiert und via Konfiguration eingebunden werden, ohne die allgemeine FileService-API zu verändern (Open/Closed-Prinzip). So bleibt die Architektur zukunftssicher und **für Erweiterungen vorbereitet**.

Schnittstellen und Klassenentwurf

Im Folgenden werden die zentralen Schnittstellen und Klassen der neuen Architektur beschrieben. Bestehende Klassen aus dem Altsystem (z. B. `FtpManager`, `FileContentService`) werden durch diese neuen Komponenten entweder **ersetzt** oder **umstrukturiert**, um die neue Architektur abzubilden. Wichtig ist, dass die UI-Klassen (z. B. Menü-Commands, MainFrame) nach dem Umbau

nahtlos die neue API verwenden können, ohne Funktionalität einzubüßen – die Anpassungen an der UI beschränken sich darauf, die neuen Methoden/Klassen aufzurufen.

- **FileService (Interface):**
  Definiert die abstrakte Schnittstelle für Dateioperationen. Über diese Schnittstelle greift die UI auf das Dateisystem (lokal oder remote) zu, ohne Details des Protokolls zu kennen. Alle Methoden sind so gestaltet, dass sie bei *jedem Aufruf vollständig definierte Parameter* erwarten (keine impliziten Zustände). Wichtige Methodensignaturen könnten sein:
- `List<FileNode> list(String path) throws FileServiceException` – Listet den Inhalt des Verzeichnisses unter dem gegebenen Pfad auf. Liefert eine Liste von `FileNode`-Objekten für Dateien/Unterverzeichnisse.
- `FilePayload readFile(String path) throws FileServiceException` – Lädt den Inhalt der Datei am gegebenen Pfad und gibt ihn in einem `FilePayload`-Container zurück (z. B. inklusive InputStream oder Byte-Array der Daten).
- `void writeFile(String path, FilePayload data) throws FileServiceException` – Schreibt den übergebenen Inhalt an den Dateipfad (z. B. zum Hochladen auf FTP oder Speichern lokal). Optionale Überladung: `writeFile(String path, InputStream dataStream)`.
- `void delete(String path) throws FileServiceException` – Löscht die Datei/das Verzeichnis am Pfad.
- `boolean createDirectory(String path) throws FileServiceException` – Legt ein neues Verzeichnis an.

Diese Methoden abstrahieren gemeinsame Operationen. Bei FTP-Pfaden wird intern eine FTP-Verbindung genutzt, bei lokalen Pfaden das lokale Dateisystem – für den Aufrufer besteht kein Unterschied. Die `FileServiceException` kapselt protokollspezifische Fehler (IOExceptions, FTPExceptions etc.), sodass die UI einheitliches Fehlerhandling durchführen kann.

Anmerkung: Für Verbindungsverwaltung kann das Interface auch Methoden enthalten wie `close()` (um z. B. Verbindungen gezielt zu schließen, falls die Anwendung dies erfordert), oder es kann einen Mechanismus zur Bereitstellung von Verbindungsparametern haben. Allerdings werden Credentials/Verbindungsdetails vorzugsweise bei der Instanziierung übergeben (siehe `VfsFileService` unten), sodass separate `connect()`-Methoden im Interface nicht zwingend nötig sind.

- **VfsFileService (Klasse, implementiert FileService):**
  Konkrete Implementierung der FileService-Schnittstelle unter Verwendung von Apache Commons VFS. Diese Klasse ist das Herzstück der neuen Architektur für Standard-Dateisysteme (lokal, FTP, FTPS etc.). Sie hält interne Ressourcen wie den `FileSystemManager` und ggf. Verbindungsoptionen (Credentials). Wesentliche Entwurfsdetails:
- Im Konstruktor oder mittels Initialisierungsmethode erhält `VfsFileService` die notwendigen **Verbindungsinformationen**. Für lokale Zugriffe genügen Standardwerte, für FTP/FTPS werden z. B. Host, Port, Benutzer und Passwort benötigt. Diese können entweder direkt im Pfad benutzt werden (nicht empfohlen wegen Klartext-Passwort) oder – besser – über einen `UserAuthenticator` gesetzt werden. Beispielsweise kann bei Erstellung eines `VfsFileService` für FTP ein `StaticUserAuthenticator` mit Benutzerdaten erzeugt und in `FileSystemOptions` registriert werden [5]. Diese Options und ggf. spezifische Konfiguration (wie `FtpFileSystemConfigBuilder.setUserDirIsRoot`) werden dann beim Aufruf von `resolveFile` verwendet. Die Klasse könnte auch Factory-Methoden bereitstellen, z. B. `VfsFileService.forLocal()` oder `VfsFileService.forFtp(host, user, pass)`, um die Erstellung mit korrekter Konfiguration zu erleichtern.
- **Hintergrund-Verbindung:** `VfsFileService` nutzt einen zentralen `FileSystemManager` (z. B. via `VFS.getManager()`). Dieser wird i. d. R. als Singleton verwaltet. Alternativ könnte

`VfsFileService` auch einen eigenen `DefaultFileSystemManager` instanziieren, doch in den meisten Fällen reicht der globale Manager. Wichtig: Es wird darauf geachtet, den Manager NICHT pro Operation zu schließen, damit VFS interne Verbindungen wiederverwenden kann [3]. Die Klasse kann z. B. beim Start einen separaten Thread oder Hook registrieren, um bei Programmende `closeFileSystem()` aufzurufen, falls nötig, aber während der Programmlaufzeit bleiben Dateisysteme geöffnet.

- **Methodenimplementierung:** Jede FileService-Methode wird mittels VFS implementiert. Beispiel `list(path)`:

  1. Ermitteln des VFS-URL-Schemas: Wenn `path` kein Schema enthält (z. B. `"C:/Ordner"` oder `"/home/user"`), kann die Implementierung automatisch `file://` voranstellen für lokale Pfade. Bei Pfaden, die mit "`ftp://`" oder "`ftps://`" beginnen, wird das Schema erkannt und ggf. um Credentials ergänzt (z. B. via hinterlegtem Authenticator in Options).
  2. Aufruf von `fsManager.resolveFile(resolvedPath, fileSystemOptions)` um ein `FileObject` zu erhalten. Dabei werden hinterlegte `FileSystemOptions` (mit Credentials oder anderen Settings) für FTP/SFTP übergeben.
  3. Wenn das `FileObject` ein Ordner ist (`fileObject.getType() == FileType.FOLDER`), werden mittels `fileObject.getChildren()` die Kind-Elemente abgerufen (als `FileObject[]`).
  4. Jedes `FileObject` der Kindliste wird in einen `FileNode` umgewandelt (siehe unten), indem dessen Metadaten gelesen werden: Name (`fileObject.getName().getBaseName()`), Pfad/URI (`fileObject.getName().getURI()`), Typ (Folder oder File), Größe (`fileObject.getContent().getSize()` falls unterstützt), Timestamp (`fileObject.getContent().getLastModifiedTime()`), etc.
  5. Die `FileObject`s sollten nach Gebrauch geschlossen werden (insbesondere bei Dateien, um Streams frei zu geben). Ordner-Objects können geöffnet bleiben, müssen aber nicht – VFS verwaltet einen Cache. Man kann vorsichtshalber am Ende `fileObject.close()` aufrufen, um Ressourcen freizugeben, wobei der FileSystem selbst offen bleibt.
  6. Die Liste der erzeugten `FileNode` wird zurückgegeben.

Ähnlich funktionieren `readFile` (mit `fileObject.getContent().getInputStream()`, Daten lesen, in `FilePayload` ablegen) und Schreib-Operationen (`fileObject.getContent().getOutputStream()` für `writeFile`, oder Nutzung von `FileUtil.copy()` innerhalb VFS). Transiente Objekte wie Streams werden in einem `try-with-resources` Block geöffnet, damit sie zuverlässig geschlossen werden. - **Spezialfälle & Fehlerbehandlung:** Falls Pfade ungültig sind oder Netzwerkfehler auftreten, fängt `VfsFileService` die entstehenden Exceptions (z. B. `FileSystemException`) ab und wirft sie als `FileServiceException` mit verständlicher Meldung weiter. Damit bleibt die Exception-Handling-Logik in der UI simpel. Zudem kann `VfsFileService` Logging betreiben, um Verbindungsaufbau, durchgeführte Operationen und Fehler zu protokollieren (hilfreich für Debugging). - **MVS-Unterstützung:** In `VfsFileService` können Haken vorgesehen werden, um erkannte MVS-Pfade speziell zu behandeln. Zum Beispiel: erkennt die Methode `list()` einen Pfad der Form `"//'Dataset.Name'"` (MVS Dataset-Notation), könnte sie einen speziellen MVS-Provider verwenden. Dies könnte aber auch außerhalb dieser Klasse durch einen eigenen Service erfolgen (siehe unten). Für den Großteil der Operationen genügt es jedoch, VFS normal zu verwenden, da der FTP-Server auf z/OS durch entsprechende List-Formate repräsentiert, welche Commons Net bereits aufschlüsselt [6]. Bei Bedarf könnte

`VfsFileService` die gelieferten `FileObject`-Namen analysieren und z. B. PDS-Mitglieder als separate `FileNode`-Typen kennzeichnen (etwa via Eigenschaft `isMember`).

- **FileNode (Klasse):**
  Repräsentiert ein **Dateielement** (Datei oder Verzeichnis) mitsamt Metadaten. Diese Klasse dient dazu, UI-neutral Informationen über Dateien anzuzeigen und Operationen darauf vorzubereiten. Wichtige Felder von `FileNode` könnten sein:

- `String name` – Name der Datei bzw. des Verzeichnisses (ohne Pfad).
- `String path` – Vollqualifizierter Pfad oder URI, der dieses FileNode eindeutig identifiziert. Für lokale Dateien evtl. ein absoluter Pfad (`C:\...\file.txt`), für FTP z. B. `"ftp://host/dir/file.txt"`. Dieser Pfad kann direkt wieder an FileService-Methoden übergeben werden.
- `boolean directory` – Kennzeichnet, ob es sich um ein Verzeichnis handelt (`true`) oder um eine Datei (`false`). Alternativ könnte man einen Enum `FileType` mit Werten wie FILE, FOLDER, LINK verwenden.
- `long size` – (optional) Dateigröße in Bytes, falls verfügbar. Bei Verzeichnissen oder nicht ermittelbar z. B. 0 oder -1.
- `Date lastModified` – (optional) Letztes Änderungsdatum.
- Weitere optionale Attribute: z. B. `String permissions` (bei Bedarf Darstellung von Zugriffsrechten), `String owner` etc., je nach unterstütztem Dateisystem.

Die `FileNode`-Klasse hat primär die Aufgabe eines **DTO (Data Transfer Object)** zwischen Service und UI. Sie enthält keine Verarbeitungslogik außer eventuell Hilfsmethoden (wie `isDirectory()` Getter). Sie wird typischerweise aus einem `FileObject` oder `java.io.File` erzeugt. Beispiel: Ein Konstruktor `FileNode(FileObject fo)` liest die oben genannten Eigenschaften aus dem `fo`. Für MVS könnte `FileNode` ein zusätzliches Feld `member` (für PDS-Mitgliedername) oder `recordFormat` (für Datasets) bekommen, falls gewünscht – dies aber erst, wenn wirklich benötigt, um die Komplexität gering zu halten.

- **FilePayload (Klasse):**
  Kapselt den **Inhalt einer Datei** oder einen Dateitransfer-Puffer. Während `FileNode` Metadaten beschreibt, enthält `FilePayload` die eigentlichen Daten einer Datei. Dies kann je nach Anwendungsfall unterschiedlich umgesetzt werden:
- Für Textdateien könnte `FilePayload` z. B. einen `String content` halten (inkl. Encoding) – hier muss aber auf Dateigröße geachtet werden.
- Allgemeiner: `InputStream inputStream` oder `byte[] data` zur Repräsentation binärer Inhalte. In Java 8 könnte man auch mit Streams arbeiten; vermutlich bietet ein InputStream die größte Flexibilität. `FilePayload` könnte eine Methode `getInputStream()` bereitstellen. Wenn die Datei klein ist, kann `FilePayload` auch direkt die Bytes im Speicher halten.
- Darüber hinaus kann `FilePayload` Meta-Informationen wie die Dateilänge oder den MIME-Typ enthalten, falls relevant (z. B. um im UI anzuzeigen oder für Transfers).

Verwendung: Die `readFile`-Methode des FileService liefert ein `FilePayload`. Intern öffnet `VfsFileService` dafür einen `InputStream` vom `FileObject` und packt ihn in ein `FilePayload`-Objekt. Im UI kann dann z. B. aus dem `FilePayload` ein Buffer gelesen oder ein Stream an einen Editor übergeben werden. Beim Schreiben umgekehrt: Die UI befüllt ein `FilePayload` mit zu sendenden Daten (oder reicht direkt einen Stream hinein), und `writeFile` nimmt dieses entgegen, öffnet einen OutputStream via VFS und schreibt die Daten.

- **Weitere Klassen / Anpassungen:**

- **FileServiceFactory (optional):** Ein Fabrikmodul, das abhängig vom Verbindungswunsch die richtige FileService-Instanz liefert. Beispielsweise könnte `FileServiceFactory.createForLocal()` eine lokale `VfsFileService` zurückgeben, während `FileServiceFactory.createForFtp(host, user, pass)` eine konfigurierte `VfsFileService` (mit FTP-Settings) liefert. Dies zentralisiert die Initialisierung und erleichtert ggf. die Einbindung von Abhängigkeiten (z. B. falls später ein DI-Container verwendet wird). Alternativ kann die Initialisierung auch direkt per Konstruktor der Implementierung erfolgen, dann ist eine Factory nicht zwingend nötig.
- **MvsFileServiceAdapter (optional, Erweiterung):** Für z/OS könnte ein spezieller Adapter bereitgestellt werden, der die `VfsFileService` dekoriert oder erweitert. Er könnte z. B. beim Auflisten erkennen, ob es sich um ein Partitioned Dataset handelt, und dann anstelle einer flachen Liste der Member evtl. zwei Ebenen erzeugen: Erst Dataset als Verzeichnis, dann Mitglieder als FileNodes. Solch ein Adapter würde das FileService-Interface implementieren und intern die Aufrufe an eine Basisklasse delegieren, ergänzt um MVS-spezifische Logik. Diese Klasse würde nur aktiviert, wenn z. B. die Verbindung ein MVS-Host ist (konfigurierbar über einen Parameter).
- **Übernahme bestehender Klassen:** Die bisherige `FtpManager`-Klasse lässt sich ablösen, da ihre Aufgaben (Verbindung aufbauen, CWD pflegen, Verzeichnisse holen) nun von `VfsFileService` übernommen werden. Ggf. verbleibende Hilfsfunktionen (z. B. Parsen von FTP-Pfaden oder Credential-Verwaltung) können in Utility-Methoden ausgelagert oder in die Service-Klasse integriert werden. `FileContentService` (falls bisher separate Klasse) kann in `FileService` aufgehen – bisherige Methoden zum Laden/Speichern von Dateien werden entsprechend im FileService abgebildet. Man könnte auch entscheiden, `FileContentService` in `FileService` umzubenennen und zu erweitern, um dem existierenden Code minimalinvasiv entgegenzukommen (dann würde `FileContentService` das FileService-Interface implementieren, intern aber VFS nutzen). Allerdings ist eine klare Trennung in ein neues Modul meist sauberer. Wichtig ist in jedem Fall, dass öffentliche Methoden in UI-Komponenten, die bisher `FtpManager` oder `FileContentService` genutzt haben, auf die neue API umgestellt werden. Falls das UI bisher z. B. `ftpManager.changeDirectory(x)` aufgerufen hat, wird daraus nun `fileService.list(x)` (mit absolutem Pfad) oder ähnliches. Durch die Vereinheitlichung entfällt in der UI die Unterscheidung zwischen lokalen und remote Operationen: Das UI entscheidet ggf. anhand der gewählten Verbindung, welche FileService-Implementierung aktiv ist, ruft aber stets die gleichen Methoden auf.

Zusammengefasst besteht die Zielstruktur aus einer **zentralen FileService-Abstraktion**, konkreten Implementierungen (hauptsächlich auf VFS basierend) und einfachen Datenklassen für Dateien und Inhalte. Diese Struktur ist **modular** und robust gegenüber Erweiterungen. Neue Dateisysteme oder Protokolle können durch zusätzliche Implementierungen oder Adapter ergänzt werden, ohne die Kernlogik oder UI ändern zu müssen. Bestehende UI-Elemente werden so angepasst, dass sie die neuen Schnittstellen nutzen, was ihre Abhängigkeit von technischen Details wie FTP-Sitzungen eliminiert.

Beispielcode für VFS-Einbindung

Um die Verwendung von Apache Commons VFS in der neuen FileService-Architektur zu verdeutlichen, folgen einige Codebeispiele. Diese demonstrieren sowohl den Zugriff auf einen FTP-Server als auch auf das lokale Dateisystem über das einheitliche API. Alle Beispiele sind Java-8-kompatibel und verwenden keine neueren Sprachfeatures. Kommentare im Code erläutern die Schritte (auf Englisch, Imperativstil).

Beispiel 1: Auflisten eines FTP-Verzeichnisses via VFS

```
FileSystemManager fsManager = VFS.getManager();
// Configure FTP options with credentials (avoid putting password in the URI)
StaticUserAuthenticator auth = new StaticUserAuthenticator(null, "ftpuser",
"ftpPass123");
FileSystemOptions opts = new FileSystemOptions();
DefaultFileSystemConfigBuilder.getInstance().setUserAuthenticator(opts,
auth);
// Optionally, set userDirIsRoot to false if we want absolute root access
instead of user home
FtpFileSystemConfigBuilder.getInstance().setUserDirIsRoot(opts, false);

// Resolve the root directory on the FTP server (connects using provided
credentials)
FileObject remoteRoot = fsManager.resolveFile("ftp://ftp.example.com/",
opts);

// List the children of the root directory
FileObject[] children = remoteRoot.getChildren();
System.out.println("Inhalt von " + remoteRoot.getName().getURI() + ":");
for (FileObject child : children) {
    // Print name and indicate if it's a directory
    String childName = child.getName().getBaseName();
    if (child.isFolder()) {
        System.out.println("[DIR] " + childName + "/");
    } else {
        System.out.println("[FILE] " + childName + " (" +
child.getContent().getSize() + " bytes)");
    }
}

// Clean up the FileObject (releases resources, connection stays open in
fsManager cache)
remoteRoot.close();
```

In diesem Code wird ein FTP-Verzeichnis mit Hilfe des VFS-API aufgelistet. Zunächst wird ein `FileSystemManager` beschafft und mit `StaticUserAuthenticator` die Anmeldung konfiguriert [5] (anstelle Benutzername/Passwort im Klartext-URI zu verwenden). Dann wird mittels `resolveFile` das Wurzelverzeichnis des FTP-Servers als `FileObject` geöffnet. Über `getChildren()` erhalten wir die Liste der Einträge, die wir als Verzeichnisse oder Dateien ausgeben. Nach der Nutzung wird `remoteRoot.close()` aufgerufen, um das `FileObject` zu schließen – die Verbindung selbst bleibt aufgrund des Connection-Pools von VFS im Hintergrund erhalten.

Beispiel 2: Zugriff auf lokale Dateien via VFS

```
FileSystemManager fsManager = VFS.getManager();
// Resolve a local directory (file:// scheme can be omitted for local files
on default file system)
FileObject localDir = fsManager.resolveFile("file:///C:/Daten/Projekte");
// List files and directories in the local directory
```

```
for (FileObject entry : localDir.getChildren()) {
    String name = entry.getName().getBaseName();
    // Determine entry type and output basic info
    if (entry.isFolder()) {
        System.out.println("[DIR]  " + name);
    } else {
        // Fetch size and last modified timestamp
        long size = entry.getContent().getSize();
        long lastMod = entry.getContent().getLastModifiedTime();
        System.out.println("[FILE] " + name + " - " + size + " bytes,
lastMod=" + lastMod);
    }
}
// Close the FileObject for the directory
localDir.close();
```

Hier wird ein lokales Verzeichnis über VFS ausgelesen. Durch das Schema `file://` (bzw. automatische Standardbehandlung) greift VFS auf das normale Dateisystem zu. Die Ausgabe zeigt, dass dieselben Methoden (`getChildren()`, `isFolder()`, `getContent().getSize()`, etc.) verwendet werden können, wie beim FTP-Zugriff – die Logik im Code bleibt **identisch**, nur der Pfad und das Schema unterscheiden sich. Dies unterstreicht den Vorteil der vereinheitlichten FileService-Architektur: die UI oder aufrufende Logik muss nicht mehr wissen, ob es sich um FTP oder eine lokale Datei handelt. VFS präsentiert eine einheitliche Sicht auf verschiedene Dateiquellen [1].

Beispiel 3: Verwendung der FileService-API (Konzeptuell)

Zum Abschluss ein fiktives Beispiel, wie der Aufruf aus Sicht der UI mit der neuen `FileService` aussehen könnte. Angenommen, wir haben eine FileService-Instanz (z. B. via Factory erzeugt), die mit einem bestimmten Ziel (lokal oder FTP) konfiguriert ist:

```
// Example: The service is already configured for a certain connection (local
or FTP)
FileService fileService = FileServiceFactory.createForFtp("ftp.example.com",
"ftpuser", "ftpPass123");
// List root directory
List<FileNode> items = fileService.list("/");
for (FileNode item : items) {
    if (item.isDirectory()) {
        System.out.println("[DIR] " + item.getName());
    } else {
        System.out.println("[FILE] " + item.getName() + " (" +
item.getSize() + " bytes)");
    }
}

// Read a file's content from FTP and save it locally
FilePayload payload = fileService.readFile("/documents/report.pdf");
try (InputStream in = payload.getInputStream();
     OutputStream out = new FileOutputStream("C:/Temp/report.pdf")) {
    byte[] buffer = new byte[4096];
```

```
    int read;
    // Copy data from the payload stream to local file
    while ((read = in.read(buffer)) != -1) {
        out.write(buffer, 0, read);
    }
}
// ... similarly, we could write a local file back to FTP using
fileService.writeFile(...)
```

In diesem fiktiven Snippet nutzt die UI die FileService, ohne sich um Details zu kümmern. Die FileService-Implementierung bestimmt anhand ihrer Konfiguration (hier via Factory als FTP-Service erzeugt) automatisch, dass die Pfade auf einem FTP-Server zu interpretieren sind. Die Methode `list("/")` liefert FileNode-Objekte des Stammverzeichnisses. Anschließend wird `readFile` benutzt, um eine Datei herunterzuladen – das zurückgegebene `FilePayload` wird hier verwendet, um die Datei auf die lokale Platte zu speichern. Beachte: Die UI muss sich weder um `FTPClient`-Befehle noch um Streams von Commons Net kümmern; all das erledigt die interne VFS-Nutzung in `fileService`. Würde `fileService` stattdessen auf eine lokale Basis konfiguriert, würde derselbe Code mit Pfaden des lokalen Dateisystems funktionieren. Diese Einheitlichkeit reduziert die Komplexität in der UI deutlich und behebt die ursprünglichen Probleme mit **zustandsbehafteten Pfaden** (CWD) vollständig.

Konkrete Umbau-Tasks (Schritt-für-Schritt)

Abschließend werden die Refactoring-Schritte aufgeführt, die notwendig sind, um von der bestehenden Implementierung zur beschriebenen Zielarchitektur zu gelangen. Diese Aufgaben können von einem Entwickler oder einem Coding-Agenten nacheinander umgesetzt werden:

1. **Apache Commons VFS integrieren:** Die Maven-Dependency für Apache Commons VFS2 (aktuelle Version, z. B. 2.10) ins Projekt aufnehmen. Sicherstellen, dass auch Apache Commons Net als transitive Abhängigkeit verfügbar ist (für FTP/FTPS). Das Projekt bauen und überprüfen, dass die Bibliothek geladen wird.

2. **FileService-Interface definieren:** Ein neues Interface `FileService` erstellen (in einem geeigneten Package, z. B. `com.example.fileservice`). Darin die methodischen Platzhalter für benötigte Operationen deklarieren: z. B. `list(String path)`, `readFile(String path)`, `writeFile(String path, FilePayload data)`, `delete(String path)`, `createDirectory(String path)` etc., jeweils mit geeigneten Exceptions. Dieses Interface bildet die zentrale Abstraktion für Dateioperationen.

3. **Datenklassen erstellen:** Die Klassen `FileNode` und `FilePayload` implementieren (ebenfalls in passendem Package, ggf. `com.example.fileservice.model`):

4. `FileNode`: Felder für Name, Pfad, Typ (Datei/Ordner), Größe, Datum etc. + Getter/Setter. Überschreibe `toString()` für Debug-Zwecke (z. B. Pfad oder Name zurückgeben). Evtl. Konstruktoren für bequemes Befüllen (z. B. `FileNode(name, path, isDirectory)`).
5. `FilePayload`: Felder je nach Design (z. B. `InputStream` oder `byte[]` + ggf. Länge). Methoden anbieten, um an die enthaltenen Daten zu kommen (`getInputStream()`, evtl. `getBytes()` wenn umgesetzt). Achte auf Java 8 Kompatibilität (keine Records verwenden).

Füge notwendig erscheinende Hilfsmethoden hinzu, z. B. statische Fabrikmethoden wie `fromBytes(byte[])` oder `fromFile(File)` je nach Use-Case.

6. Dokumentiere mittels Kommentaren (auf Englisch) kurz die Zweck der Klassen. Kompiliere und prüfe, dass die Klassen fehlerfrei sind.

7. **VfsFileService implementieren:** Erstelle die Klasse `VfsFileService` im Service-Package, die `FileService` implementiert.

8. Füge als Member einen `FileSystemManager` hinzu (initialisiert via `VFS.getManager()` entweder statisch oder im Konstruktor).
9. Falls FTP/FTPS unterstützt werden soll: Plane, wie Credentials verwaltet werden. Z.B. als Member `FileSystemOptions ftpOptions` mit vordefinierten Authenticator. Diese können im Konstruktor gesetzt werden, wenn entsprechende Parameter übergeben werden. Biete mehrere Konstruktoren an, z. B. einen für lokale Nutzung (kein Parameter, nur `FileSystemManager` setzen) und einen für FTP (Parameter host, user, pass -> erstellt authenticator).
10. Implementiere `list(String path)`: wie oben beschrieben, auflösen des Pfades via `resolveFile`, Kinder sammeln, in `FileNode` umwandeln. Achte auf Exception Handling: VFS-Exceptions in `FileServiceException` umwandeln.
11. Implementiere `readFile(String path)`: Öffne `FileObject file = fsManager.resolveFile(path, opts)`, hole InputStream, verpacke ihn in `FilePayload`. (Je nach Entscheidung: InputStream direkt in FilePayload belassen und dem Aufrufer überlassen zu schließen, oder Daten sofort lesen – bei großen Dateien lieber Stream geben).
12. Implementiere `writeFile(String path, FilePayload payload)`: Ähnlich, aber OutputStream öffnen. Unterstütze sowohl Byte-Daten als auch Streams aus dem Payload. Schließe OutputStream nach Schreibende.
13. Implementiere `delete(String path)` und `createDirectory(String path)`: Hier können VFS-Methoden `fileObject.delete()` bzw. `fileObject.createFolder()` genutzt werden.

14. Teste die VfsFileService-Methoden zunächst isoliert (z. B. in einem kleinen Main oder JUnit, falls möglich) mit einem bekannten lokalen Pfad und – falls ein Test-FTP-Server verfügbar – mit einem FTP-Pfad. Prüfe, ob die erwarteten Ergebnisse (korrekte Listings, Datei lesen/schreiben) erzielt werden.

15. **Vorbereitung der UI-Umstellung:** Identifiziere alle Stellen im UI und in Commands, wo bisher direkt FTP-spezifische Klassen verwendet werden:

16. Bspw. Aufrufe von `FtpManager.connect()`, `FtpManager.changeDirectory()`, `FtpManager.getCurrentDir()` oder `FileContentService.loadFile()` etc.

17. Ermittle, welche Funktionen die UI von diesen Klassen benötigt (z. B. Verzeichnisinhalt holen, Datei anzeigen, hoch-/runterladen, etc.). Diese Funktionen sollen durch FileService-Methoden ersetzt werden. Notiere diese Stellen für die Umbau-Schritte.

18. **Integration der FileService in UI/Commands:** Ersetze in den UI-Klassen die Verwendung der alten Klassen durch die neue `FileService`-Schnittstelle:

19. Instanziiere ggf. zu Programmstart oder beim Connect-Dialog eine passende FileService-Implementierung. Z.B.: Wenn der Benutzer eine FTP-Verbindung herstellt, erstelle `fileService = FileServiceFactory.createForFtp(host, user, pass)` und bewahre

diese Instanz an zentraler Stelle auf (im MainFrame oder einem Session-Objekt). Für lokalen Modus analog (ggf. eine lokale FileService-Instanz vorrätig halten).

20. Passe Methoden an: Wo früher `ftpManager.changeDirectory(path)` genutzt wurde, wird jetzt `fileService.list(path)` aufgerufen, und das Ergebnis (List<FileNode>) verwendet, um die UI (Dateibaum, Tabellen etc.) zu füllen. Wo `fileContentService.loadFile(path)` war, wird `fileService.readFile(path)` genutzt, um einen `FilePayload` zu erhalten, aus dem dann der Inhalt gelesen und z. B. in einem Textfeld angezeigt wird.
21. Entferne oder kommentiere alte Aufrufe, die nicht mehr benötigt werden (z. B. kein separates `ftpManager.cwd` mehr nötig, da jeder List-Aufruf den vollständigen Pfad erhält).

22. Achte darauf, dass Lesezeichen/Bookmarks jetzt einfach als Pfad-Strings gehandhabt werden können – da kein interner Zustand mehr vorgehalten wird, kann ein Bookmark einfach den absoluten Pfad speichern. Beim Aufrufen eines Bookmarks ruft die UI `fileService.list(bookmarkPath)` auf der aktiven Service-Instanz auf. Falls Bookmarks auch zwischen unterschiedlichen Verbindungen genutzt werden, muss die Applikation sicherstellen, dass die richtige FileService (mit der passenden Connection) verwendet wird.

23. **Refactoring der alten Klassen:** Sobald die UI umgestellt ist, können die alten Klassen wie `FtpManager` und `FileContentService` entweder **entfernt** oder **umgebaut** werden:

24. Option A (empfohlen): **Klassen entfernen und auf neue Architektur umstellen.** Wenn möglich, alle Verwendungen ersetzen und die Klassen komplett löschen, um Verwirrung zu vermeiden. Sollten bestimmte Hilfsfunktionen darin enthalten sein (z. B. Parsing von Pfaden oder Konvertierung von Line Endings), überführe diese Utilities in neue geeignete Orte (z. B. als statische Helfer in einer `FileUtils`-Klasse, falls nötig).
25. Option B: **Klassen beibehalten, aber intern auf FileService umleiten.** Falls aus Gründen der Rückwärtskompatibilität einige Teile der App weiterhin die alten Klassen aufrufen (z. B. Plugins oder schwieriger zu ändernde Stellen), kann man die alten Klassen so umbauen, dass sie intern die neue FileService benutzen. Z.B. `FtpManager.listDir(path)` könnte einfach `return fileService.list(path)` aufrufen. Diese Deprecated-Klassen könnte man mittelfristig entfernen. Diese Strategie stellt sicher, dass kein alter FTPClient-Code mehr aktiv ist.

26. Unabhängig von A oder B: entferne sämtliche direkten Abhängigkeiten auf `org.apache.commons.net.ftp.FTPClient` aus dem Code. Die einzige verbleibende Nutzung sollte indirekt über VFS erfolgen. Dadurch sind Probleme mit Sitzungszuständen eliminiert.

27. **Modultests und Verifikation:** Führe umfassende Tests der neuen Implementierung durch:

28. Teste Listing, Datei-Öffnen, -Speichern auf lokalen Verzeichnissen (inkl. Edge-Cases wie Root-Verzeichnis, sehr viele Dateien, fehlende Berechtigungen).
29. Teste dasselbe auf einem FTP-Server (sofern möglich ein Test-FTP einrichten): Verbinde, liste Verzeichnisse, navigiere tiefere Pfade, lade Dateien herunter, hoch, lösche, etc. Verifiziere, dass Bookmarks funktionieren (z. B. Pfad `/dir/subdir` kann jederzeit direkt gelistet werden, unabhängig von vorherigem `cwd`).
30. Falls verfügbar, teste gegen ein z/OS-FTP-Server: Liste ein Dataset und ein PDS, schaue wie die Ausgabe der FileNodes aussieht (evtl. sind PDS-Mitglieder als Dateien mit bestimmten Namensschemata sichtbar). Überprüfe, ob die Architektur damit umgehen kann oder ob weitere Anpassungen (über Adapter) nötig sind.

31. Achte besonders auf Thread-Safety, falls parallele Operationen nun möglich sein sollen: Starte z. B. zwei Threads, die gleichzeitig verschiedene Verzeichnisse über denselben FileService listen – dies sollte funktionieren, da VFS intern synchronisiert (FileSystemManager ist threadsafe laut Doku, sofern default cache genutzt wird).

32. **MVS-Spezialfälle umsetzen (bei Bedarf):** Wenn die Anforderungen es vorsehen, implementiere nun die speziellen Adapter/Provider für MVS:

33. Erstelle ggf. eine Klasse `MvsFtpFileProvider` (implementiert evtl. VFS `FileProvider`-Interface) oder eine höhere Abstraktion, welche erkennt, ob der aktuelle `FileObject` ein Hauptframe-Dataset repräsentiert. Diese Klasse könnte im Verbund mit Commons VFS registriert werden, um z. B. ein eigenes Schema (`mvs://`) zu unterstützen, das intern auf ftp übersetzt, aber Listings anders verarbeitet.
34. Alternativ, falls kein eigenes Schema gewünscht: Implementiere in `VfsFileService` oder als separater Decorator eine Logik, die beim Auflisten (`list()`) prüft, ob das Ergebnis von einem MVS-Server kommt (vielleicht via `FileObject.getName().getScheme()` = "ftp" und Server-OS = MVS, was man indirekt an Dateinamenskonvention erkennen kann). In diesem Fall parse die `FileNode`-Namen: MVS-Listings enthalten z. B. `<MEMBER>` Einträge für PDS. Man könnte z. B. einen `FileNode` mit Name "`LIBRARY(MEMBER)`" aufsplitten in einen virtuellen Unterordner "`LIBRARY`" mit Kind "`MEMBER`". Implementiere diese Logik und teste sie gegen realistische Listing-Beispiele. Dokumentiere diese Anpassungen klar im Code (z. B. Kommentar: "// Handle z/OS PDS member listing").

35. Dieser Schritt ist optional und soll nur umgesetzt werden, wenn die Anwendung tatsächlich MVS-spezifische Navigation unterstützen muss. Ansonsten kann das Grundsystem auch so bleiben, da VFS zumindest die Einträge als Strings liefert [6] (z. B. Mitglieder als einzelne FileObjects mit Namen in Klammern).

36. **Code-Dokumentation und Beispiele aktualisieren:** Zum Abschluss alle neuen Klassen und öffentlichen Methoden mit Javadoc-Kommentaren versehen (auf Englisch, gemäß Coding-Standards). Dabei insbesondere erklären, wie die FileService zu benutzen ist, und welche Unterschiede zur alten Implementierung bestehen (für zukünftige Entwickler nachvollziehbar machen). Aktualisiere ggf. die README/Dokumentation der Anwendung, um die neue Verbindungs- und Dateihandhabung zu erläutern. Füge Codebeispiele (ähnlich der oben gezeigten) hinzu, wie man mit `FileService` Dateien listet oder transferiert.

Nach Durchführung dieser Schritte sollte die Anwendung eine klare, erweiterbare Architektur besitzen. Sie ermöglicht den Zugriff auf verschiedene Dateisysteme über eine einheitliche Schnittstelle, hält FTP-Verbindungen im Hintergrund offen (für bessere Performance) und vermeidet die bisherigen Probleme mit zustandsbehafteter Navigation. Die Codebasis ist modularer, testbarer und auf künftige Anforderungen (wie weitere Protokolle oder besondere Systemplattformen) vorbereitet. Wichtigste Änderungen – wie die neue `FileService`-Schnittstelle und die Nutzung von Apache Commons VFS – wurden durch Beispielcode und Kommentare verdeutlicht, sodass ein Entwickler die Intention und Funktionsweise leicht nachvollziehen kann. [1] [2]

---

[1] Writing file system independent applications with commons-vfs | Carlos Sanchez's Weblog
https://blog.csanchez.org/2005/09/13/writing-file-system-independent-ap/

[2] Supported File Systems – Apache Commons VFS
https://commons.apache.org/proper/commons-vfs/filesystems.html

3 java - Too many connections with apache commons-vfs2 - Stack Overflow
https://stackoverflow.com/questions/53180385/too-many-connections-with-apache-commons-vfs2

4 6 MVSFTPEntryParser (Apache Commons Net 3.11.1 API)
https://commons.apache.org/proper/commons-net/apidocs/org/apache/commons/net/ftp/parser/MVSFTPEntryParser.html

5 Using The API – Apache Commons VFS
https://commons.apache.org/proper/commons-vfs/api.html