**Visual VM**

VisualVM is a powerful and extensible Java profiling tool designed to monitor, troubleshoot, and analyse Java applications. Developed by Oracle, VisualVM provides a comprehensive set of features for both local and remote application monitoring. Its user-friendly graphical interface allows developers and administrators to gain deep insights into various aspects of a Java application's runtime behaviour.

**Key Features:**

* Real-Time Monitoring:

VisualVM provides real-time monitoring of Java applications, offering insights into CPU usage, memory consumption, threads, and classes.

* VisualGC:

The VisualGC plugin enables visual monitoring of garbage collection activities, allowing users to analyze heap memory usage and garbage collection efficiency.

* Heap Dump Analysis:

VisualVM allows users to capture and analyse heap dumps, facilitating the identification of memory leaks, object retention, and overall memory utilization.

* Thread and CPU Profiling:

The tool supports thread analysis and CPU profiling, helping developers identify performance bottlenecks and optimize application code.

* JMX (Java Management Extensions) Support:

VisualVM seamlessly integrates with applications exposing JMX, enabling users to monitor and manage application components through JMX connections.

* Plug-in System:

VisualVM supports a plugin architecture, allowing the addition of custom functionality and extensions to meet specific monitoring and profiling needs.

* Advanced Profiling:

The profiling capabilities include CPU profiling, memory profiling, and various other instrumentation options for in-depth analysis of application behaviour.

* Multiple JVM Support:

VisualVM can simultaneously monitor multiple Java Virtual Machines (JVMs) either on the same machine or across different machines.

* Diagnostic Commands Integration:

The tool integrates with Java diagnostic commands, providing users with access to a wide range of command-line tools for troubleshooting and analysis.

* Connectivity Options:

VisualVM supports both local and remote application monitoring, making it versatile for development, testing, and production environments.

**Prerequisites:**

* Java Development Kit (JDK):

Ensure you have a Java Development Kit (JDK) installed on your machine. VisualVM requires JDK 6 or later.

* Download VisualVM from the official website: VisualVM Downloads

**Installation and Setup:**

* Extract VisualVM:

Once downloaded, extract the VisualVM archive to a directory of your choice.

* Navigate to the visualvm.conf file in the etc folder. Change the jdk home path

visualvm\_jdkhome="C:\Users\Suraj\Downloads\jdk-17.0.9\_windows-x64\_bin\jdk-17.0.9"

* Run VisualVM:

Navigate to the bin directory inside the extracted VisualVM folder.

Execute the visualvm.exe (on Windows) or visualvm (on Unix-like systems) executable.

* The application will be already running with the unique process Id number. Click on the Monitor to get the graph and details

**OpenCSV**

OpenCSV is a widely-used Java library for effortless handling of CSV files. It simplifies CSV reading and writing with a straightforward API, supporting diverse formats. Key features include annotation-based mapping, custom data binding, error handling, and a memory-efficient streaming API. The library allows easy configuration, supports internationalization, and is actively maintained with a vibrant community. A simple example demonstrates reading and processing a CSV file, showcasing OpenCSV's versatility and user-friendly approach.

**Transaction Download using OpenCSV**

This Java code uses OpenCSV to generate a CSV file from a paginated list of EcollectDetails objects. It writes the CSV data to a ByteArrayOutputStream, converts it to a byte array, and then returns it as a downloadable file in the HTTP response. The code handles pagination, CSV creation, and response generation using Spring's ResponseEntity. Exception handling is in place for any potential I/O errors.

*int page = 0;*

*int size = 0;*

*EcollectDetailsListRequest ecollectListRequest = new EcollectDetailsListRequest();*

*ecollectListRequest.setPage(page);*

*ecollectListRequest.setSize(size == 0 ? (int) 200000 : 10000);*

*Page<EcollectDetails> ecollectDetailsList = fetchEcollectList(ecollectListRequest);*

*try (ByteArrayOutputStream outputStream = new ByteArrayOutputStream();*

*OutputStreamWriter writer = new OutputStreamWriter(outputStream);*

*CSVWriter csvWriter = new CSVWriter(writer)) {*

*// Write CSV header*

*String[] header = {"id", "transactionHashId", "beneAccountIfsc", "beneAccountNo", "creditAccountNumber"};*

*csvWriter.writeNext(header);*

*// Write CSV data*

*for (EcollectDetails ecollectDetails : ecollectDetailsList) {*

*String[] data = {*

*ecollectDetails.getTransactionHashId().toString(),*

*ecollectDetails.getTransferUniqueNo(),*

*ecollectDetails.getCustomerCode(),*

*ecollectDetails.getBeneAccountIfsc(),*

*ecollectDetails.getBeneAccountNo(),*

*ecollectDetails.getCreditAccountNumber(),*

*ecollectDetails.getBeneFullName(),*

*ecollectDetails.getTransferType(),*

*ecollectDetails.getTransferUniqueNo(),*

*ecollectDetails.getTransferTimestamp(),*

*ecollectDetails.getTransferCcy(),*

*};*

*csvWriter.writeNext(data);*

*}*

*// Flush the writer to ensure all data is written*

*writer.flush();*

*byte[] csvBytes = outputStream.toByteArray();*

*// Create HttpHeaders and set content type*

*HttpHeaders headers = new HttpHeaders();*

*headers.setContentType(MediaType.APPLICATION\_OCTET\_STREAM);*

*headers.setContentDispositionFormData("attachment", "data.csv");*

*return new ResponseEntity<>(csvBytes, headers, HttpStatus.OK);*

*} catch (IOException e) {*

*// Handle exception if necessary*

*e.printStackTrace();*

*return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);*

*}*

*}*

The below graph was obtained when the API was hit. The heap memory utilized was 500 Mb

