**Programming Language Practical Project**

**Phase 1**

Nouraldin Hassan

Contents

[Evidence of Learning 3](#_Toc179647756)

[Program Architecture 7](#_Toc179647757)

[Program Demonstration via Screen Shots 9](#_Toc179647758)

[Unit Testing Demonstration via Screen Shots 10](#_Toc179647759)

[Source Code Commenting Example 10](#_Toc179647760)

# Evidence of Learning

**Variables (and use of API Libraries, Arrays):**

program.js:  
2 const csvParser = require('csv-parser');  
3 const readline = require('readline');

5 const Record = require('../service/record');  
6 const { filePath, loadRecords } = require('../service/file-handler');

file-handler.js:  
9 const filePath = ./keystone-throughput-and-capacity.csv';  
10 const records = [];

“readline” is a required module that is used for reading data from a readable stream (process.stdin) and used in the case of making the CLI interactive to allow user input.  
all requirements are constants so the reference value cannot be changed nor reassigned.

records is an array used to store record objects. filePath is a string variable holding the relative path to the csv dataset.

**Snippet: records.forEach (Loop Structure):**

program.js:  
126 records.forEach(record => {  
127 console.log(`${record.date} | ${record.month} | ${record.year} | ${record.company} | ${record.pipeline} | ${record.keyPoint} | ${record.latitude} | ${record.longitude} | ${record.flowDirection} | ${record.tradeType} | ${record.product} | ${record.throughput} | ${record.committedVolumes} | ${record.uncommittedVolumes} | ${record.nameplateCapacity} | ${record.availableCapacity} | ${record.varianceReason}`);  
128 if ((index + 1) % 10 === 0) {  
129 console.log('\nProgram by: Nouraldin Hassan\n');  
130 }  
131 });

The for-each loop structure provided iterates over the records repeatedly until the cutoff limit (see program.js line 112 provided in project file alongside this document) -- where in that snippet, a table-like format would be presented. The console prints out the author every 10 records processed.

**Snippet: rl.question in promptUser() (Decision Structure):**

17 rl.question('Choose an option (note that you may need to repeat an option if nothing happens):\n1. Reload Data\n2. Save Data\n3. Display Records\n4. Add Record\n5. Edit Record\n6. Delete Record\n7. Exit\n\n', (answer) => {  
18 switch (answer) {  
19 case '1':  
20 reloadRecords(() => {  
21 clearConsole(true);  
22 rl.close();  
23 promptUser();  
24 });  
25 break;  
26 case '2':  
27 clearConsole(true);  
28 persistRecords('./program/data/output.csv');  
29 console.log('Data saved.');  
30 rl.close();  
31 promptUser();  
32 break;  
33 case '3':  
34 clearConsole(true);  
35 giveDisplayOptions(rl);  
36 //displayRecords();  
37 //rl.close();  
38 //promptUser();  
39 break;  
40 case '4':  
41 addNewRecord(rl);  
42 break;  
43 case '5':  
44 editRecord(rl);  
45 break;  
46 case '6':  
47 deleteRecordPrompt(rl);  
48 break;  
49 case '7':  
50 rl.close();  
51 //console.log('Program by Nouraldin Hassan');  
52 process.exit();  
53 break;  
54 default:  
55 console.log('Invalid option.');  
56 rl.close();  
57 promptUser();  
58 break;  
59 }  
60 });

The switch-case decision structure has been chosen to handle user input for programming efficiency and readability. In the given case, the choices provided are for corresponsing options to refresh, save, read, create, update, and delete records, as well as exiting the program. Each method goes to a given function (see the program.js file). In display options for instance, the user can choose to display a single record or multiple records, followed by entering an index or another option for multiple records display respectively. For multiple records options, choosing the table-like format option allows for choosing a cutoff limit so only 5 records for example will be displayed.

**Snippet: fs.createReadStream (Exception handling):**

139 fs.createReadStream(filePath)  
140 .pipe(csvParser())  
141 .on('data', (row) => {  
142 try {  
143 console.log('Parsed Row:', row);  
144 } catch (error) {  
145  console.error('Error parsing row:', error);  
146  }  
147 })

The given try-catch code handles row parsing exceptions that may arise, catching any errors found and displaying a message that tells that a row parse has come to an error.

**Snippet: fs.createReadStream (File-IO):**

46 function saveRecords(records, outputFilePath) {  
47 const headers = 'Date,Month,Year,Company,Pipeline,Key Point,Latitude,Longitude,Direction Of Flow,Trade Type,Product,Throughput (1000 m3/d),Committed Volumes (1000 m3/d),Uncommitted Volumes (1000 m3/d),Nameplate Capacity (1000 m3/d),Available Capacity (1000 m3/d),Reason For Variance\n';  
48 const csvData = records.map(record => {  
49         return `${record.date},${record.month},${record.year},${record.company},${record.pipeline},${record.keyPoint},${record.latitude},${record.longitude},${record.flowDirection},${record.tradeType},${record.product},${record.throughput},${record.committedVolumes},${record.uncommittedVolumes},${record.nameplateCapacity},${record.availableCapacity},${record.varianceReason}`;  
50 }).join('\n');  
51   
52 fs.writeFileSync(outputFilePath, headers + csvData, 'utf8');  
53 }

The given snippet performs reading upon the csv file path parameter. Record objects get mapped into the records array during the procedure and joined together. Once the data or file is detected to be complete or reached the end, the output gets written via fs.writeFileSync() and a message appears in the associated program.js case where the function is used that tells the user that the data has been saved.

**Methods:**

record-service.js:  
34 function getRecords() {  
35 return records;  
36 }

getRecords is a function that returns the records from the record array (that would contain records, where the records info was gathered from the CSV file via File-IO) (see previous section above).

file-handler.js:  
17 function loadRecords(callback, limit = 100) {  
18 fs.createReadStream(filePath)  
19 .pipe(csvParser())  
20 .on('data', (row) => {  
21 try {  
22 //console.log(row.Date);  
23 /\*\*  
24 \* Initialize a new Record object with data from the CSV file row.  
25 \* @type {Record}  
26 \*/  
27 const record = new Record(row.Date, row.Month, row.Year, row.Company, row.Pipeline, row['Key Point'], row.Latitude, row.Longitude, row['Direction of Flow'], row['Trade Type'], row.Product, row['Throughput (1000 m3/d)'], row['Committed Volumes (1000 m3/d)'], row['Uncommitted Volumes (1000 m3/d)'], row['Nameplate Capacity (1000 m3/d)'], row['Available Capacity (1000 m3/d)'], row['Reason For Variance']);  
28 records.push(record);  
29 } catch (error) {  
30 console.error('Error parsing row:', error);  
31 }  
32 })  
33 .on('end', () => {  
34 callback(records.slice(0, Math.max(2, Math.min(limit, 1000)))); // Limit to between 2 and 1000 records  
35 })  
36 .on('error', (error) => {  
37 console.error('Error reading file:', error);  
38 });  
39 }

the loadRecords() method loads from the csv file the records to push onto the records array, and utilizes the callback parameter to perform a function to limit records, in which the limit is also defined from the function’s second parameter. .on(‘error’) does the technical error handling like with try-catch.

**Unit Testing:**

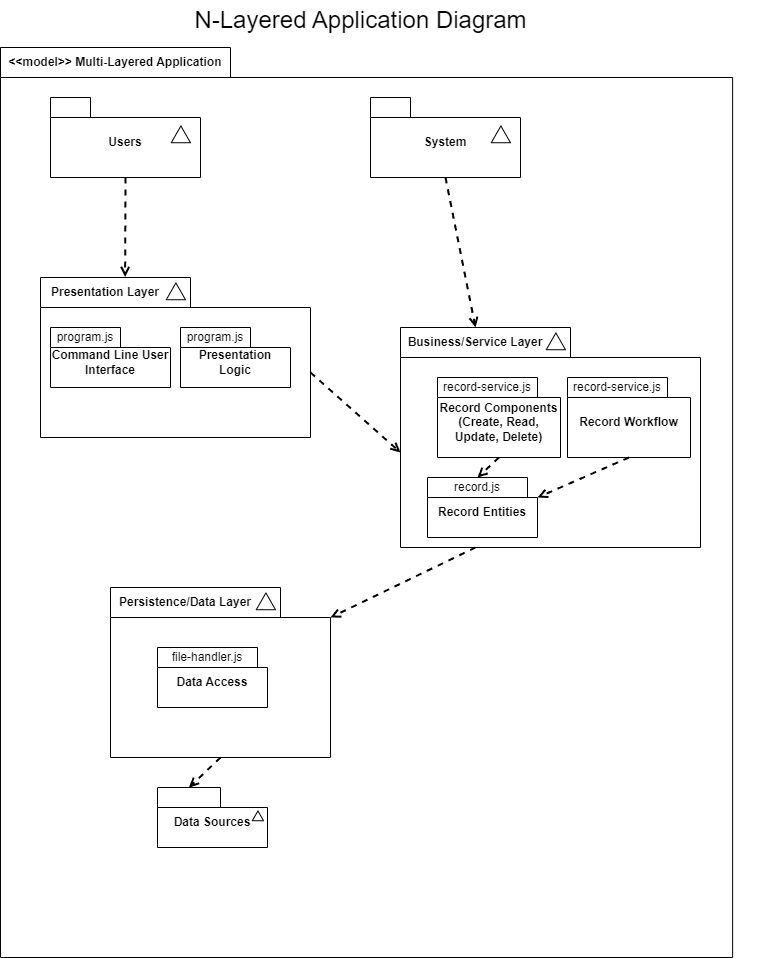
record-tests.js:  
7 initializeRecords(() => {  
8 const records = getRecords();  
9 const cutoffLimit = 22;  
10 const limitedRecords = records.slice(0, cutoffLimit);  
11   
12 assert(records.length > 0, 'Records should be loaded');  
13 assert.strictEqual(limitedRecords.length, cutoffLimit, `Expected ${cutoffLimit} records to be displayed.`);  
14   
15 for (let i = 0; i < limitedRecords.length; i++) {  
16 console.log(limitedRecords[i]);  
17 if ((i + 1) % 10 === 0) {  
18 console.log('\nProgram by Nouraldin Hassan\n');   
19 }  
20 }  
21   
22 console.log('All tests performed!');  
23 });

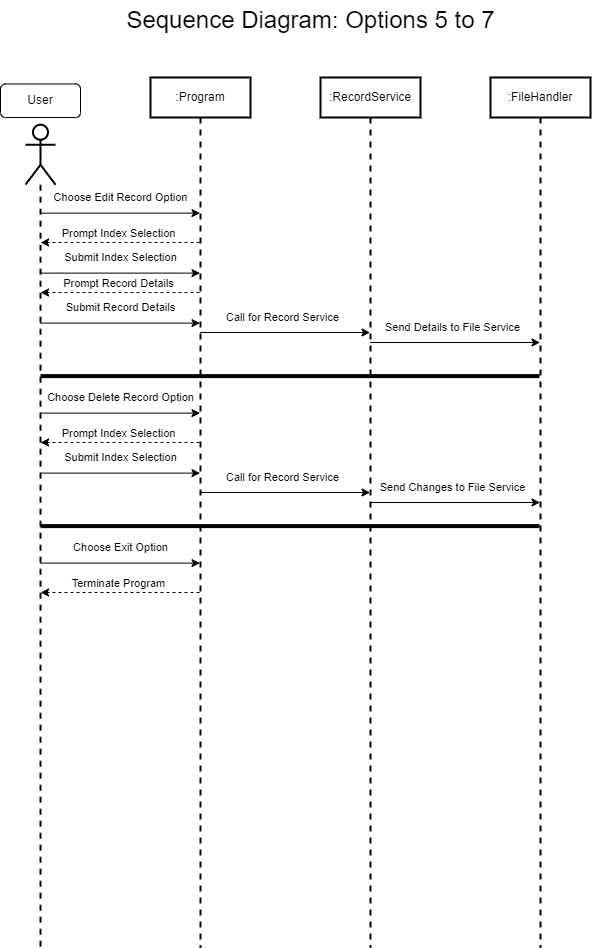
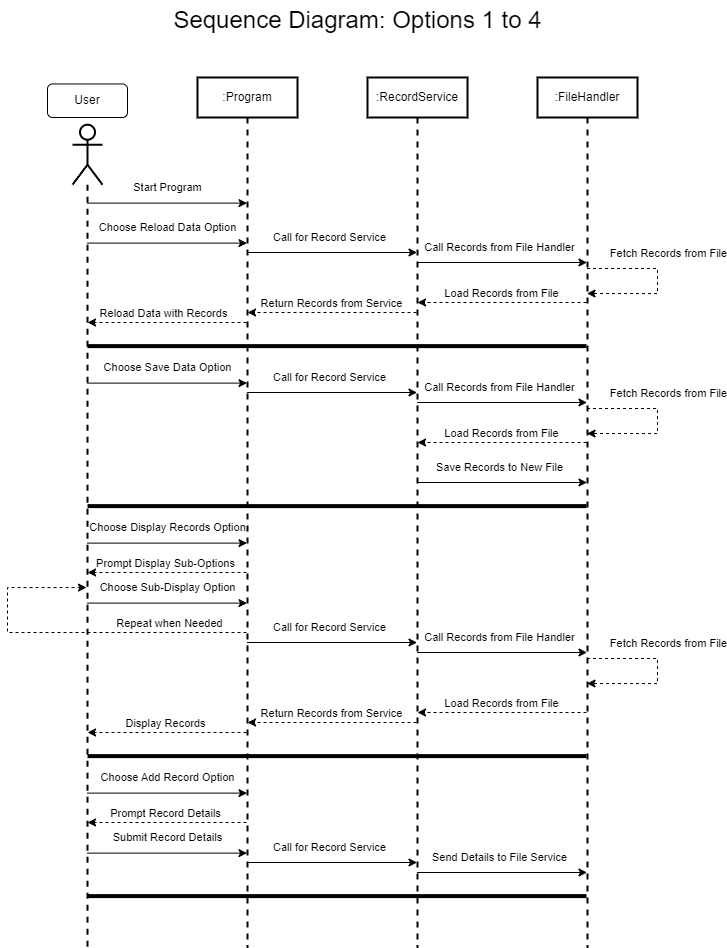
initializeRecords() is part of the unit testing procedure where a given number of records are displayed for checking cutoff limits and printing the author every 10 records.

# Program Architecture

I have divided the code files and associated code into four layers (three, if excluding test layer) to follow the N-Layered Architecture:  
Presentation Layer: Responsible for user interaction, menu display and input handling.  
Business/Service Layer: Manages in-memory data, including record operations like creation, update, and deletion.  
Persistence/Data Layer: Handles file input and output, including reading and writing to CSV files.  
Testing Layer (can be excluded from consideration): Performs unit testing procedures to confirm that given functions work as intended.  
  
Testing Layer was included because the assignment had called for the use of Unit Testing, and I needed to make a new folder alongside the three main layers for easy directory accessibility and navigation.

**UML Diagram:**

Attached is a UML Diagram that follows the Multi-Layered Architecture example.  


Attached below are two parts of Sequence Diagrams, with each section pertaining to an option.  


# Program Demonstration via Screen Shots

**Execution:**

A screenshot of a computer

Description automatically generated  
Pressing 1 reloads the data and the console texts remain the same. Reloaded data means the data is set back to default (i.e. Add/Edit/Delete functions are undone).

Pressing 2 saves the data to program/data where output.csv is created and data is from keystone-throughput-and-capacity.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated  
Pressing 3 provides the user with another prompt to choose a display option for the records to present.

A screenshot of a computer

Description automatically generated  
Pressing 1 from there brings the user to prompt an input of a record index to display. The number to input for the index represents a row to select.

A screenshot of a phone

Description automatically generated  
The row that corresponds to the index will be displayed, and the user will go back to the main option selection prompt.

A screenshot of a computer

Description automatically generated  
Repeating the steps to display a single record with an index of 99 yields the following result:

A screenshot of a computer

Description automatically generated  
Pressing 3 followed by 2 will show the options for selecting multiple records.

A screen shot of a computer

Description automatically generated  
Pressing 1 displays yet another prompt and entering a number between 2 and 1000 shows the amount of records based on the chosen number.

A screen shot of a computer

Description automatically generated  
Since 3 was chosen, three rows of records appeared. The reason why 2 is the minimum limit is because choosing a single record is already a thing in the previous prompt, and was done to show the learning of using readline.

A screenshot of a computer program

Description automatically generated  
Repeating 3, choosing 3, and then entering 22 shows the following rows:  
A screen shot of a computer

Description automatically generatedRepeating 3 and choosing 2:

A screen shot of a computer

Description automatically generated  
Unlike the first option however, the second option does not allow record limit inputs due to the way the processing logic works. The array-like format was created to show more variety of record processing.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated  
Pressing 4 shows a prompt to enter record details to add to the record data. After pressing enter, a new record added message appears.

A black screen with white text

Description automatically generated  
Pressing 5 edits a record, prompting for an index to select followed by entering new record details.

A black screen with white text

Description automatically generated A black screen with white text

Description automatically generated  
Repeating the process with some different fields changes the index at the given point and therefore the results, which will be seen later.

Pressing 6 prompts for an index to delete data from row corresponding to said index.

A screen shot of a computer

Description automatically generated  
Pressing 7 exits the program with a code of 0, meaning successful run.  
  
Checking the output (output.csv):  
A screenshot of a computer

Description automatically generated  
Row 99 has changed and row 102 now has data. Row 103 was added by accident. I typed 97; actual indexes start at 0 and the header does not count, so row 2 is index 0  
(Note that I had to redo the commands since I let the output open when doing so, forcing me to redo the entries of adding and editing).

# Unit Testing Demonstration via Screen Shots

**Programming:**

**A screen shot of a computer program

Description automatically generated**

**Execution:**

A screenshot of a computer program

Description automatically generated A screenshot of a computer program

Description automatically generated  
A computer screen shot of a program

Description automatically generated

# Source Code Commenting Example

// record-service.js  
/\*\*  
 \* @author {Nouraldin Hassan}  
\*/  
const { loadRecords, saveRecords } = require('../data/file-handler');  
const Record = require('./record');  
/\*\*  
 \* Initializes an empty array for storing records.  
 \*/  
let records = [];  
/\*\*  
 \* Initializes records by storing them in the records array.  
 \* @param {function} callback - The callback function to execute for record loading.  
 \*/  
function initializeRecords(callback) {  
    loadRecords((loadedRecords) => {  
        records = loadedRecords;  
        callback();  
    });  
}  
/\*\*  
 \* Reloads records by performing an initialization callback.  
 \* @param {function} callback - The callback function to execute after record loading.  
 \*/  
function reloadRecords(callback) {  
    initializeRecords(callback);  
}  
/\*\*  
 \* Provides record persistence though performing a record save.  
 \* @param {String} outputFilePath - The file path to output the records to.  
 \*/  
function persistRecords(outputFilePath) {  
    saveRecords(records, outputFilePath);  
}  
/\*\*  
 \* Fetches records  
 \*/  
function getRecords() {  
    return records;  
}  
/\*\*  
 \* Fetches records by index  
 \* @param {String} index - The index of the record.  
 \*/  
function getRecord(index) {  
    return records[index];  
}  
/\*\*  
 \* Adds a new record.  
 \* @param {any} record - The record to add.  
 \*/  
function addRecord(record) {  
    records.push(record);  
}  
/\*\*  
 \* Updates a record.  
 \* @param {any} index - The index of the record.  
 \* @param {any} updatedRecord - The updated record.  
 \*/  
function updateRecord(index, updatedRecord) {  
    records[index] = updatedRecord;  
}  
/\*\*  
 \* Deletes a record.  
 \* @param {any} index - The index of the record to remove.  
 \*/  
function deleteRecord(index) {  
    records.splice(index, 1);  
}  
  
module.exports = { initializeRecords, reloadRecords, persistRecords, getRecords, getRecord, addRecord, updateRecord, deleteRecord, records };  
  
//record-tests.js:  
  
/\*\*

 \* @author {Nouraldin Hassan}

 \*/

const assert = require('assert');

const { initializeRecords, getRecords } = require('../service/record-service');

initializeRecords(() => {

    const records = getRecords();

    const cutoffLimit = 22;

    const limitedRecords = records.slice(0, cutoffLimit);

    assert(records.length > 0, 'Records should be loaded');

    assert.strictEqual(limitedRecords.length, cutoffLimit, `Expected ${cutoffLimit} records to be displayed.`);

    for (let i = 0; i < limitedRecords.length; i++) {

        console.log(limitedRecords[i]);

        if ((i + 1) % 10 === 0) {

            console.log('\nProgram by Nouraldin Hassan\n');

        }

    }

    console.log('All tests performed!');

});