Name: ALI HASSAN BEK

Matricola: 217105

Agile Project

GITHUB LINK:- https://github.com/Dark77M/agiel_cli_project

Spring Shell is a framework built on top of the Spring Framework, designed for creating interactive command-line applications in Java. It simplifies the development of command-line interfaces, offering features like integration with Spring, extensibility, annotation-based configuration, scripting support, tab completion, and compatibility with other Spring projects. Developers can easily build robust and user-friendly command-line applications using Spring Shell.

This Java class is annotated with @ShellComponent, indicating that it's part of a Spring Shell application. It defines two methods annotated with @ShellMethod:.mstartActivity: This method starts a new activity, taking an optional -n or --activity-name parameter. It checks if there's already an activity in progress and logs the start time and activity name. If an exception occurs, it prints an error message and resets the activity state.StopActivity: This method stops the currently running activity, logging the stop time and activity name. It checks if there is an activity in progress and prints a message if not. The class maintains state variables (isActivityStarted and activityName) to track the status of the activity. Overall, this class provides a simple command-line interface for starting and stopping activities, logging relevant information along the way.

However, for run this one for example (Shell start -n activity 1) The output going be with the time and date and svaeLog.

```
5 usages
String activityName;

no usages  Dark77M
@ShellMethod(key = "start",value = "start activity by using name")
public void startActivity(@ShellOption(value = {"-n","--activity-name"})String name){
    try {
        if (isActivityStarted){
            System.out.println("Stop activity before start a new one!!");
            return;
        }
        String logInfo = String.format("%s;%s;%s;%s", Commons.getDate(),Commons.getTime(),"start",name);
        Commons.saveLog(logInfo);
        isActivityStarted = true;
        activityName = name;

    }catch (Exception ex){
        System.err.println(ex.getMessage());
        isActivityStarted = false;
        activityName = "";
    }
}
no usages  Dark77M
```

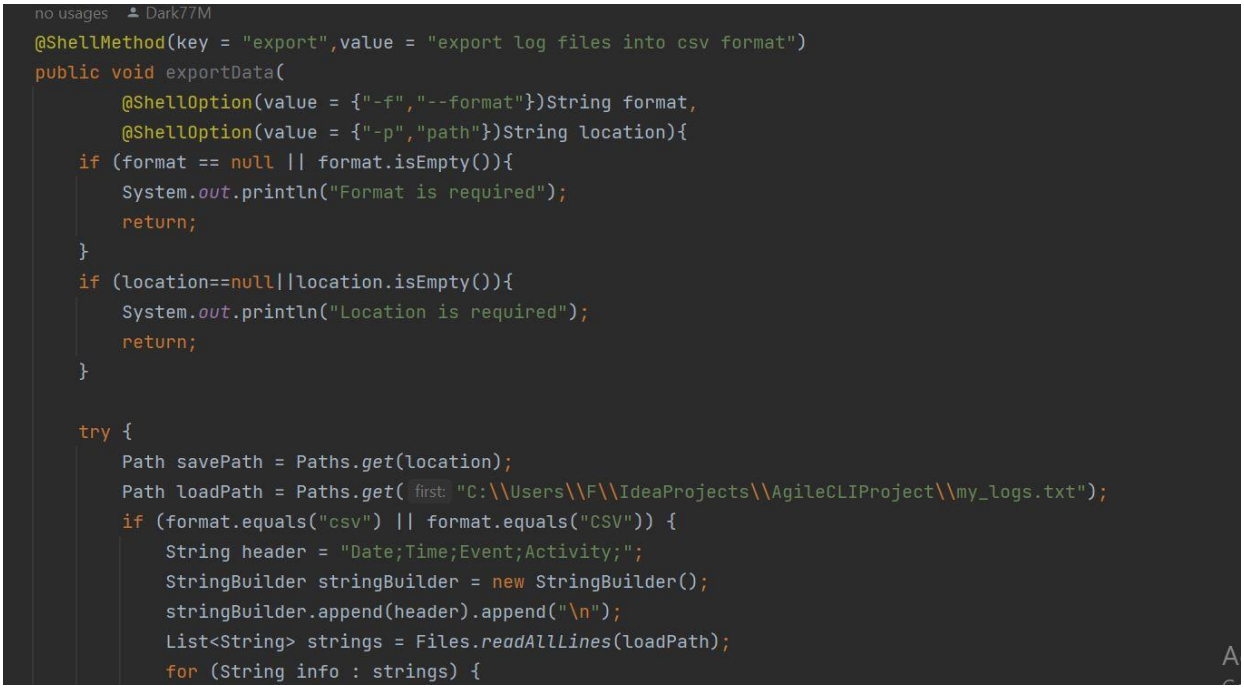
```
public class Commons {
    2 usages
    static Logger log =Logger.getLogger(ProjectCommand.class.getName());
    3 usages
    static FileHandler fileHandler;

    no usages
    public static final String[] WEEKDAY = {"MONDAY","TUESDAY","WEDNESDAY","THURSDAY","FRIDAY"};
    2 usages  Dark77M
    public static String getDate(){
        try {
            LocalDate currentDate = LocalDate.now();
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
            return currentDate.format(formatter);
        }catch (Exception ex){
            System.err.println(ex.getMessage());
        }
```

This Java class method, annotated with @ShellMethod, defines a command for exporting log files into different formats, such as CSV or JSON. It takes two parameters: Format: Specifies the desired export format (CSV or JSON). Location: Specifies the path where the exported file should be saved. The method checks for required parameters, and if they are missing, it prints error messages. It then reads log data from a specified file, processes it based on the chosen format, and writes the transformed data to the specified location. Also, For CSV format, it extracts relevant information from log lines and creates a CSV file. For JSON format, it parses log lines, builds JSON objects, and creates a JSON array. Furthermore, Any

exceptions during the process are caught and their error messages are printed. Overall, this method provides a command-line interface for exporting log data in different formats.

However, to run this one you should write (export -f csv/json -p /AND/YOUR/PathWhereYouWantSaveFile/)

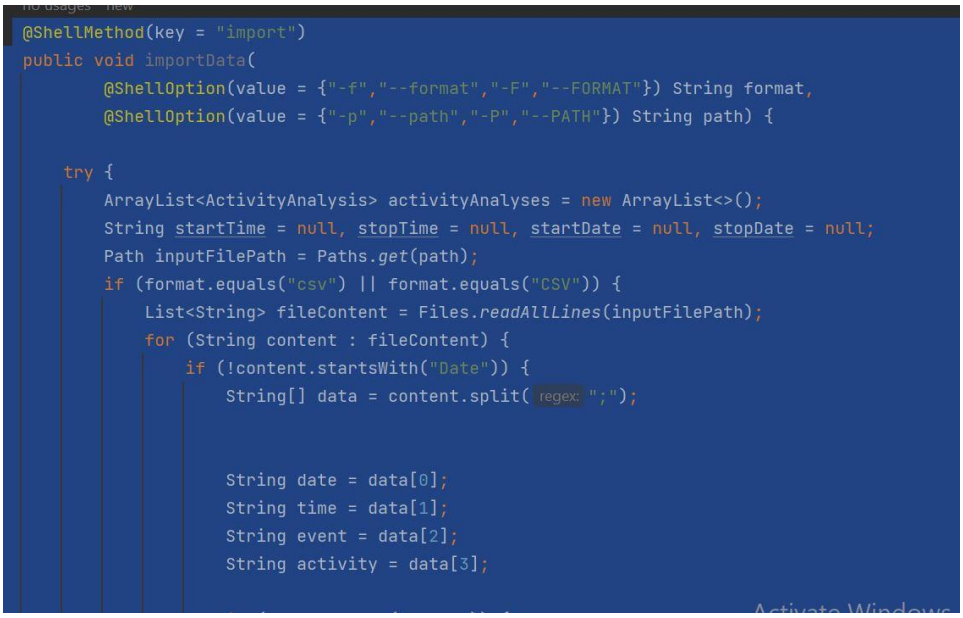


```
no usages  Dark77M
@ShellMethod(key = "export",value = "export log files into csv format")
public void exportData(
        @ShellOption(value = {"-f","--format"})String format,
        @ShellOption(value = {"-p","path"})String location){
    if (format == null || format.isEmpty()){
        System.out.println("Format is required");
        return;
    }
    if (location==null||location.isEmpty()){
        System.out.println("Location is required");
        return;
    }

    try {
        Path savePath = Paths.get(location);
        Path loadPath = Paths.get( first: "C:\\Users\\F\\IdeaProjects\\AgileCLIProject\\my_logs.txt");
        if (format.equals("csv") || format.equals("CSV")) {
            String header = "Date;Time;Event;Activity;";
            StringBuilder stringBuilder = new StringBuilder();
            stringBuilder.append(header).append("\n");
            List<String> strings = Files.readAllLines(loadPath);
            for (String info : strings) {
```

Third thing I done it is import .This Java class method, annotated with @ShellMethod, defines a command for importing data from a specified file in CSV format. It takes two parameters. format: Specifies the format of the input file (CSV). Path: Specifies the path of the input file. The method reads the content of the CSV file, parses each line, and extracts information related to activity start and stop events. It calculates the time difference between start and stop events, as well as the day of the week for each activity. The extracted data is then used to create instances of Activity Analysis and stored in an ArrayList. This method provides a command-line interface for importing and processing CSV data, converting it into a structured format for further analysis or display. It's a useful functionality for handling activity data in a more organized manner. However, I done the same function for the Json file .

However, to run this one you should write (import -f csv-json -p /AND/YOUR/PathWhereYouSavedFile/)



```
@ShellMethod(key = "import")
public void importData(
        @ShellOption(value = {"-f","--format","-F","--FORMAT"}) String format,
        @ShellOption(value = {"-p","--path","-P","--PATH"}) String path) {

    try {
        ArrayList<ActivityAnalysis> activityAnalyses = new ArrayList<>();
        String startTime = null, stopTime = null, startDate = null, stopDate = null;
        Path inputFilePath = Paths.get(path);
        if (format.equals("csv") || format.equals("CSV")) {
            List<String> fileContent = Files.readAllLines(inputFilePath);
            for (String content : fileContent) {
                if (!content.startsWith("Date")) {
                    String[] data = content.split( regex: ";");


                    String date = data[0];
                    String time = data[1];
                    String event = data[2];
                    String activity = data[3];
```

In my project was the last requirements are the following analytics:

– Cumulative time spent on each task; – Average time spent working on each week day;

– Average tracked time per day;. I create entities for the import data.

Firstly, Java method cumulativeTimeOnEachTask, calculates the cumulative time spent on each unique activity from a list of ActivityAnalysis objects and stores the results in a HashMap. It iterates through the activities, updating the cumulative time for each one.

```
private void cumulativeTimeOnEachTask(ArrayList<ActivityAnalysis>
activityAnalyses) {
    Map<String, Long> cumulativeTimeMap = new HashMap<>();

    try {
        activityAnalyses.forEach(activityAnalysis -> {
            if
(!cumulativeTimeMap.containsKey(activityAnalysis.getActivityName())){

cumulativeTimeMap.put(activityAnalysis.getActivityName(),activityAnalysis.get
Duration());
            }else{
                long previousTime =
cumulativeTimeMap.get(activityAnalysis.getActivityName());
                previousTime+=activityAnalysis.getDuration();
                cumulativeTimeMap.remove(activityAnalysis.getActivityName());
                cumulativeTimeMap.put(activityAnalysis.getActivityName(),
previousTime);
            }
        });
```

Second step was this averageTimeInWeekDays, prints the average time spent working on each weekday based on a list of ActivityAnalysis objects and the Commons.WEEKDAY array. It uses Java streams to calculate counts and sums for each weekday.

```
private void averageTimeInWeekDays(ArrayList<ActivityAnalysis>
activityAnalyses){

System.out.println("=====================================================");
    System.out.println("|== Average time spent working on each week day ==|");

System.out.println("=====================================================");

    for (String wd : Commons.WEEKDAY){
        long wdCount = activityAnalyses.stream().filter(activityAnalysis ->
activityAnalysis.getWeekDays().equals(wd)).count();
        long wdSum = activityAnalyses.stream().filter(activityAnalysis ->
activityAnalysis.getWeekDays().equals(wd))
                .mapToLong(ActivityAnalysis::getDuration).sum();
```

Last requirement was averageTimePerDay, calculates and prints the average tracked time per day for a list of ActivityAnalysis objects using Java streams. It iterates through distinct dates, calculates count, sum, and average duration, and prints the results.

```
private long getTimeDifference(String startTime, String stopTime){
    LocalTime sTime = LocalTime.parse(startTime);
    LocalTime eTime = LocalTime.parse(stopTime);

    Duration timeDifference = Duration.between(sTime, eTime);

    return timeDifference.getSeconds();
}
```