Adam Hill
William Nyffenegger
Milestone 3
2016

Introduction

An enhanced version of the server from milestone four 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.

The server supports multithreading with TCP and a single UDP thread for all tasks. To engage the server just access the correct IP and port. If the server is shut down unexpectedly, it will exit without locking the database and finish all current commands.

The server sends messages using the given ASN1 library and supports clients for sending messages. Enter text in the previously standard text format. The text is parsed by the ClientParser class and encoded or decoded depending on the direction of its arrival.

Starting a client or server instance is simple. Compile with compile.sh and run the server with run.sh. Be mindful that the server is compiled with java7. Run the client with client.sh. Client.sh takes the ip, and port as required arguments, and the type of communication as optional.

There are two test scripts script1ASN_UDP.sh and script1ASN_TCP, 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 the scripts create a more seamless experience.

Architecture

Classes in the server

- Handler.java
- Parser.java
- BackEnd.java
- TCPThreadedServer.java
- TCPHandler.java
- UDPHandler.java
- UDPDecoder.java
- UDPEventHandler.java
- ProjectReporter.java
- SendTracked.java
- ServerDecoder.java

Classes in the Client

- Client.java
- ClientParser.java
- ClientListener.java

- OptParser.java

<u>Abstracted Datatypes in the datatypes</u>

- EnterLeave.java
- Task.java
- Take.java
- Project.java
- GetProject.java
- Projects.java
- ProjectOK.java

<u>Abstracted Datatypes in the datatypes</u>

- ASN1Task.java
- ASN1Project.java
- ASN1ProjectOK.java
- ASN1GetProject.java
- ASN1GetProjects.java
- ASN1Projects.java
- ASN1Enter.java
- ASN1Leave.java

Overview of Classes:

<u>Summary of ASN1Datatypes</u>

The datatype classes are meant as abstractions to support the ASN1 classes and provide uniform functionality. They correspond almost completely with the ASN1* counterparts. All of the ASN1 types have the requisite encoder and decoder methods. The constructors for the ASN1 types take the abstracted version of themselves as a parameter, no parameters, and/or a specific name. All of the ASN1 types return an abstracted version of themselves when decoded. Their documentation will not be covered later for brevity's sake.

<u>Brief Overview of Client Classes</u>

1. Client.java: runs as either a UDP or TCP client to the server set at run time. The program needs the port and address of server to begin running; however, it runs as a plain text client in the command line afterward.
2. ClientParser.java: parses all of the output and input a client may have. Prints input to the client to the screen and prepares a byte array sent to the server.
3. ClientListener.java: listens for all udp packets arriving at the specified socket if the client is running in UDP mode.
4. OptParser.java: parses input commands to the client on startup

<u>Brief Overview of Server Classes</u>

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. TCPThreadedServer.java: handles all TCP requests for the server by opening new TCPHandler threads for each requested interaction from an IP and port combination. Those threads, once started, are independently executing and do not require any interaction to shut down or continue running. Implements Runnable and Shutdown Hooks for graceful termination

3. TCPHandler.java: manages a single interaction between a client IP and port and the server. At the end of the interaction, it closes itself. In the case of the server being shutdown, the thread will terminate gracefully.
4. UDPHandler.java: handles all UDP packet interactions by receiving client packets and replying to those packets. Takes a maximum buffer of 2^{15} bytes.
5. ServerDecoder.java: decodes all ASN1 packets that arrive at the server with a single public static method serverQuery(), which decodes and replies to the origin of the query.
6. Parser.java: uses Apache CLI to parse the command line given to the user. Will error if an invalid command is given.
7. BackEnd.java: SQLite JDBC support for the server. Static class containing all methods for interacting with the SQLite database. One critical method is openDatabase() which checks for the existence of a SQLite database with a basic framework.
8. UDPDecoder.java: instance associated with each UDPHandler that decodes all input from UDP packets and then executes the database updates and queries related to the input and returns a byte array formatted in ASN1 for a reply.
9. UDPEventTracker.java: all projects that a UDPClient is listening to and their associated tasks to be reported.
10. ProjectReporter.java: list of scheduled tasks that will reported to a listening UDPClient.
11. SendTracked.java: extends TimerTask and may be scheduled to be sent after a delay of time through a project reporter.

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 Client

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

EXIT;

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	**TCPThreadedServer.java**
Description	Listens for all client connections to the server and then starts a TCPHandler for each connection. Closes on the shutdown of the server or a failure to bind/use the socket
Constructor	public **TCPThreadedServer**(int port, String dbLocation) Checks whether proposed database location exists and if a database does not already exist with the given name then creates a database. *@param* **port** – port server listens on for TCP connections *@param* **dbLocation** — proposed location of the database
Implements	Runnable
Methods	public String **run**() Opens socket to port and creates threads for each client connection until the server is closed. Every connection created is independent once it is started.
	public **terminate**() Tells the connection handler to stop accepting new input and finish interpreting all current queries, then exit

Class	**TCPHandler.java**
Description	Handles an individual client's requests until the client ends the connection or the server is shut down
Constructor	public **TCPHandler**(Socket sock, String dbLocation) Checks whether proposed database location exists and if a database does not already exist with the given name then creates a database. *@param* **sock** – already opened connection to client *@param* **dbLocation** — proposed location of the database
Implements	Runnable
Methods	public String **run**() Accepts information from the client and responds to the client until the client closes the connection. Will close cleanly
	public **terminate**() Tells the TCP handler to stop accepting new input and finish interpreting all current queries, then exit

Class	**UDPHandler.java**
Description	Handles all UDP packets and responding to those client packets
Constructor	public **UDPHandler**(int port, String dbLocation) Checks whether proposed database location exists and if a database does not already exist with the given name then creates a database. *@param* **port** – port to open datagram socket to *@param* **dbLocation** — proposed location of the database
Implements	Runnable
Methods	public String **run**() Opens a UDP socket and begins accepting and responding to client packets until the server is shut down.
	public **terminate**() Tells the UDP handler to stop accepting new input and finish interpreting all current queries, then exit

Class	**UDPDecoder.java**
Description	Decodes requests sent by UDP and tracks all of the registered clients and tasks.
Constructor	public **UDPDecoder**() Sets the last updated time for UDP registered tasks to the time that the instance is created and then creates a list of UDPEventTrackers to track the tasks and projects by each client.
Methods	public byte[] **serverQuery**(String _dbfile, SimpleDateFormat, sdf, Decoder dec, InetAddress packet_address, int port) throws SQLException Decodes UDP packets sent to the server and modifies the list of registered clients as necessary. *@param* **_dbfile** – location of the database on the local system *@param* **sdf** – format for all dates sent in (standard is UTC) *@param* **dec** – already created decoder with the byte array from the packet placed in it *@param* packet_address – host name from which the packet originated from *@param* **port –** the port on the host that the packet originated from @throws **SQLException** – if the packets logic does not fit in the database @return the response packet in the form of a byte array
	private static int **queryProject** (Connection conn, SimpleDateFormat _sdf, Project p) Adds the project to the database and updates as necessary *@param* **conn** – already opened connection to the database *@param* _sdf – format for all dates sent in (standard is UTC) *@param* p – decoded project to be entered into the database @return 0 for success and any other number for failure to update the database.
	private static int **queryTake** (Connection conn, Take take) Sets the owner of a project *@param* **conn** – already opened connection to the database *@param* **take –** decoded take to be entered into the database *@return* 0 for success and any other number for failure to update the database.
	private static int **queryGetProjects** (Connection conn)

	Returns the names of all projects currently in the database *@param* **conn** already opened connection to the database @return list of projects in the database
	private static int **queryGetProjectsUnabridged** (Connection conn, SimpleDateFormat sdf) throws ParseException Returns the names and body of all projects in the database. *@param* **conn** – already opened connection to the database @throws **ParseException** – if the dates retrieved from the database cannot be formatted into the desired date format @return list of projects in the database as an object
	private static int **queryGetProject** (Connection conn, SimpleDateFormat sdf, String project) throws ParseException Adds the project to the database and updates as necessary *@param* **conn** – already opened connection to the database *@param* **sdf** – format for all dates sent in (standard is UTC) *@param* **project** – name of the projec @return 0 for success and any other number for failure to update the database.
	private static int **executeEnter** (Connection conn, SimpleDateFormat sdf, EnterLeave el, InetAddress ip, int port) Register a client to receive information on when tasks begin *@param* **conn** – already opened connection to the database *@param* **sdf** – format for all dates sent in (standard is UTC) *@param* **el** – decoded register command with list of projects for client to register to *@param* **ip –** host name for the client *@param* **port** – port name for the client @return 0 for success and any other number for failure to update the database.
	private static int **executeLeave** (EnterLeave el, InetAddress ip, int port) Deregister a client from the included projects in the EnterLeave object. *@param* **el** – decoded register command with list of projects for client to register to *@param* **ip –** host name for the client

	@param **port** – port name for the client @return 0 for success and any other number for failure to update the database.
	private static int **isDone** (final Date end) Register a client to receive information on when tasks begin *@param* **end** – check whether the completion date has passed for a task @return comparison between current date and now, if an exception occurs return an extraordinary value
	private static int **checkStatusTrackers** (final Date end) Check whether all currently registered UDPEventTrackers need to be updated and update them as necessary. If no more tasks remain for the event trackers then remove the event tracker.

Class	**UDPEventTracker.java**
Description	For each client this tracks the projects and tasks of those projects that client wants reports on. It is actively modified by the client to reflect completed reports and is removed if it is empty.
Constructor	public **UDPEventTracker**(InetAddress ip, int port) Initializes list of project reporters and set the ip and port (which serve as the unique identifier for the UDPEventTracker) *@param* **ip** – client's host *@param* **port –** client's port
Methods	public **update**() Checks to see if any project reporter has completed reporting all tasks and removes the project reporter if that case is found.
	public int **size**() Number of ProjectReporters in the event tracker.
	public int **addAllTasks**(String project, LinkedList<Task> tasks) Add all of the tasks for the associated project—that occur in the next hour—into the already existing ProjectReporter or into a newly created ProjectReporter *@param* **project** – name of the project which the tasks correspond to

	@param **tasks** – list of the tasks to be added to the project *@return* the number of tasks actually added to the event tracker
	public void **remove**(LinkedList<String> project) Remove all of the listed projects from the event tracker. This corresponds to a leave command. *@param* **project** – list of the projects to remove from the event trackers
	public boolean **equals**(InetAddress ip, int port) Check whether the event tracker corresponds to the ip and port of a received packet with a leave or enter command. *@param* **ip** – the ip of the received packet *@param* **port –** the port of the received packet *@return* whether the port and ip combination matches that of the event tracker
	public ProjectReporter **getPR**(String project) Returns the project reporter for the corresponding project name. *@param* **project** – name of the project to be retrieved *@return* the corresponding ProjectReporter
	public boolean **contains**(String project) Checks whether a project reporter already exists for the project. *@param* **project** – name of the project to be retrieved *@return* whether a ProjectReporter exists for the project

Class	**ProjectReporter.java**
Description	Handles all UDP packets and responding to those client packets
Constructor	public **ProjectReporter**(Timer t, String project) Sets all of the fields to the constructor values *@param* **t** – Timer which all of the projects tasks were placed in *@param* **project –** set the owner project of the project reporter

Methods	public boolean **equals**(String project) Checks whether the project owner is the same as the project inquired *@param* **project –** project name to be compared to *@return* whether project is contained in
	public boolean **active**() Check whether any of the scheduled tasks have not been executed *@return* whether a task has yet to execute or not
	public boolean **kill**() Cancel all tasks that have not yet been reported *@return* cancel all of the tasks that have not been reported
	public boolean **addTask**(Task t) If a task is scheduled to begin in the next hour, then add that task to the list of tasks to be reported. *@param* **t –** task to add *@return* false if the task is not to be reported and true if the task is
	public int **addAllTasks**(LinkedList<Task> tasks) If any of the tasks occur in the next hour, then add those tasks into the list of tasks to be reported on. *@param* **tasks –** list of tasks that may need to be reported *@return* the number of tasks actually added to the list of tasks to be completed

Class	**SendTracked.java**
Description	
Constructor	public **SendTracked**(String project, Task t) Creates a project with a corresponding task to be sent as an ASN1 byte array in a data packet. *@param* **project** – name of the project that the task belongs to *@param* **t** — task to be sent with the project

Extends	TimerTask
Implements	Runnable
Methods	public String **run**() Sends the report on a task beginning as an ASN1 project that will be received by the client and responded to.

Class	**TCPDecoder.java**
Description	Library that interprets input to the server from byte array sources.
Constructor	public **serverQuery**(String dbLocation, SimpleDateFormat sdf, Decoder dec, String ipAddress, int port) 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 *@param* **sdf** — format to save dates in *@param* **dec** — byte array to be decoded already in wrapper *@param* **ipAddress** — address of client that sent query *@param* **port** — number of the port of client that sent query *@throws* **SQLException** if creating/opening/modifying database fails
Methods	public **serverQuery**(String dbLocation, SimpleDateFormat sdf, Decoder dec, String ipAddress, int port) throws SQLException
	Takes a decoder from a known client and queries the database for information and updates. Responds to the query once finished querying the database with an encoded ASN1 string. *@param* **dbLocation** — proposed location of the database *@param* **sdf** — format to save dates in *@param* **dec** — byte array to be decoded already in wrapper *@param* **ipAddress** — address of client that sent query *@param* **port** — number of the port of client that sent query *@throws* **SQLException** if creating/opening/modifying database fails
	private static int **queryProject** (Connection conn, SimpleDateFormat _sdf, Project p) Adds the project to the database and updates as necessary

	@param **conn** – already opened connection to the database *@param* _sdf – format for all dates sent in (standard is UTC) *@param* p – decoded project to be entered into the database @return 0 for success and any other number for failure to update the database.
	private static int **queryTake** (Connection conn, Take take) Sets the owner of a project *@param* **conn** – already opened connection to the database *@param* **take –** decoded take to be entered into the database *@return* 0 for success and any other number for failure to update the database.
	private static int **queryGetProjects** (Connection conn) Returns the names of all projects currently in the database *@param* **conn** already opened connection to the database @return list of projects in the database
	private static int **queryGetProjectsUnabridged** (Connection conn, SimpleDateFormat sdf) throws ParseException Returns the names and body of all projects in the database. *@param* **conn** – already opened connection to the database @throws **ParseException** – if the dates retrieved from the database cannot be formatted into the desired date format @return list of projects in the database as an object
	private static int **queryGetProject** (Connection conn, SimpleDateFormat sdf, String project) throws ParseException Adds the project to the database and updates as necessary *@param* **conn** – already opened connection to the database *@param* **sdf** – format for all dates sent in (standard is UTC) *@param* **project** – name of the projec @return 0 for success and any other number for failure to update the database.

Class	**TCPThreadedServer.java**
	Listens for all client connections to the server and then starts a TCPHandler for

Description	each connection. Closes on the shutdown of the server or a failure to bind/use the socket
Constructor	public **TCPThreadedServer**(int port, String dbLocation) Checks whether proposed database location exists and if a database does not already exist with the given name then creates a database. *@param* **port** – port server listens on for TCP connections *@param* **dbLocation** — proposed location of the database
Implements	Runnable
Methods	public String **run**() Opens socket to port and creates threads for each client connection until the server is closed. Every connection created is independent once it is started.
	public **terminate**() Tells the connection handler to stop accepting new input and finish interpreting all current queries, then exit

Class	**TCPHandler.java**
Description	Handles an individual client's requests until the client ends the connection or the server is shut down
Constructor	public **TCPHandler**(Socket sock, String dbLocation) Checks whether proposed database location exists and if a database does not already exist with the given name then creates a database. *@param* **sock** – already opened connection to client *@param* **dbLocation** — proposed location of the database
Implements	Runnable
Methods	public String **run**() Accepts information from the client and responds to the client until the client closes the connection. Will close cleanly
	public **terminate**() Tells the TCP handler to stop accepting new input and finish interpreting all current queries, then exit

Class (Deprecated)	**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.