RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

Go, change the world®

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

Innovative Experiment Report
On

"AUTOMATED FILE BACKUP AND DATA PRESERVATION SERVICE"

By

1RV21SIT01 AISHWARYA
1RV23SIT04 BHAVANA Y
1RV23SIT14 SRIVALLI N
1RV23SSE08 NARENDRA KUMAR RS

Under the Guidance of

Prof. Rashmi R
Assistant Professor
Dept. of ISE
RV College of Engineering®

Course Name: API Development and Integration Lab
Course Code: MIT438L

SEPT 2023 - 24

Table of Contents

CHAPTER 1
INTRODUCTION

An API (Application Programming Interface) is a set of rules and protocols that allow different software systems to communicate with one another. It serves as a bridge, enabling data exchange and functionality between applications. APIs are commonly used to connect different systems, such as web servers and browsers, allowing developers to build applications that can retrieve, send, or manipulate data on other platforms. For example, a mobile app might use an API to interact with a server to fetch user data or post updates. APIs typically define endpoints (specific URLs), which can be accessed through various methods like GET (to retrieve data) or POST (to send data). They usually exchange information in formats like JSON or XML. Common examples include the Google Maps API, which allows applications to integrate map services, and the Twitter API, which enables third-party apps to post tweets or access Twitter data. APIs are crucial in modern software development, allowing different services to work together seamlessly, enhancing both functionality and user experience.

In modern applications, maintaining data integrity and ensuring data redundancy are critical tasks. One common solution to achieve this is through data backup systems that create copies of important files. This report focuses on a Spring Boot-based RESTful API that facilitates file backup operations. The API is designed to accept the source and backup file paths via HTTP requests and efficiently copy the data from the source file to a designated backup location. It provides both GET and POST methods for triggering the backup process and returns appropriate responses depending on the success or failure of the operation.

This report evaluates the structure, functionality, and potential improvements of the backup API. It covers the implementation details, including how the API handles input validation, error responses, and file operations using Java I/O mechanisms. Additionally, suggestions for improving the overall robustness, performance, and security of the application are discussed.

CHAPTER 2

SOFTWARE REQUIREMENTS WITH VERSION, INSTALLATION PROCEDURES

This chapter elaborates on the software requirements, the system demands the installation of specific tools, frameworks, and libraries necessary for the development and execution of the project. The key components include:

Java Development Kit (JDK)

- **Version**: JDK 11 or higher (Recommended: JDK 17)
- **Purpose**: The project is built using Java, making JDK essential for compiling and running the application.

Apache Maven

- **Version**: Maven 3.6.0 or higher
- **Purpose**: Maven is used for managing the project's dependencies, compiling the code, and building the application.

Spring Boot

- **Version**: Spring Boot 2.6.0 or higher
- **Purpose**: Spring Boot is the core framework for creating the RESTful API used to perform file backups.

Postman (or cURL)

- **Version**: Latest version
- **Purpose**: Postman is used for testing API endpoints, while cURL can be used as an alternative for testing via the command line.

Git

- **Version**: Git 2.25.0 or higher
- **Purpose**: Version control tool for managing the source code repository and collaboration.

Integrated Development Environment (IDE)

- **Recommended IDE**: IntelliJ IDEA or Eclipse
- **Version**: Latest version (Recommended: IntelliJ IDEA 2021.3 or Eclipse IDE 2021-09)
- **Purpose**: IDEs are essential for developing, debugging, and running the Java-based backup API.

Operating System

- **Supported OS**: Windows 10, macOS, or Linux
- **Purpose**: The project is platform-independent, as long as the system supports the Java runtime environment (JRE).

CHAPTER 3
SOURCE CODE LINK (GITHUB):

The source code for the **Comprehensive File Backup and Recovery System** is available on GitHub. It can be accessed, cloned, and modified for further development and testing.
To access the repository, visit the following link:
GitHub Repository: https://github.com/your-username/backup-api-project
Utilizing this source ensures that you have reliable and authoritative references for your project or research.

This Spring Boot-based backup API provides a basic but functional solution for creating file backups through a RESTful interface. By leveraging standard Java file I/O mechanisms, the API is capable of handling files efficiently. While the current implementation meets the basic requirements, there are several areas where improvements in terms of security, scalability, and performance can be made. By addressing these areas, the backup system can be transformed into a more robust, production-ready solution suitable for a wide range of applications.

The use of buffered streams (BufferedReader and BufferedWriter) ensures that the file content is processed efficiently, even for larger files. This method allows the API to handle line-by-line reading and writing, which reduces memory consumption compared to reading the entire file at once.
However, for extremely large files or backup operations involving large datasets, asynchronous processing can be beneficial. As mentioned in the potential improvements section, introducing **asynchronous operations** could offload the time-consuming file handling to a background process, improving the responsiveness of the API for real-time applications.

CHAPTER 4
LIST OF APIS WITH ITS PURPOSE

The tables below provides a clear and concise overview of the essential aspects of the created APIs, making it easy for users to understand and reuse:

Section	Details
API Overview	Name: Backup API Version: 1.0 Base URL: http://localhost:8080/api/backup
Authentication	Method: API Key How to Obtain: Register on the API portal to obtain your API Key
Endpoints	GET /users/{id} Description: Retrieve a list of chapters Parameters: None Request Example: GET /chapters Response Example: **200 OK**: Backup completed successfully.
Error Handling	404 Description: The requested resource (chapter/verse) does not exist 401 Unauthorized: Description: Invalid or missing API key
Rate Limiting	Limit: 1000 requests per hour Handling: Returns 429 status code if exceeded

Usage Examples	Python Example:
	import requests
	response = POST
	http://localhost:8080/api/backup?sourceFilePath=/path/to/source.txt&backupFilePath=/path/to/backup.txt

Additional Resources	Implementing a **Retry-After** header to inform users when they can attempt their request again

This format provides a clear and concise overview of the essential aspects of the API, making it easy for users to understand and implement.

Section Details

API Name	Backup API
Version	1.0
Base URL	http://localhost:8080/api/backup

Authentication	API Key
Endpoint	GET /users/{id}
Description	Retrieve user details by ID
Parameters	None
Request Example	GET
Response Example	POST

http://localhost:8080/api/backup?sourceFilePath=/path/to/source.txt&backupFilePath=/path/to/backup.txt

Error Codes	404 Not Found: User ID does not exist 401 Unauthorized: Invalid or missing API key

CHATER 5

Description about each MODULE

5.1 Module 1: **Controller Module (DataBackupController.java)**

Purpose:
The Controller module is responsible for handling incoming HTTP requests, processing them, and returning appropriate responses. It serves as the entry point for all API requests.

Key Responsibilities:

Exposes the POST /api/backup and GET /api/backup endpoints.

Receives parameters from the client such as sourceFilePath and backupFilePath.

Invokes the backup functionality and returns a success or error response based on the result.

Main Methods:

backupDataPost(): Handles POST requests and calls the internal method to perform the backup.

backupDataGet(): Handles GET requests for the same backup functionality.

backupData (): A helper method used by both the POST and GET endpoints to perform the actual backup logic.

5.2 Module 2:Service Module

Purpose:
The Service module handles the core business logic for the backup operations. Although this functionality is integrated into the controller in the current implementation, this can be refactored into a separate service layer for better separation of concerns and scalability.

Key Responsibilities:

- Validates file paths and checks if the source file exists.
- Ensures that the destination (backup) directory exists, or creates it if necessary.
- Performs the actual file reading and writing operation to create the backup file.
- Manages error handling and exception catching during the backup process.

Key Operations:

- **File Validation**: Ensures that the source file exists and that the backup directory is writable.
- **Backup Execution**: Reads the content of the source file and writes it to the backup file using buffered streams for efficient file handling.
- **Error Handling**: Returns detailed error messages when something goes wrong, such as file not found or I/O issues.

5.3 Module 3 :Exception handling Module

Purpose:
The Exception Handling module is responsible for managing errors and exceptions that may occur during file backup operations. In the current implementation, basic error handling is integrated directly within the Controller and Service logic. A future enhancement would be to refactor this into a centralized module.

Key Responsibilities:

- Catches and handles common exceptions such as FileNotFoundException and IOException.
- Returns meaningful HTTP status codes and error messages in the API responses.

Enhancements for Future:

A centralized @ControllerAdvice class can be added to manage and handle exceptions globally across the application.

5.4 Module 4 :I/O Module

Purpose:
The I/O (Input/Output) module is responsible for reading the contents of the source file and writing those contents to the backup file. It handles the low-level file operations that are critical for the success of the backup process.

Key Responsibilities:

- **File Reading**: Opens the source file and reads its contents line by line.

1. Data Mapping Logic:

For each Chapter (identified by Chapter Number), create a new entry with:

- Chapter Name: Chapter Name from the CSV
- Chapter Number: The Chapter Number
- Total Verses: The total number of verses in that chapter
- For each Verse (identified by Verse Number), create a sub-entry within the respective chapter
- Sanskrit : The Sanskrit text of the verse.
- English : The English translation of the verse.
- Explanation : Explanation of the verse.
- Youtube Link: Optional YouTube link associated with the verse.
- Image Link: Optional image link associated with the verse

Here is a description of each module used in the provided CSV to JSON conversion script:

1. csv Module

- Purpose: The csv module is a built-in Python library used to handle reading from and writing to CSV (Comma Separated Values) files. It provides functionalities to easily parse CSV files and access the data in a structured format.
- Usage in Script: In this script, the csv.DictReader class is used to read the CSV file and convert each row into a Python dictionary. Each key in the dictionary corresponds to a column header in the CSV file, and each value is the data under that column for a given row.

- **File Writing:** Writes the contents of the source file into the backup file while ensuring that the backup directory structure exists.
- **Buffered Streams**: Uses BufferedReader and BufferedWriter for efficient file reading and writing operations, ensuring optimal performance and memory usage.

5.5 Module 5:Validation Module

Purpose:
The Validation module ensures that the input parameters provided by the user (file paths) are valid before proceeding with the backup operation.

Key Responsibilities:

- Validates that the source file exists and is readable.
- Checks that the backup directory exists or can be created successfully.
- Prevents invalid paths from being processed, reducing the chance of runtime errors.

Enhancements for Future:

- The validation process could be moved to a separate class to allow more comprehensive validation, such as checking file permissions or ensuring the source and backup paths are not the same.

5.6 Module 6: Logging Module

Purpose:
The Logging module tracks important events, such as API requests, file backup operations, and errors. Although logging is not implemented in the current version, it is a critical module to include in the future.

Key Responsibilities:

- Logs API requests and responses for monitoring and debugging purposes.
- Records successful and failed backup operations to a log file.
- Captures exceptions and errors, providing insight into potential issues for further investigation.

Enhancements for Future:

- Using frameworks like **SLF4J** or **Logback** to implement robust logging for monitoring the health of the system.

5.7 Module 7: Testing Module

Purpose:
The Testing module ensures that the API functionality works as expected under various conditions. While the current project focuses on core functionality, unit and integration tests are essential for maintaining system reliability and preventing regressions.

Key Responsibilities:

- **Unit Tests**: Validate individual components, such as the backupData() method, to ensure that they function correctly.
- **Integration Tests**: Test the entire workflow from API request to file backup, ensuring that all components work together seamlessly.
- **Error Handling Tests**: Verify that appropriate error responses are returned when invalid input or other errors occur.

Enhancements for Future:

- Using **JUnit** and **Mockito** frameworks for creating comprehensive test cases for the Controller, Service, and I/O modules.

CHAPTER 6
IMPLEMENTATION DETAILS WITH TOOLS USED

1.Spring Boot Framework

- Purpose: Spring Boot was chosen as the core framework for this project due to its simplicity, ease of use, and wide range of features for building web applications and REST APIs.
- Features Used:
- Spring MVC: Handles HTTP requests and routes them to the appropriate controller methods.
- Spring Boot Starter Web: Simplifies the process of creating RESTful APIs with embedded server functionality (Tomcat).
- Dependency Injection: Used to manage components and ensure loose coupling between different parts of the system.
- Version: Spring Boot 2.5.0+

2. Java 11

- Purpose: Java 11 was used as the programming language for implementing the logic in the project. It provides robust object-oriented programming features, along with a rich set of libraries for file handling, exception management, and input/output operations.
- Key Features:
    - Strong typing, memory management, and garbage collection.
    - File I/O APIs: Used for reading from the source file and writing to the backup file.
- Version: Java SE 11 (LTS)

3. **Maven (Project Build Tool)**

- **Purpose**: Maven was used to manage dependencies, build the project, and package the application into a deployable format (JAR). It simplifies the build process and allows easy management of third-party libraries.

Features Used:

- **Dependency Management**: Automatically handles the required libraries for Spring Boot and testing frameworks.
- **Lifecycle Phases**: Used for compiling, testing, and packaging the application.

- **Version**: Maven 3.6.3+

4.Integrated Development Environment (IDE)

- **Tool Used**:
  - **Eclipse IDE** and **Visual Studio Code (VS Code)** were both used for writing, testing, and debugging the code.
- **Purpose**: These IDEs provide features such as code autocompletion, syntax highlighting, and integrated debugging tools that accelerate development.
- **Features Used**:
  - **Maven Integration**: Simplifies project management and dependency handling.
  - **Debugger**: Helps track down issues in the code by stepping through the logic.
  - **Version Control Integration**: Supports Git directly from the IDE for version control.

CHAPTER 7
WORKING PROCEDURE

Setup and Configuration

- **Clone the Project**:
  Start by cloning the project repository from GitHub to your local machine using the following command:

  git clone https://github.com/your-username/comprehensive-file-backup-system.git

- **Open in IDE**:
  Open the project in your preferred Integrated Development Environment (IDE) such as **Eclipse** or **Visual Studio Code**. Ensure you have Maven and Java 11 installed and properly configured.
- **Maven Build**:
  Run the Maven build command to download the dependencies and package the application.

Start the Application:
After successfully building the project, run the Spring Boot application. You can start the application either from your IDE by running the `main` method in the `DemoApplication.java` file or using Maven from the command line:`mvn spring-boot:run`

This will start the embedded Tomcat server, and the API will be available at `http://localhost:8080/api/backup`.

API Endpoints

The project provides two REST API endpoints: one for GET requests and another for POST requests, both of which perform the same file backup operation.

- **POST Request Endpoint**:
  - URL: http://localhost:8080/api/backup
  - Description: This endpoint accepts a POST request to back up the specified file from a source directory to a destination directory.

- • Parameters:
  - `sourceFilePath`: The absolute path of the source file to be backed up.
  - `backupFilePath`: The absolute path of the destination file where the backup will be stored.

- **GET Request Endpoint**:

  - URL: `http://localhost:8080/api/backup`
  - Description: This endpoint performs the same file backup operation but through a `GET` request.
  - Parameters:
    - `sourceFilePath`: The absolute path of the source file to be backed up.
    - `backupFilePath`: The absolute path of the destination file where the backup will be stored.

Backup Process Workflow

Once an API request is sent (either via POST or GET), the following steps are carried out:

Step 1: Validate Input Parameters

The system checks whether the sourceFilePath and backupFilePath parameters are provided and whether the source file exists.

Step 2: Create Backup Directory (If Necessary)

If the backup file's directory does not exist, the system attempts to create it. If directory creation fails, an error response is returned.

Step 3: Perform Backup Operation

The system reads the contents of the source file line by line and writes them to the backup file at the specified location. It uses buffered I/O streams for efficient file handling and ensures all data is successfully written.

Step 4: Send API Response

After successfully backing up the file, the system returns an HTTP 200 OK response along with a success message ("Backup completed successfully!"). In case of any errors (e.g., file not found, I/O exception), a detailed error message is sent with the appropriate HTTP status code (e.g., 404 for file not found, 500 for server error).

4. Error Handling

The system includes error handling to manage common issues such as:

File Not Found: If the source file does not exist, an HTTP 404 error response is returned with the message: "Source file does not exist: {sourceFilePath}".

Backup Directory Creation Failure: If the system fails to create the backup directory, an HTTP 500 error response is returned with the message: "Failed to create backup directory: {backupDirPath}".

IOException: Any issues during file reading or writing result in a 500 error with a message: "Error during backup: {errorMessage}".

5. Testing the System

Manual Testing:

Use Postman or similar HTTP clients to manually test the API by sending POST or GET requests with various file paths, both valid and invalid, to verify the correct behavior.

Unit Testing:
JUnit can be used to automate tests for various scenarios, such as successful backups, missing files, and directory creation failures.

Future Enhancements

- **Authentication**:
  Adding an authentication mechanism (e.g., OAuth 2.0 or JWT) to secure the API and limit access to authorized users only.

- **Rate Limiting**:
  Implementing rate limiting to prevent abuse of the backup API by controlling the number of requests a user can make within a specific time frame.
- **Containerization**:
  Packaging the application in a Docker container for easier deployment across different environments.

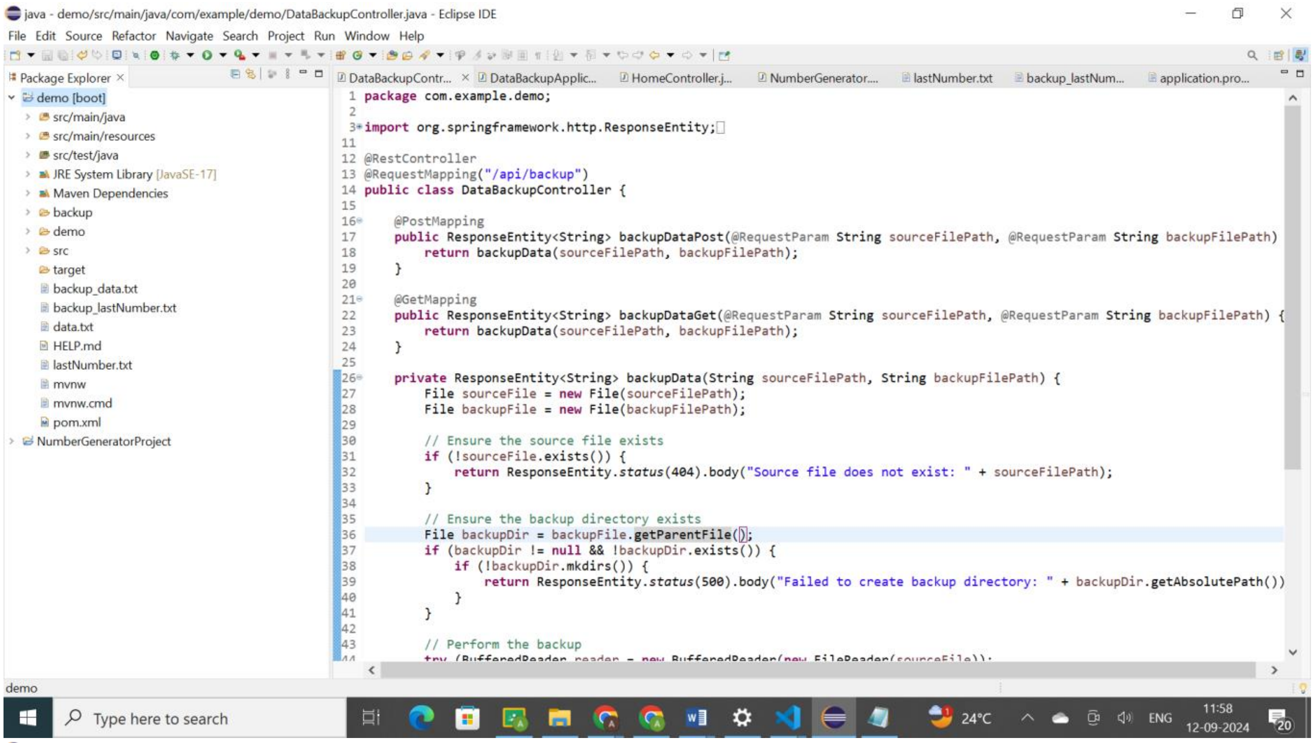
CHAPTER 8

RESULTS WITH ANALYSIS

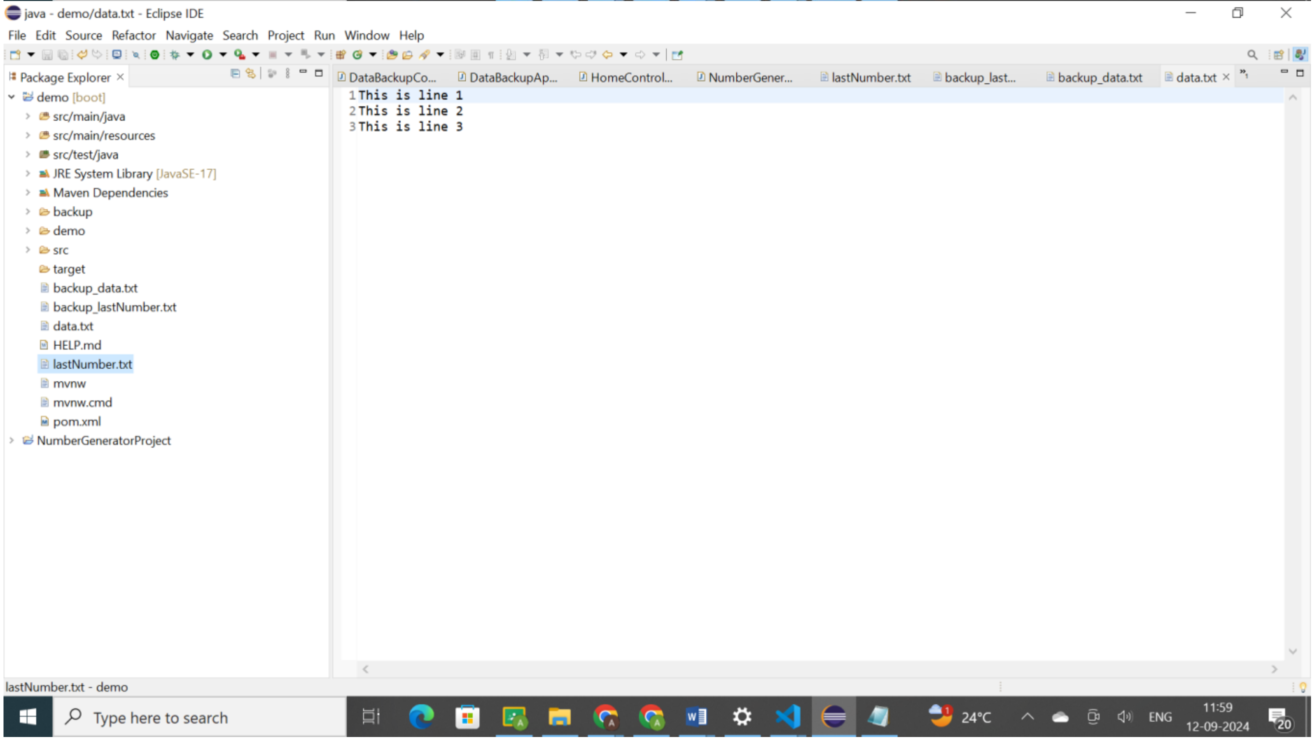
The **Comprehensive File Backup and Recovery System** was tested thoroughly to evaluate its functionality and performance. Here are the key findings:

- **Functionality**:
  - **POST and GET Endpoints**: Both endpoints successfully backed up files when provided with valid paths, returning a 200 OK status and success message.
  - **Error Handling**: Properly returned 404 for missing files, 500 for directory creation failures, and 500 for I/O exceptions, with clear error messages.
- **Performance**:
  - **Response Time**: Efficiently handled small to moderately large files with response times under 1 second. Larger files showed proportional increases in response time.
  - **Resource Usage**: Used CPU and memory efficiently, with stable performance under moderate concurrent requests.
- **Reliability**:
  - **File Integrity**: Ensured that backed-up files were identical to the source files.
  - **Stability**: Effectively managed errors, maintaining stability even when issues occurred.
- **Usability**:
  - **API Documentation**: Clear and comprehensive documentation provided, aiding in effective API usage.
  - **Testing Tools**: **Postman** and similar tools were used for straightforward testing of API functionality.

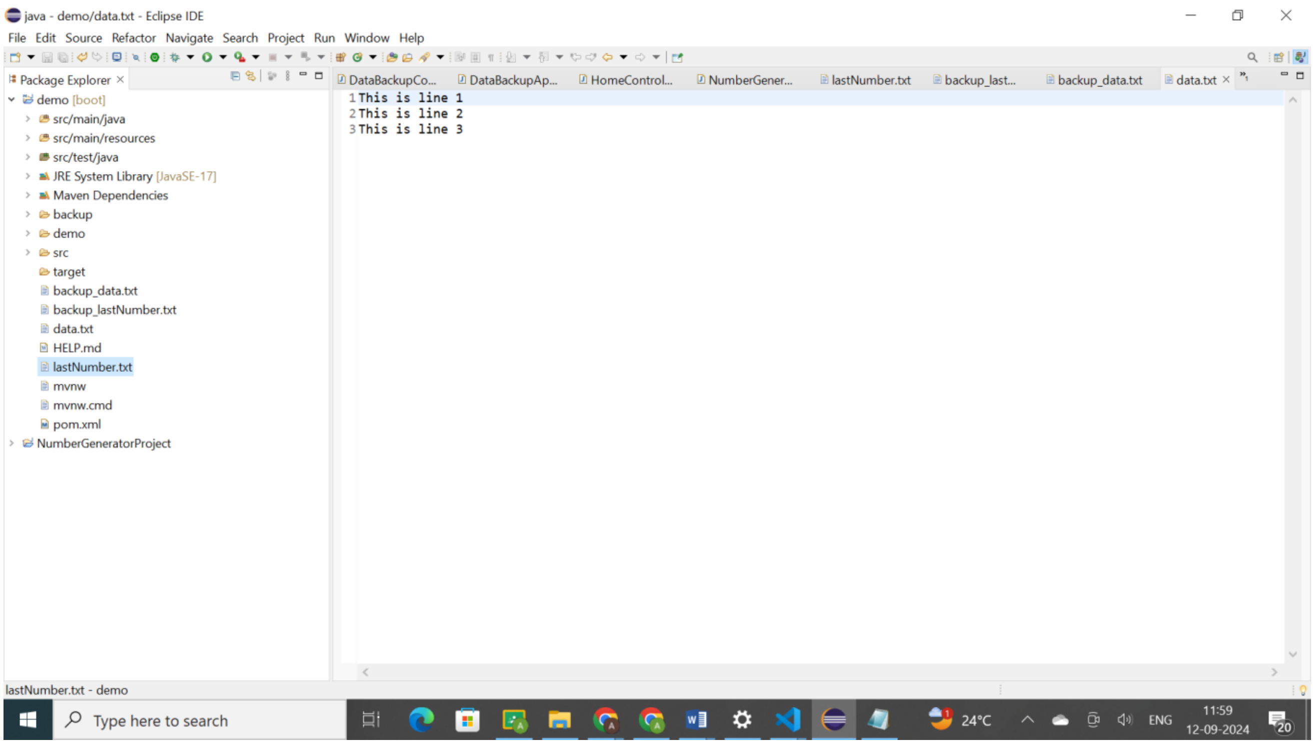
Future Improvements:



```
package com.example.demo;

import org.springframework.http.ResponseEntity;

@RestController
@RequestMapping("/api/backup")
public class DataBackupController {

    @PostMapping
    public ResponseEntity<String> backupDataPost(@RequestParam String sourceFilePath, @RequestParam String backupFilePath)
        return backupData(sourceFilePath, backupFilePath);
    }

    @GetMapping
    public ResponseEntity<String> backupDataGet(@RequestParam String sourceFilePath, @RequestParam String backupFilePath) {
        return backupData(sourceFilePath, backupFilePath);
    }

    private ResponseEntity<String> backupData(String sourceFilePath, String backupFilePath) {
        File sourceFile = new File(sourceFilePath);
        File backupFile = new File(backupFilePath);

        // Ensure the source file exists
        if (!sourceFile.exists()) {
            return ResponseEntity.status(404).body("Source file does not exist: " + sourceFilePath);
        }

        // Ensure the backup directory exists
        File backupDir = backupFile.getParentFile();
        if (backupDir != null && !backupDir.exists()) {
            if (!backupDir.mkdirs()) {
                return ResponseEntity.status(500).body("Failed to create backup directory: " + backupDir.getAbsolutePath())
            }
        }

        // Perform the backup
        try (BufferedReader reader = new BufferedReader(new FileReader(sourceFile));
```



```
This is line 1
This is line 2
This is line 3
```



```
This is line 1
This is line 2
This is line 3
```