Setup and Interaction Guide for App1, App2 and Future plans

1 Introduction

This document provides detailed instructions for setting up, running, and interacting with App1 and App2 to manage employee data and file transfers using the EmployeeService and FileTransferService. App1 acts as the server, implementing the logic for these services, while App2 serves as the client, sending requests to access them. The guide includes setup instructions for Eclipse, database creation with sample data (25 employee records), file storage configuration, a complete list of required gRPC JARs, explanations of employee operations and file conflicts (identical and different), and the gRPC request workflow (before a request, on next, on error, on complete).

2 Overview

- App1:
  - Implements and hosts two services: EmployeeService and FileTransferService.
  - EmployeeService: Manages employee records (create, read, query) in a MySQL database, with operations like adding employees or listing them by role.
  - FileTransferService: Handles file uploads, downloads, and metadata operations, including conflict resolution for uploads.
  - Contains all logic to process requests, such as querying the database or storing files, as defined in the gRPC service implementations.
- App2:
  - Sends requests to App1's services via a console-based menu.
  - Does not implement any service logic; it only constructs and sends gRPC requests to App1 and displays responses.
- Connection:
  - App2 connects to App1 using gRPC over localhost and port 8080 (default).
  - App1's services are defined in proto files (in Common_File), and App2 uses these to generate requests.
- Workflow:
  - Set up the MySQL database with sample data and configure file storage.
  - Configure and run App1 to start the server and make services available.
  - Run App2 to connect to App1 and use the menu to send requests for employee or file operations.
  - App1 processes requests and sends responses, handling file conflicts when necessary.

3 Prerequisites

- Java 21 JDK: Install and verify with java -version in a command prompt or terminal.
- MySQL Server: Installed and running on localhost:3306 with admin access to create databases.
- Eclipse IDE: For importing and running projects.

- Project Files: Download App1, App2, and Common_File folders (containing gRPC proto files, gRPC JARs, and MySQL Connector JAR).
- File Storage Directory: Use the existing storage folder provided in the App1 project.

4 Setup Instructions

4.1 Install and Configure Prerequisites

1. Install Java 21 JDK:
   - Download from the Oracle website or OpenJDK.
   - Follow the installer instructions to set it up.
   - Set JAVA_HOME environment variable and add JDK's bin folder to your PATH (search online for "set JAVA_HOME Windows" or "set JAVA_HOME Linux/Mac" if needed).
   - Verify: Open a command prompt (Windows) or terminal (Linux/Mac), type java -version, and ensure it shows Java 21.
2. Install MySQL Server:
   - Download from the MySQL website.
   - Follow the installer to set up the server, noting your username (e.g., root) and password.
   - Start the server (usually automatic after installation).
3. Install Eclipse IDE:
   - Download Eclipse IDE for Java Developers from the Eclipse website.
   - Run the installer and follow the prompts.
   - Launch Eclipse after installation.
4. Download Project Files:
   - Obtain App1, App2, and Common_File folders from the project repository or your team.
   - Save them in an easy-to-find directory, e.g., C:_project (Windows) or /home/you-username/grp_project (Linux/Mac).

4.2 Set Up the MySQL Database

1. Access MySQL:
   - Open a MySQL client like MySQL Workbench (installed with MySQL) or use the command line by typing mysql -u root -p in a command prompt or terminal.
   - Enter your MySQL username (e.g., root) and password when prompted.
2. Create Database and Table:
   - In your MySQL client, run these commands to create a database called employee_db and a table called employees:

```
CREATE DATABASE employee_db; USE
employee_db;
CREATE TABLE employees (
    id BIGINT AUTO_INCREMENT PRIMARY KEY, name
    VARCHAR(255) NOT NULL,
    role VARCHAR(100) NOT NULL, salary
    DOUBLE NOT NULL,
    district VARCHAR(100) NOT NULL,
    branch VARCHAR(100) NOT NULL
);
```

- Press Enter after each command. You should see a confirmation like ”Query OK” for each.

3. Insert Sample Data:

- Run the following SQL command to insert 25 employee records, covering 12 roles (Developer, Analyst, Manager, Designer, Tester, DevOps, Support, HR, Sales, Marketing, Finance, Security):

```
INSERT INTO employees (name, role, salary, district, branch) VALUES ('John
    Smith', 'Developer', 75000, 'New York', 'Main'),
    ('Alice Johnson', 'Analyst', 62000, 'California', 'West'),
    ('Robert Brown', 'Manager', 90000, 'Texas', 'South'),
    ('Emma Davis', 'Designer', 68000, 'Florida', 'East'), ('Michael
    Lee', 'Tester', 65000, 'Illinois', 'Central'), ('Sarah Wilson',
    'DevOps', 78000, 'Washington', 'North'), ('David Clark',
    'Support', 55000, 'Georgia', 'South'),
    ('Laura Martinez', 'HR', 60000, 'Ohio', 'Central'),
    ('James Taylor', 'Sales', 70000, 'Arizona', 'West'), ('Olivia
    Moore', 'Marketing', 67000, 'Colorado', 'North'),
    ('William Anderson', 'Finance', 72000, 'Massachusetts', 'East'),
    ('Sophia Thomas', 'Security', 74000, 'Virginia', 'Main'), ('Daniel
    White', 'Developer', 76000, 'Nevada', 'West'),
    ('Emily Harris', 'Analyst', 63000, 'Oregon', 'North'), ('Thomas
    Lewis', 'Manager', 91000, 'Pennsylvania', 'East'), ('Ava Walker',
    'Designer', 69000, 'Michigan', 'Central'), ('Joseph Hall', 'Tester',
    66000, 'North Carolina', 'South'), ('Isabella Allen', 'DevOps',
    79000, 'New Jersey', 'Main'), ('Charles Young', 'Support', 56000,
    'Tennessee', 'South'), ('Mia King', 'HR', 61000, 'Indiana',
    'Central'),
    ('Ethan Wright', 'Sales', 71000, 'Utah', 'West'),
    ('Amelia Lopez', 'Marketing', 68000, 'Maryland', 'East'),
    ('Alexander Scott', 'Finance', 73000, 'Wisconsin', 'North'),
    ('Harper Green', 'Security', 75000, 'Minnesota', 'Central'),
    ('Benjamin Adams', 'Developer', 77000, 'Missouri', 'South');
```

- In MySQL Workbench, copy-paste the command into a query window and click the lightning bolt icon to run. In the command line, paste and press Enter.
- The data includes varied names, salaries ($55,000-$91,000), districts (U.S. states), and branches (Main, West, South, East, Central, North), with 2-3 employees per role.

4. Verify:

- Run these commands to confirm the data:

```
SELECT COUNT(*) FROM employees;
SELECT role, COUNT(*) FROM employees GROUP BY role;
```

- The first command should show 25 records. The second should list each role with 2-3 employees (e.g., Developer: 3, Analyst: 2).
- If you see errors or no data, ensure the command ran correctly or check with your team.

4.3 Set Up the File Storage Directory

1. Locate the Existing Storage Folder:
   - The App1 project includes a folder named storage for file operations, located at App1/storage.
   - Find its full path, e.g.:
     - Windows: C:_project1
     - Linux/Mac: /home/yourusername/grpc_project/App1/storage
   - Open your file explorer, navigate to the App1 folder, and confirm the storage folder exists.
   - Ensure your user account has permission to read and write to this folder (usually automatic).
2. Copy the Storage Path:
   - Right-click the storage folder in your file explorer, select Properties (Windows) or Get Info (Mac), and copy the full path (e.g., C:_project1).
   - Save this path in a text editor (e.g., Notepad) for use in configuring App1 later.

4.4 Import Projects into Eclipse

1. Open Eclipse:
   - Double-click the Eclipse icon on your desktop or search for ”Eclipse” in your applications.
   - When Eclipse starts, it asks for a workspace (a folder where your projects are stored). Choose a folder like C:-workspace (Windows) or /home/yourusername/eclipse-workspace (Linux/Mac), or accept the default.
   - Click Launch or OK. Eclipse opens, showing a welcome screen or a blank workspace.
2. Import App1 Project:
   - At the top of the Eclipse window, find the menu bar and click File.
   - From the dropdown, select Open Projects from File System.
   - In the dialog box that appears, click the Directory button next to ”Import source.”
   - Navigate to the App1 folder in your project directory (e.g., C:_project1).

- Select the App1 folder and click OK or Open. The folder path should appear in the "Import source" field.
- Ensure App1 is checked in the list of detected projects (it should appear automatically).
- Click Finish. The App1 project should appear in the "Package Explorer" on the left side of Eclipse.
- If it doesn't appear, repeat the steps, ensuring you selected the correct App1 folder (not a subfolder).

3. Import App2 Project:
   - Repeat the same steps for App2:
     - Click File > Open Projects from File System.
     - Click Directory, navigate to App2 (e.g., C:_project2), and select it.
     - Click OK, ensure App2 is checked, and click Finish.
   - App2 should now appear in the "Package Explorer" alongside App1.
4. Link AppService__Common_file:
   - AppService__Common_file contains shared files needed by both projects. You'll link it to App1 and App2.
   - For App1:
     - In the "Package Explorer," right-click the App1 project.
     - Select Build Path > Configure Build Path.
     - In the dialog box, go to the Source tab.
     - Click Link Source.
     - In the new dialog, click Browse, navigate to AppService__Common_file inside common__file folder (e.g., C:_project_File_Common_file), and select it.
     - Click OK, then Finish.
     - Back in the "Build Path" dialog, click Apply and Close.
   - Repeat for App2:
     - Right-click App2 > Build Path > Configure Build Path > Source > Link Source.
     - Browse to AppService__Common_file, select it, and click OK > Finish.
     - Click Apply and Close.
   - This links the gRPC proto files needed for both projects to communicate.
   - Verify:
     - In the "Package Explorer," expand App1 and App2. You should see AppService_Common_file listed under each project.
     - If you see a red exclamation mark on either project, check the "Problems" view (Window > Show View > Problems) and ensure you selected the correct Common_File folder.

- If confused, ask your team for the exact path to Common_File.

4.5 Add Dependencies to Projects

1. Add JARs to App1:
   - In the ”Package Explorer,” right-click App1 > Build Path > Configure Build Path.
   - In the dialog, go to the Libraries tab > Modulepath > Add External JARs.
   - Navigate to jar inside Common__file (e.g., C:_project_File_jar).
   - Select the following 30 gRPC JARs and the MySQL Connector JAR (hold Ctrl or Cmd to select multiple):
     - failureaccess-1.0.2
     - grpc-api-1.70.0
     - grpc-census-1.70.0
     - grpc-context-1.70.0
     - grpc-core-1.70.0
     - grpc-inprocess-1.70.0
     - grpc-netty-1.70.0
     - grpc-netty-shaded-1.70.0
     - grpc-protobuf-1.70.0
     - grpc-protobuf-lite-1.70.0
     - grpc-services-1.70.0
     - grpc-stub-1.70.0
     - grpc-testing-1.70.0
     - grpc-util-1.70.0
     - guava-33.3.1-jre
     - opencensus-api-0.31.1
     - opencensus-contrib-grpc-metrics-0.31.1
     - perfmark-api-0.27.0
     - protobuf-java-3.25.6
     - netty-buffer-4.1.119.Final
     - netty-codec-4.1.119.Final
     - netty-codec-http2-4.1.119.Final
     - netty-codec-http-4.1.119.Final
     - netty-codec-socks-4.1.119.Final
     - netty-common-4.1.119.Final
     - netty-handler-4.1.119.Final
     - netty-handler-proxy-4.1.119.Final

- mysql-connector-java-8.0.x.jar
- netty-resolver-4.1.119.Final
- netty-transport-4.1.119.Final
- netty-transport-native-unix-common-4.1.119.Fina

- Also select javax.annotation-api-1.3.2.jar from the same folder.
- Click Open, then Apply.
- In the ”Libraries” tab, click Order and Export, check all added JARs, and click Apply and Close.

2. Add JARs to App2:
   - Repeat for App2: Right-click App2 > Build Path > Configure Build Path > Libraries > Modulepath > Add External JARs.
   - Select the same 30 gRPC JARs and javax.annotation-api-1.3.2.jar from Common_File_jar, but do not select mysql-connector-java-8.0.x.jar.
   - Click Open, Apply, check all JARs in Order and Export, and click Apply and Close.
3. Verify:
   - Check the ”Problems” view (Window > Show View > Problems) for errors.
   - If errors appear, ensure all JARs were added correctly and Common_File is linked. Ask your team if issues persist.

4.6 Configure App1

1. Open Main.java:
   - In the ”Package Explorer,” expand App1 > src > Main, and double-click Main.java to open it in the editor.
2. Update Database Credentials:
   - Find these lines in Main.java:

```
String dbUrl = "jdbc:mysql://localhost:3306/employee_db";
String dbUsername = "your_mysql_username"; // e.g., "root"
String dbPassword = "your_mysql_password"; // e.g., "your_secure_password";
```

   - Replace your_mysql_username with your MySQL username (e.g., root).
   - Replace your_mysql_password with your MySQL password.
3. Update File Storage Path:
   - Find this line:

```
String fileStoragePath = "path_to_storage";
```

   - Replace path_to_storage with the storage folder path you copied earlier, e.g.:

```
String fileStoragePath = "C:\\Users\\YourUsername\\grpc_project\\App1\\storage"; //
    Windows
// or
String fileStoragePath = "/home/yourusername/grpc_project/App1/storage"; // Linux/
    Mac
```

- Use double backslashes (
) for Windows paths, single forward slashes (/) for Linux/Mac.
- Replace YourUsername and grpc_project with your actual username and project directory.

4. Save:
   - Click File > Save or press Ctrl+S (Cmd+S on Mac) to save Main.java.

4.7 Configure App2

1. Open Main.java:
   - In the ”Package Explorer,” expand App2 > src > Main, and double-click Main.java.
2. Verify Server Connection:
   - Find these lines:

```
String host = "localhost";
int port = 8080;
```

   - Ensure they match App1’s settings (localhost and 8080 are defaults). Change only if your team specifies a different host or port.
3. Save:
   - Click File > Save or press Ctrl+S (Cmd+S on Mac).

5 Starting App1

1. Run App1:
   - In the ”Package Explorer,” right-click Main.java in App1’s Main package.
   - Select Run As > Java Application.
   - The server starts on port 8080, connects to MySQL, and initializes services.
   - In the ”Console” view (bottom of Eclipse), you should see:

```
Server started on port 8080
Storage directory initialized on App1: C:\Users\YourUsername\grpc_project\App1\
    storage
```

2. Verify:
   - Check the ”Console” for errors about the database or storage path.
   - If no errors, App1 is ready to handle requests from App2.

6 Starting App2

1. Run App2:
   - Right-click Main.java in App2's Main package > Run As > Java Application.
   - App2 connects to App1 and displays a menu in the "Console":

```
COMBINED gRPC CLIENT (App2)
Connected to server at localhost:8080
Available options:
1. Employee Operations
2. File Transfer Operations
3. Exit
Enter option (1-3):
```

2. Verify:
   - Ensure no connection errors in the "Console."
   - The menu should respond when you type 1, 2, or 3 and press Enter.

7 How App1 and App2 Connect

App1 and App2 communicate using gRPC, a system for sending requests and responses between a server and a client. App1 is the server, implementing the EmployeeService and FileTransferService, which contain the logic for all operations (e.g., querying the database, storing files). App2 is the client, sending requests to these services without implementing any logic itself.

- gRPC Connection:
  - App1 runs on localhost:8080, hosting the services defined in proto files (in Common_File).
  - App2 creates a gRPC channel to connect to App1:

```
channel = ManagedChannelBuilder.forAddress(host, port).usePlaintext().build();
```

  - Uses plaintext for simplicity (no encryption, suitable for local testing).
- Stubs:
  - App2 uses stubs (generated from proto files) to send requests to App1's services.
  - Blocking stubs are used for simple requests (e.g., get employee by ID), waiting for a single response.
  - Async stubs are used for streaming requests (e.g., file uploads), handling multiple messages.
- Service Implementation:
  - App1 contains the code for EmployeeService and FileTransferService, written in Java. For example:
    * EmployeeService: Queries the MySQL database (e.g., SELECT * FROM em- ployees WHERE id = ?).
    * FileTransferService: Saves files to the storage directory and checks for conflicts.

  - App2 only sends requests to App1, which processes them and sends back responses.
- Request-Response:
  - App2 sends a request (e.g., ”get employee ID 1” or ”upload file report.pdf”).
  - App1 processes the request (e.g., queries database, saves file).
  - App1 sends a response (e.g., employee details, upload status).
  - App2 displays the response.

7.1 Understanding gRPC Request Workflow

To make a gRPC request from App2 to App1, several steps occur: before a request, on next (during processing), on error (if something goes wrong), and on complete (when finished). Below, we explain each using the example of requesting an employee by ID.

7.1.1 Before a Request

- What Happens:
  - App1 Must Be Running: App1’s server must be started on localhost:8080 with services available.
  - App2 Connects: App2 creates a gRPC channel:

```
channel = ManagedChannelBuilder.forAddress("localhost", 8080).usePlaintext().build();
```

  - Create a Stub: App2 creates a stub for EmployeeService:

```
EmployeeServiceBlockingStub stub = EmployeeServiceGrpc.newBlockingStub(channel);
```

  - User Input: App2 prompts:

```
Enter employee ID: 1
```

    App2 builds a request:

```
EmployeeRequest request = EmployeeRequest.newBuilder().setId(1).build();
```

  - Resources Ready: App1 ensures database access:

```
Connection conn = DriverManager.getConnection(dbUrl, dbUsername, dbPassword);
```

- Why It Matters: Ensures App1 is ready and App2 sends valid requests.
- Example: Enter ID 1, App2 builds EmployeeRequest(id=1).

7.1.2 On Next

- What Happens:
  - App2 Sends the Request:

```
EmployeeResponse response = stub.getEmployeeById(request);
```

- App1 Processes: Queries the database:

```
SELECT * FROM employees WHERE id = 1;
```

- App1 Sends Response:

```
EmployeeResponse.newBuilder()
    .setId(1)
    .setName("John Smith")
    .setRole("Developer")
    .setSalary(75000.0)
    .setDistrict("New York")
    .setBranch("Main")
    .build();
```

- App2 Displays:

```
Employee details from App1:
ID: 1
Name: John Smith Role:
Developer Salary:
75000.00 District: New
York Branch: Main
```

- Why It Matters: Ensures correct processing and user feedback.
- Example: App1 returns employee ID 1's details; App2 shows them.

7.1.3 On Error

- What Happens:
  - Error Occurs: E.g., ID 999 doesn't exist.
  - App1 Sends Error:

```
Status.NOT_FOUND.withDescription("Employee not found for ID: 999")
```

  - App2 Displays:

```
Error from App1: Employee not found for ID: 999
```

  - Recovery: Returns to menu.
- Why It Matters: Helps users correct issues.
- Example: Enter ID 999, see error, try again.

7.1.4 On Complete

- What Happens:
  - Request Finishes: App1 sends final response.
  - App2 Finalizes: Shows result, returns to menu.
  - Cleanup: Closes streams if needed:

```
requestObserver.onCompleted();
```

- Why It Matters: Confirms success and frees resources.
- Example: Shows employee details, awaits next input.

7.1.5 Why This Workflow Matters

Ensures reliable communication. App1 handles logic; App2 focuses on interaction.

7.1.6 Tips for gRPC Requests

- Start App1 first.
- Use valid inputs (e.g., numeric IDs like 1).
- Read error messages.
- Test file uploads for streaming.

8 App2 Methods: Employee Operations

Select option 1 for Employee Operations.

8.1 Understanding Employee Operations

Employee operations manage employee data in a MySQL database on App1. App2 sends requests to App1 to add, find, or list employees.

8.1.1 What Are Employee Operations?

13 actions, e.g., adding an employee or listing by role. Data is stored in the employees table with columns: id, name, role, salary, district, branch. Sample data includes 25 employees across 12 roles.

8.1.2 How Employee Operations Work

1. Enter Details: E.g., ID 1.
2. App2 Sends Request: Via gRPC to App1.
3. App1 Queries Database:

```
SELECT * FROM employees WHERE id = 1;
```

4. App1 Sends Results: Employee ID 1's details.

5. App2 Shows Results:

```
Employee details from App1:
ID: 1
Name: John Smith
Role: Developer
Salary: 75000.00
District: New York
Branch: Main
```

8.1.3 Why Employee Operations Matter

Efficient and flexible for managing staff data.

8.1.4 Tips for Using Employee Operations

- Test with sample IDs (e.g., 1 to 25).
- Use exact role names (e.g., ”Developer”).
- Explore all 13 options.

8.2 Employee Operation Methods

8.2.1 Create Employee (Option 1)

- What it does: Adds a new employee.
- Input:

```
Enter name: Jane Doe
Enter role: Developer
Enter salary: 80000
Enter district: California
Enter branch: West
```

- Output:

```
Created successfully on App1:
ID: 26
Name: Jane Doe
Role: Developer
Salary: 80000.00
District: California
Branch: West
```

8.2.2 Get Employee by ID (Option 2)

- What it does: Finds an employee by ID.
- Input:

```
Enter employee ID: 1
```

- Output:

```
Employee details from App1:
ID: 1
Name: John Smith
Role: Developer
Salary: 75000.00
District: New York
Branch: Main
```

- If not found: "Error from App1: Employee not found for ID: [ID]."

8.2.3 Other Methods

- Get Employees by Role (Option 3): Lists employees by role.
- Get Employees by Salary (Option 4): Lists by exact salary.
- Get Employees by District (Option 5): Lists by district.
- Get Employees by Branch (Option 6): Lists by branch.
- Get Employees (Option 7): Lists all employees.
- Get Employees by Role and Branch (Option 8): Lists by role and branch.
- Get Employees with Salary > (Option 9): Lists above a salary.
- Get Employees with Salary Between (Option 10): Lists in a salary range.
- Get Employees by Role and Min Salary (Option 11): Lists by role and minimum salary.
- Get Employees by District Ordered by Salary (Option 12): Lists by district, sorted by salary.
- Get Employee Names and Roles by Branch (Option 13): Lists names and roles by branch.

9 App2 Methods: File Transfer Operations

Select option 2 for File Transfer Operations.

9.1 Understanding File Conflicts During Uploads

Conflicts occur when uploading a file with the same name as an existing file in App1's storage directory. App1 uses SHA-256 hashes to detect identical (same content) or different (different content) conflicts.

9.1.1 What Are File Conflicts?

- Identical Conflict: Same name and content (e.g., uploading report.pdf when an identical report.pdf exists).
- Different Conflict: Same name, different content (e.g., uploading a new report.pdf with different text).

9.1.2 How App1 Detects Conflicts

App1 checks for conflicts after an upload by comparing the SHA-256 hash of the new file with the hash of the existing file (if any) in the storage directory (App1/storage). SHA-256 is a secure method that creates a unique ”fingerprint” (a 32-character code) for each file based on its content. If the hashes match, the files are identical; if they differ, the files have different content, even if the names are the same.

- Process:

```
MessageDigest digest = MessageDigest.getInstance("SHA-256");
byte[] hash = digest.digest(fileBytes);
```

- Why SHA-256?: It ensures accurate detection of file differences, even for large files, without comparing every byte.

9.1.3 Types of Conflicts and How They’re Handled

- Identical Conflict:
  - App2 shows:

```
Identical file detected on App1: report.pdf
Options:
  Leave existing file
  Upload with a new filename
Enter choice (1-2):
```

  - Choice 1: No action; the existing file remains.
  - Choice 2: User enters a new filename (e.g., report_v2.pdf), and App1 saves the file with that name.
- Different Conflict:
  - App2 shows:

```
Conflict detected on App1: Different file with same name exists: report.pdf
Options:
  Override existing file
  Upload with a new filename
  Cancel upload
Enter choice (1-3):
```

  - Choice 1: App1 replaces the existing file with the new one.
  - Choice 2: User enters a new filename, and App1 saves the file.
  - Choice 3: App1 discards the uploaded file.

9.1.4 Why This Matters

Conflict handling prevents accidental overwrites or duplicates, ensuring users control how files are managed.

9.1.5 Tips to Avoid Conflicts

- Use unique filenames (e.g., add dates or versions like report_2025.pdf).
- Use List Files to check existing files before uploading.

9.2 File Transfer Methods

The following methods allow App2 to manage files on App1's storage directory (App1/storage) using the FileTransferService. Each method uses gRPC to communicate with App1, and large files are handled by breaking them into small chunks for efficient transfer.

9.2.1 List Files (Command: list)

- What it does:
  - Shows a list of all files stored in App1's storage directory.
  - Helps users check available files before uploading or downloading.
- How it works:
  - User Input: In App2's console, type:

```
Enter command: list
```

  - App2 Sends Request: App2 sends a gRPC request to App1's FileTransferService using a blocking stub:

```
FileListResponse response = stub.listFiles(FileListRequest.newBuilder().build());
```

  - App1 Processes: App1 scans the storage directory (App1/storage) and collects file-names:

```
File folder = new File(storagePath);
String[] files = folder.list();
```

  - App1 Sends Response: App1 returns a list of filenames.
  - App2 Displays:

```
Files on App1:
- report.pdf
- image.jpg
```

  - Note: This uses a simple gRPC call (not streaming), as the list is typically small.
- Why it matters: Lets users see what files are available without downloading them.

9.2.2 Upload File (Command: upload)

- What it does:
  - Sends a file from the user's computer to App1's storage directory.
  - Handles large files by breaking them into small chunks and checks for conflicts.
- How it works:
  - User Input: In App2's console, type:

```
Enter command: upload
Enter path of file to upload: C:\Users\YourUsername\Documents\report.pdf
```

  - App2 Reads File: App2 opens the file and divides it into small chunks (e.g., 64 KB), like breaking a book into pages:

```
File file = new File(filePath);
ByteBuffer buffer ;
Buffer = ByteBuffer.allocateDirect(1024 * 1024); 1mb
```

  - App2 Sends Chunks: App2 uses a gRPC async stub to stream chunks to App1:

```
if (requestObserverHolder[0] != null) {
   requestObserverHolder[0].onNext(FileChunk.newBuilder()
  .setFileName(fileName)
  .setContent(ByteString.copyFrom(buffer))
  .setSessionId(sessionId)
  .build());
   }

buffer.clear();
```

    App2 shows progress every 10 MB:

```
Sent 1.25/2.50 MB (50.0%)
```

  - App1 Receives Chunks: App1 writes chunks to a temporary file:

```
OutputStream output = new FileOutputStream(tempFile);
output.write(request.getData().getValue().toByteArray());
```

  - App1 Checks Conflicts: After upload, App1 calculates the SHA-256 hash of the new file and compares it with the existing file (if any) in App1/storage:

```
MessageDigest digest = MessageDigest.getInstance("SHA-256");
byte[] hash = digest.digest(fileBytes);
```

  - Handle Conflicts:
    - No Conflict: App1 saves the file as report.pdf.
    - Identical Conflict: App2 prompts:

```
Identical file detected on App1: report.pdf
Options:
  Leave existing file
  Upload with a new filename
  Enter choice (1-2):
```

- Choice 1: No action.
- Choice 2: User enters report_v2.pdf; App1 saves it.

* Different Conflict: App2 prompts:

```
Conflict detected on App1: Different file with same name exists: report.pdf
Options:
  Override existing file
  Upload with a new filename
  Cancel upload
Enter choice (1-3):
```

- Choice 1: App1 overwrites the existing file.
- Choice 2: User enters report_v2.pdf; App1 saves it.
- Choice 3: App1 discards the file.

– App1 Finalizes: Saves or discards the file based on user choice.

– App2 Displays:

```
Upload completed successfully for: report.pdf
```

- Why it matters: Enables efficient file uploads with conflict resolution.

9.2.3 Download File (Command: download)

- What it does:
  – Retrieves a file from App1's storage directory to the user's computer.
  – Handles large files by streaming in 64 KB chunks.
- How it works:
  – User Input: In App2's console, type:

```
Enter command: download
Available files:
  report.pdf
  image.jpg
Enter file number (0 to cancel): 1
Enter directory path: C:\Users\YourUsername\Downloads
Enter filename (press Enter for original):
```

  – App2 Sends Request: App2 sends a gRPC request to App1's FileTransferService:

- App1 Reads File: App1 opens the file and splits it into 64 KB chunks:
- App1 Sends Chunks: App1 streams chunks to App2 via gRPC:
- App2 Receives Chunks: App2 writes chunks to a file:

  App2 shows progress every 10 MB:

```
Received 1.25/2.50 MB (50.0%)
```

- App2 Finalizes: App2 confirms completion:

```
Download completed successfully for: report.pdf
```

- Why it matters: Allows efficient retrieval of files from App1.

10 Future Enhancements

To improve the scalability and reliability of App1 and App2, several enhancements are planned to transition the system toward a production-ready architecture.

- Current Limitation: App2 connects directly to App1's server (localhost:8080). If the server fails, App2 cannot send requests, leading to service disruptions.
- Next Feature: Load Balancer Integration: Introduce a load balancer to distribute requests and responses between App2 (client) and multiple App1 instances (servers). This ensures that if one server fails, the load balancer redirects requests to another available server, maintaining service continuity.
- Scaling to Production: Expand the system to emulate a real-world production environment, supporting 5 servers (App1 instances) and 5 clients (App2 instances). All in- stances will communicate through an intermediate load balancer to manage traffic effi-ciently and ensure high availability.
- Long-Term Goal: Apply this architecture to a real-world project. The project will be divided into manageable components and developed using plain gRPC and Java, leveraging the load-balanced infrastructure to handle large-scale, distributed workloads.