CS1211 PROJECT PHASE 02

Fields	Details
Section	A
Group Members USN and Names	Amruthesh C Hiremath - 1RUA24CSE0042 Anant Nagaraj Hegde - 1RUA24CSE0045
Project Title	UNIVERSITY LOST AND FOUND SYSTEM (UNIFIND)
Date of Submission	25-04-2025

Course Instructor
(Signature with date)

Team Lead
(Signature with date)

Project Title:

University Lost and Found System (UNIFIND)

Description of technologies used:

The University Lost & Found Network project leverages a robust stack of technologies across front-end, back-end, and database layers to deliver a seamless platform for managing lost and found items on a university campus. Below is a detailed breakdown of the technologies employed:

Front-End Technologies

- **HTML5**: Provides the foundational structure for all web pages (home.html, report.html, search.html, about.html, main_about.html). It uses semantic tags and attributes like charset and viewport for accessibility and responsiveness.

- **CSS3:** Handles styling via styles.css, incorporating advanced features such as:
  - **CSS Variables**: Defined in :root for consistent theming (e.g., --primary: #3498db).
  - **Gradients**: Used for backgrounds (e.g., background: linear-gradient(135deg, #d2f8f0, #e6f0ff)).
  - **Media Queries**: Ensures responsive design (e.g., @media (max-width: 768px)).
  - **Dark Mode**: Implements a toggleable dark theme using the .dark-mode class.
  - **Transitions**: Adds smooth animations (e.g., transition: transform 0.3s ease-in-out).

- **JavaScript (ES6)**: Powers interactivity in script.js, utilizing:
  - **Event Listeners**: For DOM manipulation (e.g., document.addEventListener('DOMContentLoaded', ...)).
  - **Fetch API**: For asynchronous communication with the back-end (e.g., fetch('http://localhost:5000/api/lost-items')).
  - **Local Storage**: Manages user sessions (e.g., localStorage.setItem('theme', 'dark')).
  - **Dynamic Rendering**: Updates UI elements like search results and user authentication status.

- **Font Awesome**: Supplies icons for navigation and UI enhancement (e.g., <i class="fas fa-search"></i>), sourced via CDN (https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.0/css/all.min.css).

Back-End Technologies:

- **Node.js**: Serves as the runtime environment, enabling server-side JavaScript execution in server.js.
- **Express.js**: A lightweight framework for Node.js that simplifies:
  - **Routing**: Defines API endpoints (e.g., app.get('/api/lost-items', ...)).
  - **Middleware**: Manages request processing (e.g., app.use(cors())).
- **Multer**: Handles file uploads for item photos, configured with disk storage in server.js (e.g., const upload = multer({ storage })).
- **CORS**: Enables cross-origin resource sharing, allowing front-end requests to the back-end (e.g., app.use(cors())).

Database Technologies:

- **MySQL**: A relational database management system hosting the project's data, defined in Lost_and_Found.sql. It includes tables like User, Lost_Item, Found_Item, and Claim with appropriate relationships.
- **mysql2**: A Node.js module for MySQL interaction, supporting promises for asynchronous queries (e.g., mysql.createConnection({...})).

Other Tools:

- **Cloudflare**: Provides email protection via obfuscation scripts in HTML files and potentially CDN services.
- **Git**: Implied for version control, as evidenced by GitHub links in main_about.html.
- **Docker**: Mentioned in home.html under technologies but not explicitly implemented in the provided code.

This technology stack ensures a responsive, user-friendly interface, efficient server-side processing, and reliable data management.

Database connection implementation:

The database connection is implemented in server.js using the mysql2 library, establishing a link between the Node.js back-end and the MySQL database. Here's a detailed explanation:

- **Configuration**: The connection is initialized with a configuration object specifying:
  - host: 'localhost': Local MySQL server.
  - user: 'root': Default MySQL user.
  - password: 'Amrutheshhere': User-specific password (note: hardcoded for development; should be secured in production).
  - database: 'sem_project': Target database created in Lost_and_Found.sql.

Connection Code:

```
const db = mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: 'Amrutheshhere',
      database: 'sem_project'
    });
    db.connect((err) => {
      if (err) {
        console.error('❌ Database connection failed:', err.stack);
        return;
      }
      console.log('✅ Connected to MySQL database');
    });
```

- **Features**:
  - **Error Handling**: Checks for connection errors and logs them with stack traces.
  - **Promise Support**: Uses mysql2's promise API (e.g., db.promise().query()) for asynchronous operations, enhancing code readability and error management.
  - **Persistent Connection**: Established once at server startup, reused for all queries.

- **Schema**: Defined in Lost_and_Found.sql, it includes:
  - **Tables**: User, Lost_Item, Found_Item, Claim, Location, Category, Feedback, Attachment, Admin.
  - **Relationships**: Enforced via foreign keys (e.g., FOREIGN KEY (Reported_By) REFERENCES User(User_ID)).
  - **Constraints**: Uses ENUM for status fields and UNIQUE for email and category names.

This setup ensures reliable data access and integrity across the application.

Detailed explanation of each CRUD operation with a code snippet:

The project implements CRUD operations to manage users, lost items, and found items via API endpoints in server.js, with front-end interactions in script.js. Below is a comprehensive breakdown:

Create Operations

User Signup:

- **Purpose**: Registers a new user in the User table.
- **Endpoint**: POST /api/signup
- **Process**: Takes user details from the request body, inserts them into the database, and returns a success message with the new user's ID.
- **Front-End Integration**: Triggered by the signup form in home.html via signupUser() in script.js.

Code Snippet:

```
app.post('/api/signup', async (req, res) => {

  const { First_Name, Last_Name, Email, Phone, User_Type, Department,
Password } = req.body;

  const query = `INSERT INTO User (First_Name, Last_Name, Email,
Phone, User_Type, Department, Password) VALUES (?, ?, ?, ?, ?, ?, ?)`;

  const values = [First_Name, Last_Name, Email, Phone, User_Type,
Department, Password];

  try {

    const [result] = await db.promise().query(query, values);

    res.status(201).json({ message: 'Signup successful', userId:
result.insertId });

  } catch (err) {

    console.error('❌ Signup failed:', err);

    res.status(500).json({ error: 'Signup failed', details:
err.message });

  }

});
```

- **Details**:
  - **Input Validation**: Relies on front-end required fields; back-end assumes valid data (additional validation could be added).
  - **Security**: Password is stored plaintext (should be hashed in production using a library like bcrypt).
  - **Response**: Returns HTTP 201 with the new userId.

Report Lost Item

- **Purpose**: Adds a new lost item to the Lost_Item table.
- **Endpoint**: POST /api/lost-items

- **Process**: Accepts form data including a photo, resolves category and location IDs, and inserts the record.
- **Front-End Integration**: Submitted via lostReportForm in report.html using FormData in script.js.

Code Snippet:

```
app.post('/api/lost-items', upload.single('Photo_Path'), async (req, res) => {

  const { Reported_By, Category_ID: categoryName, Location_ID: locationName, Item_Name, Description, Lost_Date, Lost_Time, Color, Features, Status } = req.body;

  const Photo_Path = req.file ? req.file.path : null;

  try {

    const [categoryRows] = await db.promise().query('SELECT Category_ID FROM Category WHERE Category_Name = ?', [categoryName]);

    if (categoryRows.length === 0) return res.status(400).json({ error: 'Invalid category name' });

    const categoryId = categoryRows[0].Category_ID;

    const [locationRows] = await db.promise().query('SELECT Location_ID FROM Location WHERE Building_Name = ?', [locationName]);

    if (locationRows.length === 0) return res.status(400).json({ error: 'Invalid location name' });

    const locationId = locationRows[0].Location_ID;

    const query = `INSERT INTO Lost_Item (Reported_By, Category_ID, Location_ID, Item_Name, Description, Lost_Date, Lost_Time, Color, Features, Photo_Path, Status) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)`;

    const values = [Reported_By, categoryId, locationId, Item_Name, Description, Lost_Date, Lost_Time || null, Color, Features, Photo_Path, Status || 'Open'];

    const [result] = await db.promise().query(query, values);
    res.status(201).json({ message: 'Lost item reported successfully',
Lost_Item_ID: result.insertId });

  } catch (err) {

    console.error('Error inserting lost item:', err);

    res.status(500).json({ error: 'Failed to report lost item' });

  }

});
```

- **Details**:
    - **File Upload**: Uses Multer to store photos in ./uploads/.
    - **Validation**: Checks for valid category and location names, returning 400 if invalid.
    - **Default Status**: Sets to 'Open' if not provided.
    - **Asynchronous**: Uses async/await for sequential query execution.

Report Found Item

- **Purpose**: Adds a new found item to the Found_Item table.
- **Endpoint**: POST /api/found-items
- **Process**: Similar to lost item reporting, with fields adjusted for found items.
- **Front-End Integration**: Submitted via foundReportForm in report.html.

Code Snippet:

```
app.post('/api/found-items', upload.single('Photo_Path'), async (req,
res) => {

  const { Reported_By, Category_ID: categoryName, Location_ID:
locationName, Item_Name, Description, Found_Date, Found_Time, Color,
Features, Status } = req.body;

  const Photo_Path = req.file ? req.file.path : null;

  try {
    const [categoryRows] = await db.promise().query('SELECT
Category_ID FROM Category WHERE Category_Name = ?', [categoryName]);

    if (categoryRows.length === 0) return res.status(400).json({
error: 'Invalid category name' });

    const categoryId = categoryRows[0].Category_ID;

    const [locationRows] = await db.promise().query('SELECT
Location_ID FROM Location WHERE Building_Name = ?', [locationName]);

    if (locationRows.length === 0) return res.status(400).json({
error: 'Invalid location name' });

    const locationId = locationRows[0].Location_ID;

    const query = `INSERT INTO Found_Item (Reported_By, Category_ID,
Location_ID, Item_Name, Description, Found_Date, Found_Time, Color,
Features, Photo_Path, Status) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?)`;

    const values = [Reported_By, categoryId, locationId, Item_Name,
Description, Found_Date, Found_Time || null, Color, Features,
Photo_Path, Status || 'Unclaimed'];

    const [result] = await db.promise().query(query, values);

    res.status(201).json({ message: 'Found item reported
successfully', Found_Item_ID: result.insertId });

  } catch (err) {

    console.error('Error inserting found item:', err);

    res.status(500).json({ error: 'Failed to report found item' });

  }

});
```

- **Details**:
  - **Similarities**: Mirrors lost item logic with Found_Date and Found_Time.
  - **Default Status**: Sets to 'Unclaimed' if not specified.

Read Operations

Fetch Categories

- **Purpose**: Retrieves all categories for form dropdowns.
- **Endpoint**: GET /api/categories
- **Process**: Queries the Category table and returns all rows.
- **Front-End Integration**: Populates dropdowns in report.html via populateDropdowns() in script.js.

Code Snippet:

```
app.get('/api/categories', async (req, res) => {

  try {

    const [results] = await db.promise().query('SELECT * FROM
Category');

    res.json(results);

  } catch (err) {

    console.error('Error fetching categories:', err);

    res.status(500).json({ error: 'Failed to fetch categories' });

  }

});
```

- **Details**:
  - **Simplicity**: No filtering, returns all categories.
  - **Usage**: Used to dynamically populate <select> elements.

Search Lost Items:

- **Purpose**: Retrieves lost items based on filters.
- **Endpoint**: GET /api/lost-items
- **Process**: Builds a dynamic SQL query with optional filters (category, location, date, status, keyword) and joins related tables.
- **Front-End Integration**: Triggered by the search form in search.html.

Code Snippet:

```
app.get('/api/lost-items', async (req, res) => {

  const { category, location, date, status, keyword } = req.query;

  let query = `SELECT li.*, c.Category_Name, l.Building_Name,
u.First_Name, u.Last_Name FROM Lost_Item li JOIN Category c ON
li.Category_ID = c.Category_ID JOIN Location l ON li.Location_ID =
l.Location_ID JOIN User u ON li.Reported_By = u.User_ID WHERE 1=1`;

  const values = [];

  if (category) { query += ' AND c.Category_Name = ?';
values.push(category); }

  if (location) { query += ' AND l.Building_Name LIKE ?';
values.push(`%${location}%`); }

  if (date) { query += ' AND li.Lost_Date >= ?'; values.push(date); }

  if (status) { query += ' AND li.Status = ?'; values.push(status); }

  if (keyword) { query += ' AND (li.Item_Name LIKE ? OR li.Description
LIKE ?)'; values.push(`%${keyword}%`, `%${keyword}%`); }

  try {

    const [results] = await db.promise().query(query, values);

    res.json(results);

  } catch (err) {

    console.error('Error fetching lost items:', err);

    res.status(500).json({ error: 'Failed to fetch lost items' });

  }

});
```

- **Details**:
  - **Dynamic Query**: Uses WHERE 1=1 as a base for appending conditions.
  - **Joins**: Links Lost_Item with Category, Location, and User for comprehensive data.
  - **Filtering**: Supports partial matches with LIKE for location and keyword.

Search Found Items

- **Purpose**: Retrieves found items with filtering.
- **Endpoint**: GET /api/found-items
- **Process**: Similar to lost items, with adjustments for found item fields and status mapping.
- **Front-End Integration**: Displays results in search.html.

Code Snippet:

```
app.get('/api/found-items', async (req, res) => {

  const { category, location, date, status, keyword } = req.query;

  let query = `SELECT fi.*, c.Category_Name, l.Building_Name,
u.First_Name, u.Last_Name FROM Found_Item fi LEFT JOIN Category c ON
fi.Category_ID = c.Category_ID LEFT JOIN Location l ON fi.Location_ID
= l.Location_ID LEFT JOIN User u ON fi.Reported_By = u.User_ID WHERE
1=1`;

  const values = [];

  if (category) { query += ' AND c.Category_Name = ?';
values.push(category); }

  if (location) { query += ' AND l.Building_Name LIKE ?';
values.push(`%${location}%`); }

  if (date) { query += ' AND fi.Found_Date >= ?'; values.push(date); }

  if (status) {

    const dbStatus = status.toLowerCase() === 'found' ? 'Unclaimed' :
status;
```

```
    query += ' AND fi.Status = ?';

    values.push(dbStatus);

  }

  if (keyword) { query += ' AND (fi.Item_Name LIKE ? OR fi.Description
LIKE ?)'; values.push(`%${keyword}%`, `%${keyword}%`); }

  try {

    const [results] = await db.promise().query(query, values);

    res.json(results);

  } catch (err) {

    console.error('Error fetching found items:', err);

    res.status(500).json({ error: 'Failed to fetch found items' });

  }

});
```

- **Details**:
  - **LEFT JOIN**: Used to handle cases where category or location might be missing.
  - **Status Mapping**: Converts 'found' to 'Unclaimed' for consistency.

Update Operations

- **Purpose**: Updates the status of a lost or found item (e.g., by an admin).
- **Endpoint**: PUT /api/admin/items/:type/:id (hypothetical, based on trace).
- **Process**: Validates the status and updates the corresponding table.

Code Snippet :

```
app.put('/api/admin/items/:type/:id', async (req, res) => {

  const { type, id } = req.params;

  const { Status } = req.body;
  const validLostStatuses = ['Open', 'Claimed', 'Resolved'];

  const validFoundStatuses = ['Unclaimed', 'Claimed'];

  try {

    let query, table, idField;

    if (type === 'lost') {

      if (!validLostStatuses.includes(Status)) return
res.status(400).json({ error: 'Invalid status for lost item' });

      table = 'Lost_Item';

      idField = 'Lost_Item_ID';

    } else if (type === 'found') {

      if (!validFoundStatuses.includes(Status)) return
res.status(400).json({ error: 'Invalid status for found item' });

      table = 'Found_Item';

      idField = 'Found_Item_ID';

    } else {

      return res.status(400).json({ error: 'Invalid item type' });

    }

    query = `UPDATE ${table} SET Status = ? WHERE ${idField} = ?`;

    const [result] = await db.promise().query(query, [Status, id]);

    if (result.affectedRows === 0) return res.status(404).json({
error: 'Item not found' });

    res.json({ message: 'Status updated successfully' });

  } catch (err) {

    console.error('Error updating status:', err);
    res.status(500).json({ error: 'Failed to update status' });

  }

});
```

- **Details**:
    - **Validation**: Ensures status matches allowed values.
    - **Dynamic Table**: Switches between Lost_Item and Found_Item based on type.
    - **Feedback**: Returns 404 if no rows are affected.

Delete Operations:

- **Observation**: The provided code lacks explicit delete endpoints, which aligns with a lost and found system prioritizing data retention.
- **Conceptual Approach**: If implemented, a delete endpoint might look like:
    - **Endpoint**: DELETE /api/admin/items/:type/:id
    - **Purpose**: Removes an item (restricted to admins).

Code Snippet:

```
app.delete('/api/admin/items/:type/:id', async (req, res) => {

  const { type, id } = req.params;

  try {

    const table = type === 'lost' ? 'Lost_Item' : 'Found_Item';

    const idField = type === 'lost' ? 'Lost_Item_ID' :
'Found_Item_ID';

    const query = `DELETE FROM ${table} WHERE ${idField} = ?`;

    const [result] = await db.promise().query(query, [id]);

    if (result.affectedRows === 0) return res.status(404).json({
error: 'Item not found' });

    res.json({ message: 'Item deleted successfully' });
  } catch (err) {
    console.error('Error deleting item:', err);
    res.status(500).json({ error: 'Failed to delete item' });
  }
});
```

- **Details**:
  - **Authorization**: Should include admin checks (not implemented).
  - **Safety**: Retains data by default unless explicitly deleted.

Source code:

https://github.com/Amruth-hiremath/UniFind
