**Programming Language Practical Project**

**Phase 3**

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) [1][2]:**

csv-db-handler.js:

const mysql = require('mysql2/promise');  
const createCsvWriter = require('csv-writer').createObjectCsvWriter;

CSV Writer is a required module that is used for writing data – in this case, from the database table, to create a new CSV file.

The promise part in mysql2 deals with the connection in an asynchronous manner.

const dbConfig = {  
    host: '127.0.0.1',  
    user: 'root',  
    password: rootpassword',  
    database: 'nodejs\_instance',  
};  
  
dbConfig is needed for creating a connection to the database itself. dbConfig has also been used in csv-import-exec.js, which is a script intended to be run directly to add the csv file to the database.

There is also the same implementation used in dbconfig.js itself but for the creation of a connection pool to have utilized for record-service.js.

**Snippet: async function getRecords() (promise pool query execution)[2]:**

record-service.js:  
42 async function getRecords() {  
43 try {  
44 const [rows, fields] = await promisePool.execute('SELECT \* FROM records');  
45 return rows;  
46 } catch (error) {  
47 console.error('Error fetching records:', error);  
48 throw error;  
49 }  
50 }

The promise pool provided executes the select all from table query within the constant array variable of rows, in which it gets returned inside a try-catch loop, so any errors get handled (see dbconfig.js line 13 provided alongside this document for the module export of the promise pool).

**Snippet: CSV importing (Database File Processing Structure and Exception Handling):**

22 try {  
23         connection = await mysql.createConnection(dbConfig);  
24         console.log('Connected to the database.');  
25         const records = [];  
26         console.log(`Reading CSV file: ${filePath}`);  
27         await new Promise((resolve, reject) => {  
28             fs.createReadStream(filePath)  
29                 .pipe(csv({  
30                     headers: [  
31                         'date', 'month', 'year', 'company', 'pipeline', 'keypoint', 'latitude', 'longitude',  
32                         'directionofflow', 'tradetype', 'product', 'throughput1000m3d', 'committedvolumes1000m3d',  
33                         'uncommittedvolumes1000m3d', 'nameplatecapacity1000m3d',  
'availablecapacity1000m3d',  
34                         'reasonforvariance'  
35                     ],  
36                     skipLines: 1 // Skip the header row if it's present  
37                 }))  
38                 .on('data', (row) => {  
39                     //console.log('Row data:', row);  
40                     const record = [  
41                         formatDate(row.date),  
42                         parseInt(row.month),  
43                         parseInt(row.year),  
44                         row.company,  
45                         row.pipeline,  
46                         row.keypoint,  
47                         parseFloat(row.latitude),  
48                         parseFloat(row.longitude),  
49                         row.directionofflow,  
50                         row.tradetype,  
51                         row.product,  
52                         parseFloat(row.throughput1000m3d || 0),  
53                         parseFloat(row.committedvolumes1000m3d || null),  
54                         parseFloat(row.uncommittedvolumes1000m3d || null),  
55                         parseFloat(row.nameplatecapacity1000m3d || null),  
56                         parseFloat(row.availablecapacity1000m3d || null),  
57                         row.reasonforvariance,  
58                     ];  
59                     if (record[0]) {  
60                         records.push(record);  
61                     } else {  
62                         console.warn('Skipping invalid row:', row);  
63                     }  
64                 })  
65                 .on('end', resolve)  
66                 .on('error', reject);  
67         });  
68         console.log(`Finished reading CSV file. ${records.length} records found.`);  
69         const truncateQuery = `TRUNCATE TABLE records`;  
70         // console.log('Truncating the records table...');  
71         await connection.execute(truncateQuery);  
72         if (records.length > 0) {  
73             const query = `  
74                 INSERT INTO records (  
75                     record\_date, record\_month, record\_year, company, pipeline, key\_point, latitude, longitude,   
76                     direction\_of\_flow, trade\_type, product, throughput, committed\_volumes,   
77                     uncommitted\_volumes, nameplate\_capacity, available\_capacity, reason\_for\_variance  
78                 ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)`;  
79             await connection.beginTransaction();  
80             for (const record of records) {  
81                 //console.log(`Inserting record: ${JSON.stringify(record)}`);  
82                 await connection.execute(query, record);  
83             }  
84             await connection.commit();  
85             console.log('All records have been successfully imported into the database.');  
86         } else {  
87             console.log('No records to import.');  
88         }  
89     }

The database file decision structure has been chosen to handle csv file input for programming efficiency and simplicity without the need to open MySQL Workbench and go through the table data importing process. In the given case, the headers are shown but the header row is skipped so the headers don’t cause the importing process to encounter an error and not go through with the import… The record array contains row information and are parsed as needed to be properly imported onto the database table. The next step that happens is truncating the table so the import process can actually happen, then it follows by having a query made to insert into the table the new values, and a database transaction starts. A for-loop iterates over all known rows and executes the query each time to make sure all CSV rows are sent to the database. After that, a commit happens to the database, completing the import procedure.

The given try-catch code included handles any exception that may arise, catching any errors found and displaying a message that tells that the CSV could not be imported for any reason.

**Snippet: async function viewRecordById (Database-IO):**

140 async function viewRecordById(rl) {  
141 rl.question('Enter record ID: ', async (id) => {  
142 try {  
143 const record = await getRecord(id);  
144 if (record) {  
145 console.log('Record details:');  
146 console.log(record);  
147 } else {  
148 console.log('Record not found.');  
149 }  
150 } catch (error) {  
151 console.error('Error fetching record:', error);  
152 }  
153 rl.close()  
154 promptUser();  
155 });  
156 }

The given snippet performs reading upon the data table, in which the record variable awaits the getRecord function. From there, the promise pool can execute the select all from table query. Record objects get selected from the table during the procedure and displayed to the console.

**Methods:**

program.js:  
183 async function addNewRecord(rl) {  
184 rl.question('Enter the record details in the following format (date, month, year, company, pipeline, keyPoint, latitude, longitude, flowDirection, tradeType, product, throughput, committedVolumes, uncommittedVolumes, nameplateCapacity, availableCapacity, varianceReason):\n', async (input) => {  
185 const details = input.split(',').map(item => item.trim());  
186 if (details.length !== 17) {  
187 console.log('Invalid input. Please provide all 17 fields.');  
188 rl.close()  
 189 promptUser();  
190 return;  
191 }  
192 try {  
193 const newRecord = new Record(...details);  
194 await addRecord(newRecord);  
195 console.log('Record added successfully!');  
196 } catch (error) {  
197 console.error('Error adding record:', error);  
198 }  
199 rl.close()  
200 promptUser();  
201 });  
202 }

addNewRecord is a function that allows the user to enter the row details to then add to the database, modified to allow for the date to be typed in either DD/MM/YYYY format or YYYY-MM-DD format.

Other methods were shown prior (see the previous sections above).

**Unit Testing:**

record-tests.js: (there are way too many code rows; I won’t have enough time for the rest of the assignment if line numbers are done)  
const assert = require('assert');  
const fs = require('fs');  
const mysql = require('mysql2/promise');  
const { Readable } = require('stream');  
const importCSV = require('../data/csv-import');  
let mockConnection = null;  
// Simulate a mock connection (in-memory simulation)  
function mockSetup() {  
 mockConnection = {  
 execute: function (...args) {  
 this.\_calls = this.\_calls || [];  
 this.\_calls.push(args);  
 return Promise.resolve();  
 },  
 beginTransaction: function () {  
 return Promise.resolve(); // Simulating beginning of transaction  
 },  
 commit: function () {  
 return Promise.resolve(); // Simulating commit  
 },  
 rollback: function () {  
 return Promise.resolve(); // Simulating rollback  
 },  
 end: function () {  
 return Promise.resolve(); // Simulating closing the connection  
 },  
 \_calls: [] // Store the calls to `execute`  
 };  
 // Mock the MySQL connection  
 mysql.createConnection = function () {  
 return Promise.resolve(mockConnection);  
 };  
 // Mock fs.createReadStream to simulate reading CSV data  
 fs.createReadStream = function () {  
 const records = [  
 ['2024-04-01', 4, 2024, 'TransCanada Keystone Pipeline GP Ltd.', 'Keystone pipeline', 'International boundary at or near Haskett, Manitoba', 48.9989, -97.9577, 'south', 'export', 'domestic light', 2.34, null, null, null, 98.6, 'Capacity may vary month to month based on CER Regulatory Directive, Downstream Restrictions, Curtailment/Interruptions, Force Majeure and System Operating Factor'],  
 ['2024-05-01', 5, 2024, 'TransCanada Keystone Pipeline GP Ltd.', 'Keystone pipeline', 'International boundary at or near Haskett, Manitoba', 48.9989, -97.9577, '', '', '', 93.64, 1.24, null, null, null, '']  
 ];  
 // Use Readable stream to simulate CSV input  
 const readableStream = new Readable({  
 read() {  
 records.forEach(record => {  
 this.push(Buffer.from(record.join(','), 'utf-8') + '\n'); // Simulate line-by-line pushing to the stream  
 });  
 this.push(null); // End of stream  
 }  
 });  
 return readableStream;  
 };  
}  
// Test Case 1: Should truncate the table and insert records  
async function testTruncateAndInsert() {  
 mockSetup();  
 const fakeCSVFilePath = '../data/output.csv';  
 // Simulate importCSV reading and inserting data into the database  
 await importCSV(fakeCSVFilePath)  
 // Assert that the table was truncated  
 assert.strictEqual(mockConnection.\_calls[0][0], 'TRUNCATE TABLE records');  
 // Assert that insertions only happen when records are valid  
 if (mockConnection.\_calls.length > 1) {   
 // If we have insertions, assert the values  
 assert.strictEqual(mockConnection.\_calls.length, 3); // 1 for TRUNCATE, 2 for INSERT  
 // Check the values inserted into the database  
 assert.deepStrictEqual(mockConnection.\_calls[1][0], 'INSERT INTO records');  
 assert.deepStrictEqual(mockConnection.\_calls[1][1], [  
 '2024-04-01', 4, 2024, 'TransCanada Keystone Pipeline GP Ltd.', 'Keystone pipeline', 'International boundary at or near Haskett, Manitoba', 48.9989, -97.9577, 'south', 'export', 'domestic light', 2.34, null, null, null, 98.6, 'Capacity may vary month to month based on CER Regulatory Directive, Downstream Restrictions, Curtailment/Interruptions, Force Majeure and System Operating Factor'  
 ]);  
 assert.deepStrictEqual(mockConnection.\_calls[2][0], 'INSERT INTO records');  
 assert.deepStrictEqual(mockConnection.\_calls[2][1], [  
 '2024-05-01', 5, 2024, 'TransCanada Keystone Pipeline GP Ltd.', 'Keystone pipeline', 'International boundary at or near Haskett, Manitoba', 48.9989, -97.9577, '', '', '', 93.64, 1.24, null, null, null, ''  
 ]);  
 } else {  
 // If no insertions were made, assert that no INSERT statements were executed  
 console.log('No valid records were inserted.');  
 }  
 // Ensure commit was called once if records were inserted  
 if (mockConnection.\_calls.length > 1) {  
 assert.strictEqual(mockConnection.commit.mock ? mockConnection.commit.mock.calls.length : 1, 1); // Only 1 commit  
 }  
 // Ensure connection was closed  
 assert.strictEqual(mockConnection.end.mock ? mockConnection.end.mock.calls.length : 1, 1); // Only 1 end  
}  
// Test Case 2: Should handle errors during CSV import  
async function testHandleErrorDuringImport() {  
 mockSetup();  
 // Simulate a database connection error by mocking createConnection to reject  
 mysql.createConnection = function () {  
 return Promise.reject(new Error('Database connection error'));  
 };  
 const fakeCSVFilePath = '../data/output.csv';  
 try {  
 await importCSV(fakeCSVFilePath);  
 } catch (error) {  
 // Assert that the correct error message was thrown  
 assert.strictEqual(error.message, 'Database connection error');  
 }  
 // Ensure that the rollback method was called  
 assert.strictEqual(mockConnection.rollback.mock ? mockConnection.rollback.mock.calls.length : 1,  
1);  
}  
// Test Case 3: Should handle empty CSV file  
async function testHandleEmptyCSVFile() {  
 mockSetup();  
 const fakeCSVFilePath = '../data/blank.csv';  
 // Simulate an empty CSV to simulate no records  
 fs.createReadStream = function () {  
 const emptyStream = new Readable({  
 read() {  
 this.push(null); // Empty stream, no data  
 }  
 });  
 return emptyStream;  
 };  
 await importCSV(fakeCSVFilePath);  
 // Assert that only the TRUNCATE statement was executed (no insertions)  
 assert.strictEqual(mockConnection.\_calls.length, 1); // Only the TRUNCATE query should be called  
 assert.strictEqual(mockConnection.commit.mock ? mockConnection.commit.mock.calls.length : 0, 0); // No commits since no records were inserted  
}

// Run all tests  
async function runTests() {  
 try {  
 await testTruncateAndInsert();  
 await testHandleErrorDuringImport();  
 await testHandleEmptyCSVFile();  
 console.log('All tests passed!');  
 } catch (error) {  
 console.error('Some tests failed:', error);  
 }  
}  
runTests();  
  
There are three tests, in which all have been separated in different files so NodeJS can recognise them as different tests.  
Database connectivity is simulated, as well as CSV data reading. The first test intends to test table truncation and insertion. The second test intends to handle CSV import errors. The third test intends to handle an empty CSV file.  
  
[1] G. Bulmash, "Build an API with Postman, Node.js, and MySQL," *Postman*, 6 June 2023. [Online]. Available: https://blog.postman.com/build-an-api-with-postman-node-js-and-mysql/. [Accessed 3 November 2024]

[2] M. Kazanski, "Using mysql2 npm package on Google Cloud SQL," StackOverflow, 23 August 2018. [Online]. Available: https://stackoverflow.com/questions/51955252/using-mysql2-npm-package-on-google-cloud-sql. [Accessed 3 November 2024].

# Program Changes

I have attempted to keep the division of 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-database data, including record operations like creation, update, and deletion.  
Persistence/Data Layer: Handles data and file input and output, including reading and writing to CSV files from or to the database.  
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.  
  
The framework chosen helps keep the project organized on the entire program. I was able to keep some things remaining, but others required me to create a new script file or add content to the same script file since for some reason, a couple of the scripts didn’t want to cooperate with the new changes I had made, even though things should have logically worked.  
There were even some things, such as the record limit, that I wanted to use but I didn’t have a chance to refactor that procedure for use with the database, so I plan to do that in Phase 4.  
  
While single-threading was the only option to use, tasks were still able to be done asynchronously in order to get the CSV file loaded at the point in program execution. The user interface was partially affected in terms of speed, but the intended procedure of the UI remained the same.

# 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).

A screenshot of a computer

Description automatically generated

Pressing 2 saves the data to program/data where output-from-database.csv is created and data is from keystone-throughput-and-capacity.  
  
A screenshot of a computer

Description automatically generated

The database table known as “records” has been truncated and re-populated, so the row id is set back to 1 (assignment instructions said not to show the database).  
  
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 ID to display. The number to input for the ID represents a row to select.

A black screen with white text

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

A screenshot of a computer program

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

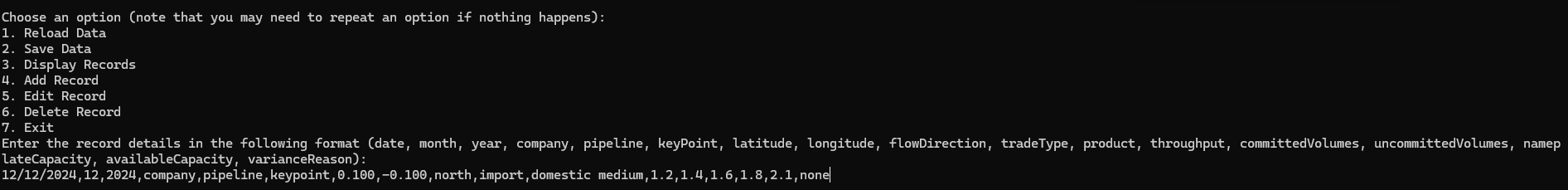
A screenshot of a computer

Description automatically generated   
Pressing 3 followed by 2 will show all records in the database (note that date and some other fields do not show since they would appear as undefined).

A screenshot of a computer program

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 screen shot of a computer

Description automatically generated  
A screenshot of a computer program

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

A screenshot of a computer

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

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

A computer screen shot of a black screen

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

Description automatically generated  
Row 1 has changed and row 354 and 355 now has data. actual IDs start at 1 and the header does not count, so row 2 is index 3, id 2.

# Unit Testing Demonstration via Screen Shots

**Programming:**

**A screen shot of a computer program

Description automatically generated  
Note, line 5 changed to say “const { importCSV } = require('../data/csv-db-handler');”**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Execution:**

**The tests run on different .js files so NodeJS recognises them as different tests.**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

Note: that is intentional

A screenshot of a computer program

Description automatically generated

**All Tests Combined:**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**There should have been some test failures due to the exceptions and intentionally-coded throws.**

# Source Code Commenting Example

I have updated the source code comments prior to adding them here.

// record-service.js  
/\*\*

\* @author {Nouraldin Hassan}

\*/

const { loadRecords, saveRecords } = require('../data/file-handler');

const Record = require('./record');

//const { query } = require('../database/dbconfig');

const promisePool = require('../data/dbconfig');

const mysql = require('mysql2');

/\*\*

\* 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.

\* @deprecated reloadDatabaseFromCSV() from csv-db-handler.js is to be used instead.

\*/

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

\*/

async function getRecords() {

try {

const [rows, fields] = await promisePool.execute('SELECT \* FROM records');

return rows;

} catch (error) {

console.error('Error fetching records:', error);

throw error;

}

}

/\*\*

\* Fetches records by index

\* @param {String} index - The index of the record.

\*/

async function getRecord(index) {

try {

const [rows, fields] = await promisePool.execute('SELECT \* FROM records WHERE id = ?', [index]);

return rows[0];

} catch (error) {

console.error('Error fetching record:', error);

throw error;

}

}

/\*\*

\* Adds a new record.

\* @param {any} record - The record to add.

\*/

async function addRecord(record) {

try {

const { date, month, year, company, pipeline, keyPoint, latitude, longitude,

flowDirection, tradeType, product, throughput, committedVolumes,

uncommittedVolumes, nameplateCapacity, availableCapacity, varianceReason } = record;

const isDateFormatted = /^\d{4}-\d{2}-\d{2}$/.test(date);

const formattedDate = isDateFormatted ? date : (() => {

const dateParts = date.split('/');

return `${dateParts[2]}-${dateParts[0].padStart(2, '0')}-${dateParts[1].padStart(2, '0')}`; // Format to YYYY-MM-DD

})();

// Resolving empty string issues for decimal fields needing a value

const safeCommittedVolumes = committedVolumes === "" ? null : parseFloat(committedVolumes);

const safeUncommittedVolumes = uncommittedVolumes === "" ? null : parseFloat(uncommittedVolumes);

const safeNameplateCapacity = nameplateCapacity === "" ? null : parseFloat(nameplateCapacity);

const safeAvailableCapacity = availableCapacity === "" ? null : parseFloat(availableCapacity);

const result = await promisePool.execute(

'INSERT INTO records (record\_date, record\_month, record\_year, company, pipeline, key\_point, latitude, longitude, direction\_of\_flow, trade\_type, product, throughput, committed\_volumes, uncommitted\_volumes, nameplate\_capacity, available\_capacity, reason\_for\_variance) ' +

'VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)',

[formattedDate, month, year, company, pipeline, keyPoint, latitude, longitude, flowDirection,

tradeType, product, throughput, safeCommittedVolumes, safeUncommittedVolumes,

safeNameplateCapacity, safeAvailableCapacity, varianceReason]

);

console.log('New record added:', result);

} catch (error) {

console.error('Error adding record:', error);

throw error;

}

}

/\*\*

\* Updates a record.

\* @param {any} index - The index of the record.

\* @param {any} updatedRecord - The updated record.

\*/

async function updateRecord(index, updatedRecord) {

try {

const { date, month, year, company, pipeline, keyPoint, latitude, longitude,

flowDirection, tradeType, product, throughput, committedVolumes,

uncommittedVolumes, nameplateCapacity, availableCapacity, varianceReason } = updatedRecord;

const result = await promisePool.execute(

'UPDATE records SET record\_date = ?, record\_month = ?, record\_year = ?, company = ?, pipeline = ?, key\_point = ?, ' +

'latitude = ?, longitude = ?, direction\_of\_flow = ?, trade\_type = ?, product = ?, throughput = ?, ' +

'committed\_volumes = ?, uncommitted\_volumes = ?, nameplate\_capacity = ?, available\_capacity = ?, ' +

'reason\_for\_variance = ? WHERE id = ?',

[date, month, year, company, pipeline, keyPoint, latitude, longitude, flowDirection,

tradeType, product, throughput, committedVolumes, uncommittedVolumes,

nameplateCapacity, availableCapacity, varianceReason, index]

);

console.log('Record updated:', result);

} catch (error) {

console.error('Error updating record:', error);

throw error;

}

}

/\*\*

\* Deletes a record.

\* @param {any} index - The index of the record to remove.

\*/

async function deleteRecord(index) {

try {

const result = await promisePool.execute('DELETE FROM records WHERE id = ?', [index]);

console.log('Record deleted:', result);

} catch (error) {

console.error('Error deleting record:', error);

throw error;

}

}

module.exports = { initializeRecords, reloadRecords, persistRecords, getRecords, getRecord, addRecord, updateRecord, deleteRecord, records };