Adam Hill
William Nyffenegger
Milestone 2
5th March 2016

Introduction

The basic server for milestone two is enclosed. Compile the server and its dependencies with compile.sh, which runs the command make all on the folder. This creates the bin folder with all of the class files and the external jars in it. To run the compiled server, run the shell script run.sh like so: run.sh port# or run.sh -p port# -d filename. Once opened, the server listens on the specified port for input. It does not reply to empty input, but replies to every other type of input.

There is a test script labled script1.sh that can be run to test the server. This can also be used as an example of what to do if this documentation and the included READ.ME is not enough. We recommend not to use piping though, as script1.sh does, to create a better seamless experience.

Architecture

Classes in the server

- Handler.java
- Parser.java
- LogicEngine.java
- BackEnd.java

Brief Overview of Classes

1. Handler.java: contains the main class for the server. Once opened it waits for a client to connect and then loops through the input from the client. It uses Parser.java to parse the initial settings from the command line for the server and LogicEngine.java to parse input from clients.
2. Parser.java: uses Apache CLI to parse the command line given to the user. Will error if an invalid command is given.
3. LogicEngine.java: parses input strings, executes commands, and builds responses to client. LogicEngine uses BackEnd to conduct all database transactions. Does not error unless database does not exist or database is locked.
4. BackEnd.java: SQLite JDBC support for the server. Stores all projects and tasks. Should not produce error unless database does not exist or database structure is foreign (you've picked a random database for it to use) even then it will be hard to break.

User Manual

The user manual is split into a discussion of the classes created to bring the server to life. Classes are listed in order of their significance and their prevalence for the client of the server. As stated in the overall structure, this whole thing is ultimately pretty simple.

Commands

This server supports only a few commands, which will be outlined here. Note the placement of colons and semicolons, as it is critical for the request to parse correctly. Everything is case-sensitive, and all commands MUST be in all capital letters with underscores in place of spaces.

- PROJECT_DEFINITION:projectName;TASKS:numTasksN;taskNameN1;N1startTime; N1endTime;taskNameN2; N2startTime;N2endTime; … and so on until N tasks are filled

  PROJECT_DEFINITION – Command. Declares a project to be added to the database
  projectName – user-defined name of the project
  TASKS – required command to follow immediately after the project name
  numTasksN – the number of tasks to be included in the project. Each task must be given a name, start time, and end time. The next task will be listed immediately after the end time of the previous task
  NstartTime – the time the task will be started
  NendTime – the time the task will be completed

- TAKE;USER:userName;PROJECT:projectName;taskName

  TAKE – Command. Declares that a user will be claiming a specific task in a project
  USER – Required. Indicates that the user's name will be given next
  userName – the name of the user claiming the task
  PROJECT – Required. Indicates the project name will be given next
  projectName – this specifies the already existing project from which the user will be claiming a task from
  taskName – the specific task in the given project that the user is claiming

- GET_PROJECTS

  GET_PROJECTS – Command. Lists all of the projects currently defined in the database

- GET_PROJECT;projectName

  GET_PROJECT – Command. Tells the server to fetch an existing project with the specified name from the database, and list its tasks
  projectName – the specific project to be listed

Time Format

All times used in commands given to the server are in UTC. That is:

YYYY-MM-DD:HHhMMmSSsXXXZ

Where all capital letters (except Z) are actual numbers, and all lowercase letters (also except Z) are given to the server as the literal character they represent. Letters before the colon correspond to the year, month, and day respectively, and letters after the colon represent the time in hours, minutes, seconds, and milliseconds. The letter Z tells us that the time is specifically formatted in UTC, and it must be part of the command for parsing to work correctly.

Example:

2016-03-05:01h20m36s512Z
March 5, 2016 at 1:20:36 A.M. and 512 milliseconds
(the time at which this section of the documentation was written)

Exiting The Server

Use either of the following commands to exit the server safely with exit code 0:

EXIT;QUIT

Class Descriptions

Class	**Handler.java**
Description	Brings up TCP socket connection and reads commands until ended. Program can be closed by using the command EXIT or QUIT. Accepts a single connection at a time. If a connection is closed, then the server waits for a new client to open another connection. Main class that runs the server
Methods	public static void **main**(final String [] args) Bring the server online by interpreting the command line arguments and accept input until a connection is closed and wait for the next connection. *@param* **args** — command line arguments **-p port# -d databasefile.db**
	private static LinkedList<String[]> **setArgs**() Returns the expected format to be input into CLI as the options used to bring the server online. Basic version of input to getopt_long *@return* returns the arguments as a LinkedList of String arrays where each array is a command

Class	**LogicEngine.java**
Description	Interprets lines of input from the server into actions in the database and replies. This is the font end to the database handling done in BackEnd.java. LogicEngine does not handle opening or closing a socket. It handles a single line of input (string of input ending in a new line) and outputs the result from executing that input.
Constructor	public **LogicEngine**(String dbLocation) throws SQLException Checks whether proposed database location exists and if a database does not already exist with the given name then creates a database. *@param* **dbLocation** — proposed location of the database *@throws* **SQLException** if creating/opening database fails
Methods	public String **parseInput**(String input, String IP, int port) throws SQLException Given a single line of input from the client. Execute and analyze that input. On success prepend each individual command with OK and output the appropriate response. On failure prepend the command with Fail and print out all subsequent failures. *@param* **input** — string of commands (as many as won't break the JVM) *@param* **IP** — the IP of the client which sent commands *@param* **port** — which port they were sent from on client's computer *@return* **output** — response to parsing and executing commands *@throws* **SQLException** if connection fails to close
	private static void **projectOutput**(StringBuilder output, String[] commands, int commandsLength, int index) When a project is created successfully this specifies the routine for appending that project back to the output. Created specifically because of the number of tasks associated with the project output. Is only called on successful PROJECT_DEFINITION and does not return anything *@param* **output** — string builder *@param* **commands** — list of commands *@param* **index** — index of the command to begin indexing from *@param* **tasksIndex** — the index of the last token of the tasks associated with the project

	private static void **failureFormat**(StringBuilder output, String[] commands, int commandsLength, int index) For whenever a failure occurs in parsing data. Adds fail for all remaining commands after encountering a bad command in input to the StringBuilder included in the input. This outputs all remaining data from the input after the failure, but first prepending the word "Fail" to the output. *@param* **output** — string builder *@param* **commands** — list of commands *@param* **commandsLength** — total number of commands *@param* **index** — index to begin failure formatting from
	private boolean **checkStatus**(String project, LinkedList<String[]> tasks) Given a project, go through all of the projects tasks. If those tasks have gone from waiting to done, change their status in the table to done. *@param* **project** — project name *@param* **tasks** — the number of tasks to read *@return* whether all status were successfully checked. Will return false if a table is locked or corrupted
	private int **isDone**(String end) Checks whether a task is past its completion point *@param* **end** — string representing the time of completion *@return* compareTo() output after string has been formatted for simple date format. Zero or greater means that the task has finished.
	static void **appendOutput**(StringBuilder output, String append) Given an output StringBuilder, append the specified string to the StringBuilder *@param* **output** — an already created string builder *@param* **appended** — string to append to output
	public void **closeLogicEngine**() Close the connection to the database which effectively ends LogicEngine. Should be run every time a logic engine is released from memory.

Class	**BackEnd.java**
Description	Class for doing all back end management required to run the server. Contains methods for opening connections to databases, adding projects, and adding tasks. It also contains methods for the retrieval of information from the database. The intended use of BackEnd is to support LogicEngine and nothing else. A list of the most important methods in the classes follows: createProject, insertTask, setUser, setStatus, getTasks, getProjects, openConnection, closeConnection
Constructor	public **BackEnd**(String dbPath) throws SQLException Creates database if not already created with projects_list. Each instance is ready to handle queries and updates for the database that is input into its system. *@param* **dbPath** — the relative path to the file. May error if program does not have write permissions to location. *@throws* **SQLException** error caused when database is locked, does not exist, or directory does not exist or permissions are not given
Methods	public boolean **createProject**(Connection conn, String projectName, int tasks) Creates a project by inserting it into the list of projects in the database and creating a table for all of the tasks in the project. If project is already created, it replaces the project completely. Note: Erases project completely if it already exists so there is an implicit assumption that project names are unique. Prepending with times may be wisest solution to prevent erasing. *@param* **conn** — currently opened connection to an sqlite database *@param* **projectName** — name of the project wished to be created *@param* **tasks** — number of tasks in the project *@return* whether project was created successfully
	public LinkedList<String> **getAllProjects**(Connection conn) throws SQLException Go to the PROJECTS_LIST table and return all of the projects as a list of strings (names) *@param* **conn** — currently opened connection to an sqlite database *@return* string of names. Returns "Failure" as first and only string in list if it fails to read from the database This is weird but also surprisingly useful.

	public void **setStatus**(Connection conn, String project, String task, int status) Set the completion status of a project if the project exists. Cannot be called on nonexistent task *@param* **conn** — currently open sqlite database connection *@param* **project** — project name *@param* **task** — task name *@param* **status** — integer status (0 = done, 1 = waiting)
	public boolean **setUser**(Connection conn, String project, String task, String user) Set the owner of a task. *@param* **conn** — currently open connection to an sqlite database *@param* **project** — already created project name *@param* **task** — already created task name *@param* **user** — the owner to be added *@return* whether adding owner was successful
	public boolean **insertTask**(Connection conn, String projectName, String task, String start, String end, String IP, int port) Insert a task into the given project's table of tasks *@param* **conn** — currently open connection to sqlite database *@param* **projectName** — already created project *@param* **task** — name of proposed task addition *@param* **start** — beginning time of task *@param* **end** — completion time of task *@param* **IP** — Client IP that task was sent from *@param* **port** — Client port task was sent from *@return* whether task was added successfully
	public boolean **isValidDate**(String date) Takes an input date and tests it for the specified string pattern. If that pattern is not as expected it returns that the input date is a bad creation of the expected time format. *@param* **date** — start or end time to be evaluated *@return* whether string represents a properly formatted date or not

	public LinkedList<String[]> **getTasks**(Connection conn, String project) Get all of the tasks in a project and return them as a linked list of string arrays *@param* **conn** — currently open sqlite connection *@param* **project** — project name *@return* LinkedList<String[]>. If the list is empty then it failed to find any tasks for the project (errored).
	private String **getTaskTable**(String projectName) Each project has a different task project list. Problem queries have their issue characters replaced *@param* **projectName** — project name *@return* gets the unique task table name for a project
	public int **getNumberTasks**(Connection conn, String project) Get the number of tasks associated with a project *@param* **conn** — currently open connection to sqlite database *@param* **project** — project name to get number from *@return* number of tasks for project
	private void **openDatabase**(String dbFile) throws SQLException Run at startup to make sure the database is accessible and has a projects table *@param* **dbFile** — proposed location for database *@throws* **SQLException** — if the database location is not accessible or writable
	public Connection **openConnection**() throws SQLException Opens connection to sqlite database with the location given the constructor *@return* an open sqlite Connection to the database *@throws* **SQLException** — if the database is locked or there are concurrency issues
	public void **closeConnection**(Connection c) throws SQLException Close a connection to the database *@param* **c** — currently open connection to database *@throws* **SQLException** if the connection cannot be closed

Class	**Parser.java**
Description	Interprets options and command lines passed to it from the Handler or potentially from other sources in the future.
Constructor	public **Parser**(LinkedList<String[]> options) throws IllegalArgumentException Takes in the list of options and tries to build a command line. *@param* **options** — list of string arrays where each array is an option *@throws* **IllegalArgumentException** if improper number of arguments were given
Methods	public Options **getOptions**() Returns list of command line options associated with the input that was given. *@return* Options object which contains all of the options a command line may contain
	public CommandLine **getCMD**(Options opts, String[] args) throws ParseException Evaluates a string array representing command line arguments for commands and returns its interpretation of the array as a CommandLine containing all relevant options. Will throw a parse exception if an illegal argument is entered. *@param* **opts** — options to be used if a different set of command line options is desired *@param* **args** — command line represented as string array *@return* **CommandLine** — the parsers interpretation of the command line arguments. *@throws* **ParseException**
	public CommandLine **getCMD**(String[] args) throws ParseException Given an array of strings representing the tokens entered on the command line, return the already created command line parser's interpretation of those commands with the options initialized at run time. *@param* **args** — string array of tokens from command line *@return* **CommandLine** — parser's interpretation of the given commands *@throws* **ParseException**

	public static CommandLineParser **getDefaultParser**() { Returns a parser for use by whoever desires *@return* **CommandLineParser** — parser for interpreting command lines represented as string arrays of tokens

Conclusion

This implementation of a server, while somewhat limited, completes the task it was intended to handle, quickly and efficiently. The JDBC API was used for interfacing with the SQLite database and the Apache CLI API was used for some of our string parsing. Both have proved helpful in dealing with our problems. We have thoroughly tested and debugged the program and we can say that it is basically impossible to break unless the specified database directory does not exist, or the database has been closed in some way.