# Assignment 1: Basic Backend Development

**Release Date:** January 19, 2025

**Due Date:** February 9, 2025, 11:59 PM EST

**Weight:** 12.5% of final grade

## Overview

In this assignment, you will build the foundation of the Paper Management System by implementing a RESTful API using Express.js with SQLite as the database. You will create endpoints to manage paper metadata and implement proper error handling.

## Learning Objectives

After completing this assignment, you will be able to:

- Implement a RESTful API using Express.js
- Design and implement database operations using SQLite
- Handle HTTP requests and responses
- Implement proper error handling
- Write input validation middleware

## Requirements

### 1. Database Schema

Your implementation should create a SQLite database named `paper_management.db` in the root directory with the following schema:

```
CREATE TABLE papers (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    authors TEXT NOT NULL,
    published_in TEXT NOT NULL,
    year INTEGER NOT NULL CHECK (year > 1900),
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Note:

- The table creation should be implemented in `database.js`
- Timestamps are automatically managed by SQLite

## 2. API Endpoints

Implement the following REST API endpoints:

### POST /api/papers

Creates a new paper

- Required fields:
  - `title` : non-empty string
  - `authors` : non-empty string (comma-separated list of authors)
  - `published_in` : non-empty string
  - `year` : integer greater than 1900
- Automatic fields:
  - `id` : automatically generated primary key
  - `created_at` : automatically set to current timestamp by database
  - `updated_at` : automatically set to current timestamp by database
- Input Validation:
  - Ensure all required fields are present
  - Validate year is a valid integer greater than 1900

    - Return **all** validation errors together if multiple exist
- Responses:
    - 201: Created (returns the created paper with ID and timestamps)
    - 400: Bad Request (validation errors)

**Request Body:**

```
{
  "title": "Required, string",
  "authors": "Required, string",
  "published_in": "Required, string",
  "year": "Required, integer > 1900"
}
```

An Request Example:

```
{
  "title": "Example Paper Title",
  "authors": "John Doe, Jane Smith",
  "published_in": "ICSE 2024",
  "year": 2024
}
```

**Success Response:**

- Status: 201 Created
- Returns the created paper with all fields including automatic ones

```
{
  "id": "Auto-generated, integer",
  "title": "string",
  "authors": "string",
  "published_in": "string",
  "year": "integer",
  "created_at": "timestamp",
  "updated_at": "timestamp"
}
```

A Success Response Example:

```
{
  "id": 1,
  "title": "Example Paper Title",
  "authors": "John Doe, Jane Smith",
  "published_in": "ICSE 2024",
  "year": 2024,
  "created_at": "2025-01-19T10:30:00.000Z",
  "updated_at": "2025-01-19T10:30:00.000Z"
}
```

ℹ️ Timestamps in responses must be in ISO 8601 format. More details of timestamp format can be found in Implementation Notes

**Error Response:**

1. Missing Required Fields (400 Bad Request):

   Return **all** validation errors in the messages array to help API users fix all issues at once. The following is an example when all the four required fields are missing:

```
{
  "error": "Validation Error",
  "messages": [
    "Title is required",
    "Authors are required",
    "Published venue is required",
    "Published year is required"
  ]
}
```

2. Invalid Year (400 Bad Request):

```
{
  "error": "Validation Error",
  "messages": ["Valid year after 1900 is required"]
}
```

## GET /api/papers

Retrieves a list of papers with optional filtering and pagination

**Query Parameters:**

- `year`: integer - Filter papers by exact year match
- `published_in`: string - Filter by case-insensitive partial match
- `limit`: integer - Maximum number of results (default: 10, max: 100)
- `offset`: integer - Number of results to skip (default: 0)

**Input Validation:**

- `year` (if provided):
  - Must be an integer greater than 1900
- `limit`:
  - Must be a positive integer
  - Maximum value: 100
  - Default: 10
- `offset`:
  - Must be a non-negative integer
  - Default: 0

**Filter Behavior:**

- Query parameters should be processed exactly as provided, without trimming (e.g. a whitespace-only string `" "` should be treated as a string, not as an unprovided parameter) or rounding (e.g. floating-point numbers should not be rounded to integers)
- Range queries (e.g. `year=2019–2024`, `limit=10–20`) are not supported in this assignment. If encountered, the API should return the following error response

```
{
  "error": "Validation Error",
  "message": "Invalid query parameter format"
}
```

- Multiple filters are combined with AND logic
- If no filters are provided, returns all papers (subject to pagination)
- Returns empty array `[]` if no papers match the filters

**Success Response:**

- Status: 200 OK
- Returns an array of paper objects

```
[
  {
    "id": "Auto-generated, integer",
    "title": "string",
    "authors": "string",
    "published_in": "string",
    "year": "integer",
    "created_at": "timestamp",
    "updated_at": "timestamp"
  }
  // ... more papers
]
```

Timestamps in responses must be in ISO 8601 format. More details of timestamp format can be found in Implementation Notes

**Error Response:**

Invalid Query Parameters (400 Bad Request):

```
{
  "error": "Validation Error",
  "message": "Invalid query parameter format"
}
```

### Note: Simplified Error Handling

Error handling for invalid query parameters is simplfied in this assignment. Whether there is one invalid parameter or multiple invalid parameters, the response will always be:

```
{
  "error": "Validation Error",
  "message": "Invalid query parameter format"
}
```

For example, both these requests:

- `GET /api/papers?year=1000` (single invalid parameter)
- `GET /api/papers?year=1000&offset=-1` (multiple invalid parameters)

will receive the same error response shown above.

---

## GET /api/papers/:id

Retrieves a specific paper by its ID.

**URL Parameters:**

- `id` : integer - The ID of the paper to retrieve

**Validation Process:**

- Validate ID format:
  - Must be a valid integer
  - Must be positive
- Check paper existence:
  - Must exist in the database

**Success Response:**

- Status: 200 OK
- Returns a single paper object

```json
{
  "id": "integer",
  "title": "string",
  "authors": "string",
  "published_in": "string",
  "year": "integer",
  "created_at": "timestamp",
  "updated_at": "timestamp"
}
```

Timestamps in responses must be in ISO 8601 format. More details of timestamp format can be found in Implementation Notes

**Error Responses:**

1. Invalid ID Format (400 Bad Request):

```json
{
  "error": "Validation Error",
  "message": "Invalid ID format"
}
```

2. Paper Not Found (404 Not Found):

```json
{
  "error": "Paper not found"
}
```

---

## PUT /api/papers/:id

Updates an existing paper by its ID.

**URL Parameters:**

- `id` : integer - The ID of the paper to update

### Input Validation:

- All validation rules from POST apply
- ID must be valid and exist

### Request Body:

```json
{
  "title": "Required, string",
  "authors": "Required, string",
  "published_in": "Required, string",
  "year": "Required, integer > 1900"
}
```

A Request Example:

```json
{
  "title": "Updated Paper Title",
  "authors": "John Doe, Jane Smith",
  "published_in": "IEEE TSE",
  "year": 2024
}
```

### Success Response:

- Status: 200 OK
- Returns the updated paper with all fields
- `updated_at` timestamp is automatically updated
- `created_at` timestamp remains unchanged

```json
{
  "id": "integer",
  "title": "string",
  "authors": "string",
  "published_in": "string",
  "year": "integer",
  "created_at": "timestamp",
  "updated_at": "timestamp"
}
```

Timestamps in responses must be in ISO 8601 format. More details of timestamp format can be found in Implementation Notes

### Error Response (400 Bad Request):

1. Invalid ID Format (400 Bad Request):

```
{
  "error": "Validation Error",
  "message": "Invalid ID format"
}
```

2. Missing Required Fields (400 Bad Request):

```
{
  "error": "Validation Error",
  "messages": [
    "Title is required",
    "Authors are required",
    "Published venue is required",
    "Published year is required"
  ]
}
```

3. Invalid Year (400 Bad Request):

```
{
  "error": "Validation Error",
  "messages": ["Valid year after 1900 is required"]
}
```

4. Paper Not Found (404 Not Found):

```
{
  "error": "Paper not found"
}
```

### Note: Validation Processing Logic

When handling `PUT` requests with multiple potential errors (e.g. an invalid ID and an invalid request body), always validate the ID first before checking the request body. This is because:

- Path parameters (like ID) should be validated before processing the request body
- If the ID is invalid, there's no need to validate the request body since the resource cannot be identified
- This aligns with the logical request processing flow:
  1. Validate the ID format
  2. Validate the update data
  3. Check if the resource exists

For example: A `PUT /api/papers/0` request with the request body

```
{
  "authors": "John Doe",
  "published_in": "ICSE 2024",
  "year": 2024
  // missing title
}
```

The response should be:

```
{
  "error": "Validation Error",
  "message": "Invalid ID format"
}
```

Since the ID validation fails first, the request body is not checked.

---

## DELETE /api/papers/:id

Deletes a specific paper by its ID.

**URL Parameters:**

- `id` : integer - The ID of the paper to delete

**Input Validation:**

- ID must be a valid integer
- ID must exist in the database

**Success Response:**

- Status: 204 No Content
- No response body

**Error Responses:**

1. Invalid ID Format (400 Bad Request):

```json
{
  "error": "Validation Error",
  "message": "Invalid ID format"
}
```

2. Paper Not Found (404 Not Found):

```json
{
  "error": "Paper not found"
}
```

## Implementation Notes

1. Data Format Requirements:
    - All timestamps in responses must be in ISO 8601 format (e.g. "2025-01-19T10:30:00.000Z") and in the UTC timezone (indicated by the 'Z' suffix)
    - Convert the timestamp generated by SQLite to ISO 8601 format. Since SQLite provides timestamps with second-level precision, the converted timestamp should end with `"000Z"`. There is no need to generate a new timestamp for milliseconds precision
    - String inputs are used exactly as provided (including any leading or trailing spaces)
2. Response Requirements:
    - All responses must follow the JSON formats specified in this document

- Timestamps in responses must be in ISO 8601 format
- Response structures must exactly match the specifications

3. Search/Filter Behavior:
    - `year` must be exact match
    - `published_in` must be case-insensitive partial match
    - `limit` defaults to 10 if not provided
    - `offset` defaults to 0 if not provided

4. Input Processing:
    - Required fields must not be:
        - null
        - undefined
        - empty string ( `""` )
        - string containing only spaces ( `"  "` )
    - Year must be a valid integer greater than 1900
    - When a string input is valid, it should be stored exactly as provided (including any leading or trailing spaces)

## Project Structure

```
assignment1/
├── src/
│   ├── server.js        # Express application setup (DO NOT MODIFY)
│   ├── database.js      # Database operations (TODO: Implement)
│   ├── routes.js        # API routes (TODO: Implement)
│   └── middleware.js    # Custom middleware (TODO: Implement)
└── package.json
```

⚠️ You only need to implement the code in `database.js`, `routes.js`, and `middleware.js`. The `server.js` file is already set up correctly and should not be modified.

## Getting Started

1. Clone the Git repository containing starter code:

```
git clone https://github.com/cying17/ece1724-web-assignments.git
```

The starter code of Assignment 1 can be found under assignment1/

The starter code provides:

- A complete `server.js` with Express setup (DO NOT MODIFY)
- Basic structure for `database.js`, `routes.js`, and `middleware.js`
- TODOs and hints for implementation
- Test cases for verification

You need to:

- Implement the database operations in `database.js`
- Implement the route handlers in `routes.js`
- Implement the middleware functions in `middleware.js`

ℹ️ Feel free to modify the starter code in any file except `src/server.js`, but limit your changes to only what is necessary to meet the requirements specified in this handout.

ℹ️ The starter code and sample test cases of the other three assignments will also be pushed to this Git repository.

2. Navigate to assignment directory:

```
cd ece1724-web-assignments/assignment1
```

3. Install dependencies:

```
npm install
```

4. Start the development server:

```
npm run dev
```

This will start the server with `nodemon`, which automatically restarts when you make changes.

Note:

- The server runs on port 3000
- The database file ( `paper_management.db` ) will be created automatically in the root directory
- You can use tools like Thunder Client or cURL to test your API endpoints

## Testing Your Implementation

Sample test cases are provided under `assignment1/tests/` to help you verify your implementation.

The provided test cases cover basic functionality of your implementation. Passing these tests indicates that your basic implementation is working, but does not guarantee full marks.

### Writing Your Own Tests

You are encouraged to write additional test cases. `sample.test.js` is provided as a reference to demonstrate:

- How to structure test cases
- How to test API endpoints
- How to handle database operations in tests
- How to verify responses

You should write additional tests to cover:

- Edge cases

- Error scenarios
- Input validation
- Different filter combinations
- Database operations

## Running the Tests

1. Make sure you have installed all dependencies:

```
npm install
```

2. Run a specific test file:

```
npm test -- tests/sample.test.js
```

3. Run all tests:

```
npm test
```

4. Run tests in watch mode:

```
npm test -- --watch
```

Watch mode automatically reruns tests when you make changes, which is useful during development.

## Important Notes

1. Error Messages:
   - The auto-grader will check for **exact** error messages
   - Make sure your error responses match the formats specified in this handout
   - Verify all required fields and constraints

2. Additional Testing:
    - The provided sample tests only cover basic functionality
    - Additional test cases will be used for grading
    - You should test edge cases yourself
    - Test all error scenarios mentioned in the requirements
    - Check all filtering and pagination options

3. Testing Tips:
    - Implement and test one endpoint at a time
    - Use watch mode during development ( `--watch` flag)
    - Test both success and error cases
    - Verify response formats match the specifications exactly
    - Clean test data between runs
    - Check status codes carefully
    - Test all required fields and constraints
    - Make sure your error responses exactly match the specified formats

## Submission Instructions

1. Create a zip file named `assignment1.zip` with the following structure:

```
assignment1.zip/
├── src/
│   ├── server.js        # Express application setup
│   ├── database.js      # Database operations
│   ├── routes.js        # API routes
│   └── middleware.js    # Custom middleware
├── package.json
└── README.md
```

! Ensure that when the zip file is extracted, the `src/` directory, `package.json`, and `README.md` appear directly inside the extracted folder, without an additional nested `assignment1/` directory. **Submissions with incorrect structures may not be graded correctly**.

⚠️ Do **NOT** include the `node_modules/` directory in your submission, as it is typically large. The auto-grader will run `npm install` to install all dependencies specified in `package.json`.

Note:

- The `README.md` is included to encourage good documentation practices. While it won't be graded, it's an important part of real-world development.
- Your `README.md` should provide basic instructions on how to run your project, such as `npm install`, and a brief overview of your project. This is to help you practice writing clear and professional documentation, which is highly valued in the industry.

2. Submit the zip file to Quercus

## Grading Scheme (100 points)

1. Database Operations (5 points)
   - Table creation with correct schema (5)
2. API Endpoints (95 points)
   - `POST /api/papers` (20)
     - Successful creation (5)
     - Error handling and validation (15)
       - Missing fields (6)
         - Not provided (2)
         - String of only spaces (2)
         - `null` (2)
       - Invalid `year` (8)
         - `1900` (2)

        - `"1901a"` (2)
        - `1901.1` (2)
        - `""` (2)
    - Multiple validation errors (1)
- `GET /api/papers` (35)
  - Basic retrieval (5)
  - Filtering and pagination (10)
    - `year` filter (2)
    - `published_in` case-insensitive partial match (3)
    - `limit` and `offset` (3)
    - Default `limit` (2)
  - Error handling (20)
    - Invalid `year` (8)
      - `1900` (2)
      - `1901a` (2)
      - `1901.0` (2)
      - `2019-2024` (2)
    - Invalid `limit` (6)
      - `90.5` (2)
      - `90a` (2)
      - `0` (2)
    - Invalid `offset` (6)
      - `1.5` (2)
      - `1a` (2)
      - `-1` (2)
- `GET /api/papers/:id` (10)

    - Successful retrieval (4)
    - Error handling (6)
        - Invalid `id` (4)
            - `abc` (2)
            - `1a` (2)
        - Not found error (2)
- `PUT /api/papers/:id` (20)
    - Successful update (5)
    - Error handling and validation (15)
        - Invalid `id` (4)
            - `abc` (2)
            - `1a` (2)
        - Missing fields (5)
            - Not provided (2)
            - String of only spaces (2)
            - `null` (1)
        - Invalid `year` (4)
            - `1900` (2)
            - `"1901a"` (1)
            - `"1901.1"` (1)
        - Not found error (2)
- `DELETE /api/papers/:id` (10)
    - Successful deletion (4)
    - Error handling (6)
        - Invalid `id` (4)

    - `abc` (2)
    - `1a` (2)
  - Not found error (2)

> ℹ️ Handling an invalid id like `1a` is crucial because improper validation can expose your system to security risks.
>
> For example, if you use raw SQL queries without validation, an attacker could try `id=1 OR 1=1` in the URL (`GET /api/papers/1 OR 1=1`). This could trick your database into returning all records instead of just one.
>
> Additionally, invalid input might cause unexpected errors that leak sensitive information. Always validate input to prevent these issues.

## Implementation Requirements

1. Server Configuration:
   - Server must run on port 3000
   - Database file must be named `paper_management.db` in root directory
2. Code Requirements:
   - Use async/await for all database operations
   - Implement proper input validation before database operations
   - Follow error response formats exactly
   - Return appropriate HTTP status codes
   - All responses must be in JSON format
3. Error Handling:
   - Validate all inputs
   - Handle all error cases specified in requirements
   - Include precise error messages as specified
   - Implement proper database error handling

## Grading Process

Your submission will be graded automatically. The auto-grader will:

1. Set up your project
2. Run test cases against your API
3. Verify database operations
4. Check error handling

Make sure to test your implementation thoroughly before submission.

## Development Tips

1. Systematic Approach:
   - Start with database schema implementation
   - Implement endpoints one at a time
   - Test each endpoint thoroughly before moving to next
   - Follow the provided TODOs and hints
2. Testing Strategy:
   - Run tests frequently during development
   - Test both success and error cases
   - Verify response formats match specifications
   - Test edge cases manually
3. Common Pitfalls to Avoid:
   - Not following error response formats exactly
   - Missing input validation
   - Incorrect HTTP status codes
   - Not handling all error cases
   - Using callbacks instead of async/await

## Resources

- API Development:
  - Express.js Documentation
  - SQLite Documentation
  - REST API Best Practices
- Course Materials:
  - Lecture 1: JavaScript Fundamentals
  - Lecture 2: Backend Development Fundamentals

## Questions?

1. Discussion Board:
   - Post questions on course discussion board
   - Search existing discussions first
   - Use clear titles and provide relevant code snippets
2. Office Hours:
   - Time: Wednesday, 2:00 PM — 4:00 PM
   - Location: Room 7206, Bahen Centre for Information Technology
3. Tips for Getting Help:
   - Start early to allow time for questions
   - Be specific about your problem
   - Share what you've tried
   - Include relevant error messages

Last updated on February 26, 2025