CS331 Project: Retrieval Augmented Generation
Assignment 7

Ahlad Pataparla — 2201017
Anushka Srivastava — 2201030
Arya Sahu — 2201033
Khushi Mandal — 2201108

March 31, 2025

Contents

1 Part A: Data Access Layer (DAL) Implementation

1.1 Introduction to the Data Access Layer

The Data Access Layer (DAL) is a crucial architectural component that serves as an abstraction between the application logic and the database. It encapsulates all database operations, providing a clean API for the rest of the application to interact with the data store without being concerned with the underlying implementation details.

Key benefits of a well-designed DAL include:

- **Separation of Concerns**: Isolates database operations from business logic
- **Code Reusability**: Common data access patterns are implemented once and reused
- **Maintainability**: Changes to the database schema require modifications only in the DAL
- **Testability**: Mock implementations can be substituted for testing
- **Flexibility**: Switch database technologies with minimal impact on application code

1.2 Database Schema Design

For our fashion e-commerce application, we've designed a database schema that satisfies the requirements for storing user information and clothing items.

1.2.1 User Management Schema

```
-- User table for storing regular users
CREATE TABLE user (
    username VARCHAR(25) NOT NULL PRIMARY KEY,
    email VARCHAR(255) NOT NULL UNIQUE,
    password_hash CHAR(60) NOT NULL
);

-- Administrator table for storing admin users
CREATE TABLE administrator (
    username VARCHAR(25) NOT NULL PRIMARY KEY,
    email VARCHAR(255) NOT NULL UNIQUE,
    password_hash CHAR(60) NOT NULL,
    security_question VARCHAR(250) NOT NULL,
    security_answer_hash CHAR(60) NOT NULL
);
```

Listing 1: User and Administrator Schema

1.2.2 Fashion Items Schema

```
-- Clothing Items table
CREATE TABLE clothing_items (
    id BIGINT,
    gender TEXT,
```

```
    masterCategory TEXT,
    subCategory TEXT,
    articleType TEXT,
    baseColour TEXT,
    season TEXT,
    year DOUBLE,
    usage TEXT,
    productDisplayName TEXT
);
```

Listing 2: Clothing Items Schema

The clothing_items table stores fashion product data from the styles_filtered.csv dataset, which contains approximately 44,000 records with various clothing attributes.

1.3 Data Access Layer Implementation

My implementation focuses on creating a straightforward data access layer that facilitates database connections and query execution. The DAL is implemented in Node.js using the MySQL2 library with connection pooling for improved performance.

1.3.1 Database Connection Module

The foundation of our DAL is a database connection module that establishes and manages connections to MySQL:

```
"use strict";

const config = require("./config");
const mysql = require("mysql2");

const pool = mysql.createPool({
    host: config.MYSQL_HOST,
    user: config.MYSQL_USER,
    password: config.MYSQL_PASSWORD,
    database: config.MYSQL_DATABASE,
    waitForConnections: true,
    connectionLimit: 10,
    queueLimit: 0
}).promise();

module.exports = pool;
```

Listing 3: Database Connection Module (db.js)

This module creates a connection pool with a maximum of 10 connections, which are automatically managed. The `promise()` method ensures all database operations return promises for easier async/await usage.

1.3.2 User Management Routes

The user management functionality is split into several route handlers that interact with the database:

```
// signup.js route handler
signupRouter.post("/", async (request, response) => {
  let { username, email, password } = request.body;
```

```
4
5   // Validate required fields
6   if (!username || !password || !email) {
7     return response.status(400).json({
8       error: "username, password, and email are required",
9     });
10  }
11
12  // Trim input fields
13  username = username.trim();
14  password = password.trim();
15  email = email.trim();
16
17  // Validate username
18  if (username.length < 3) {
19    return response.status(400).json({
20      error: "username must be at least 3 characters long",
21    });
22  }
23
24  // Validate password
25  if (password.length < 3) {
26    return response.status(400).json({
27      error: "password must be at least 3 characters long",
28    });
29  }
30
31  // Validate email
32  if (!validator.validate(email)) {
33    return response.status(400).json({
34      error: "Invalid email",
35    });
36  }
37
38  // Check for existing username
39  const [userWithUsername] = await dbConn.query(
40    "SELECT * FROM user WHERE username=?",
41    [username],
42  );
43  if (userWithUsername.length !== 0) {
44    return response.status(409).json({
45      error: "A user with that username already exists",
46    });
47  }
48
49  // Check for existing email
50  const [userWithEmail] = await dbConn.query(
51    "SELECT * FROM user WHERE email=?",
52    [email],
53  );
54  if (userWithEmail.length !== 0) {
55    return response.status(409).json({
56      error: "A user with that email already exists",
57    });
58  }
59
60  // Hash password
61  const saltRounds = 10;
```

```
  const passwordHash = await bcrypt.hash(password, saltRounds);

  // Insert new user
  await dbConn.query(
    "INSERT INTO user (username, email, password_hash) VALUES (?, ?, ?)
    ",
    [username, email, passwordHash],
  );

  return response.status(201).end();
});
```

Listing 4: User Signup Implementation

```
// Login implementation for standard users
loginRouter.post("/user", async (request, response) => {
  const { username, password } = request.body;

  const query = "SELECT * FROM user WHERE username=?";
  const [rows] = await dbConn.query(query, [username]);
  const userWithUsername = rows[0];

  if (!userWithUsername) {
    return response.status(401).json({
      error: "Invalid username",
    });
  }

  const passwordCorrect = await bcrypt.compare(
    password,
    userWithUsername.password_hash,
  );

  if (!passwordCorrect) {
    return response.status(401).json({
      error: "Invalid password",
    });
  }

  const userForToken = {
    username: userWithUsername.username,
    email: userWithUsername.email,
  };

  const token = jwt.sign(userForToken, process.env.SECRET, {
    expiresIn: 60 * 60,
  });

  response.status(200).send({
    token,
    username: userWithUsername.username,
    email: userWithUsername.email,
    type: "standard_user",
  });
});
```

Listing 5: User Login Implementation

These route handlers directly interact with the database to perform user operations:

- The signup handler validates user input, checks for existing users, and creates new accounts
- The login handler verifies credentials and generates JWT tokens for authenticated users
- Additional handlers (not shown) handle user management and password changes

1.3.3 Product Data Access

The products are managed through FastAPI in Python, demonstrating the flexibility of our data access approach across different technologies:

```python
@app.get("/api/products", response_model=ProductsResponse)
async def get_products(limit: int = 10):
    """Return a selection of products."""
    if ml_model.df is None or ml_model.df.empty:
        raise HTTPException(status_code=500, detail="Product data not available.")

    try:
        # Ensure limit is reasonable
        limit = min(limit, len(ml_model.df))
        limit = max(1, limit)  # Ensure limit is at least 1

        random_ids = sample(ml_model.df["id"].tolist(), limit)
        # Use get_item to ensure consistent data retrieval and formatting
        products = [Item(**get_item(pid)) for pid in random_ids if get_item(pid) is not None]

        if not products:
            return ProductsResponse(products=[])

        return ProductsResponse(products=products)
    except Exception as e:
        logger.error(f"Error fetching products: {e}", exc_info=True)
        raise HTTPException(status_code=500, detail="Error fetching products")
```

Listing 6: Product API Implementation (Simplified)

2 Part B: Testing

This section provides an analysis of White Box Testing and Black Box Testing approaches applied to the e-commerce application's Data Access Layer.

2.1 White Box Testing

White Box Testing (also known as Glass Box or Structural Testing) examines the internal structures, logic, and code paths of the application. It requires knowledge of the implementation details and focuses on:

- Code coverage (statement, branch, path coverage)

- Internal logic and data structures
- Error handling and exception paths
- Control flow and conditional logic

2.1.1 Test Case 1: User Creation

This test focuses on verifying the password hashing functionality and error handling in the user creation process:

```
// signup.test.js
const signupHandler = require('../routes/signup');
const dbConn = require('../utils/db');
const bcrypt = require('bcrypt');

jest.mock('../utils/db');
jest.mock('bcrypt');

describe('User Signup', () => {
  let req, res;

  beforeEach(() => {
    req = {
      body: {
        username: 'testuser',
        email: 'test@example.com',
        password: 'password123'
      }
    };

    res = {
      status: jest.fn().mockReturnThis(),
      json: jest.fn().mockReturnThis(),
      end: jest.fn()
    };

    // Mock database responses
    dbConn.query.mockImplementation((query, params) => {
      if (query.includes('SELECT') && params[0] === 'testuser') {
        return [[]]; // No existing username
      }
      if (query.includes('SELECT') && params[0] === 'test@example.com')
    {
        return [[]]; // No existing email
      }
      return [{ affectedRows: 1 }]; // Successful insert
    });

    bcrypt.hash.mockResolvedValue('hashed_password');
  });

  test('should create a new user with valid data', async () => {
    // Call route handler directly
    await signupHandler.post(req, res);

    // Verify response
    expect(res.status).toHaveBeenCalledWith(201);
```

```
    expect(res.end).toHaveBeenCalled();

    // Verify bcrypt was called
    expect(bcrypt.hash).toHaveBeenCalledWith('password123', 10);

    // Verify DB query was called with correct parameters
    expect(dbConn.query).toHaveBeenCalledWith(
      'INSERT INTO user (username, email, password_hash) VALUES (?, ?, ?)',
      ['testuser', 'test@example.com', 'hashed_password']
    );
  });

  test('should handle database errors', async () => {
    // Set up db.query to throw an error
    dbConn.query.mockRejectedValueOnce(new Error('Database connection error'));

    // Call route handler
    await signupHandler.post(req, res);

    // Verify error handling
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({ error: 'Database connection error' });
  });
});
```

Listing 7: White Box Test for User Signup

2.1.2 Test Case 2: Login Handler - Authentication Flow

This test verifies the authentication logic in the login handler:

```
// login.test.js
const loginHandler = require('../routes/login');
const dbConn = require('../utils/db');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');

jest.mock('../utils/db');
jest.mock('bcrypt');
jest.mock('jsonwebtoken');

describe('User Login', () => {
  let req, res;

  beforeEach(() => {
    req = {
      body: {
        username: 'testuser',
        password: 'password123'
      }
    };

    res = {
      status: jest.fn().mockReturnThis(),
      json: jest.fn().mockReturnThis(),
```

```
    send: jest.fn()
  };

  process.env.SECRET = 'test_secret';
  jwt.sign.mockReturnValue('test-jwt-token');
});

test('should return token on successful login', async () => {
  // Mock database returning a user
  dbConn.query.mockResolvedValue([[{
    username: 'testuser',
    email: 'test@example.com',
    password_hash: 'hashed_password'
  }]]);

  // Mock password verification
  bcrypt.compare.mockResolvedValue(true);

  // Call the route handler
  await loginHandler.post('/user', req, res);

  // Verify response
  expect(res.status).toHaveBeenCalledWith(200);
  expect(res.send).toHaveBeenCalledWith({
    token: 'test-jwt-token',
    username: 'testuser',
    email: 'test@example.com',
    type: 'standard_user'
  });

  // Verify JWT was created properly
  expect(jwt.sign).toHaveBeenCalledWith(
    { username: 'testuser', email: 'test@example.com' },
    'test_secret',
    { expiresIn: 60 * 60 }
  );
});

test('should return 401 for invalid username', async () => {
  // Mock database returning no users
  dbConn.query.mockResolvedValue([[]]);

  // Call the route handler
  await loginHandler.post('/user', req, res);

  // Verify response
  expect(res.status).toHaveBeenCalledWith(401);
  expect(res.json).toHaveBeenCalledWith({ error: 'Invalid username'
 });
});

test('should return 401 for incorrect password', async () => {
  // Mock user in database but incorrect password
  dbConn.query.mockResolvedValue([[{
    username: 'testuser',
    email: 'test@example.com',
    password_hash: 'hashed_password'
  }]]);
```

```
82
83      // Password comparison fails
84      bcrypt.compare.mockResolvedValue(false);
85
86      // Call the route handler
87      await loginHandler.post('/user', req, res);
88
89      // Verify response
90      expect(res.status).toHaveBeenCalledWith(401);
91      expect(res.json).toHaveBeenCalledWith({ error: 'Invalid password'
        });
92    });
93  });
```

Listing 8: White Box Test for Login Authentication

2.2 Black Box Testing

Black Box Testing (also known as Functional Testing) evaluates the behavior of the application without knowledge of its internal implementation. It focuses on:

- Input/output validation
- Functional requirements verification
- User interface testing
- Integration between components
- End-to-end workflows

2.2.1 Test Case 1: User Registration API

This test verifies the user registration functionality from the client perspective:

```
1  // userRegistration.test.js
2  const request = require('supertest');
3  const app = require('../app');
4
5  describe('User Registration API', () => {
6      test('should register a new user with valid data', async () => {
7          // Given
8          const userData = {
9              username: 'newuser',
10             email: 'new@example.com',
11             password: 'securepassword'
12         };
13
14         // When
15         const response = await request(app)
16             .post('/api/signup')
17             .send(userData);
18
19         // Then
20         expect(response.status).toBe(201);
21     });
22
```

```
23      test('should reject registration with existing username', async () =>
        {
24          // Given
25          const userData = {
26              username: 'existinguser', // Assume this user exists
27              email: 'unique@example.com',
28              password: 'password123'
29          };
30
31          // When
32          const response = await request(app)
33              .post('/api/signup')
34              .send(userData);
35
36          // Then
37          expect(response.status).toBe(409);
38          expect(response.body).toHaveProperty('error');
39          expect(response.body.error).toContain('already exists');
40      });
41
42      test('should reject registration with invalid email', async () => {
43          // Given
44          const userData = {
45              username: 'validuser',
46              email: 'not-an-email',
47              password: 'password123'
48          };
49
50          // When
51          const response = await request(app)
52              .post('/api/signup')
53              .send(userData);
54
55          // Then
56          expect(response.status).toBe(400);
57          expect(response.body).toHaveProperty('error');
58          expect(response.body.error).toContain('Invalid email');
59      });
60
61      test('should reject registration with short password', async () =>
        {
62          // Given
63          const userData = {
64              username: 'validuser',
65              email: 'valid@example.com',
66              password: 'pw'  // Too short
67          };
68
69          // When
70          const response = await request(app)
71              .post('/api/signup')
72              .send(userData);
73
74          // Then
75          expect(response.status).toBe(400);
76          expect(response.body).toHaveProperty('error');
77          expect(response.body.error).toContain('must be at least');
78      });
```

```
});
```

Listing 9: Black Box Test for User Registration API

2.2.2 Test Case 2: User Authentication API

This test verifies the login functionality from the client perspective:

```
// userAuthentication.test.js
const request = require('supertest');
const app = require('../app');

describe('User Authentication API', () => {
    test('should authenticate with valid credentials', async () => {
        // Given
        const credentials = {
            username: 'testuser',  // Assume this user exists
            password: 'password123'
        };

        // When
        const response = await request(app)
            .post('/api/login/user')
            .send(credentials);

        // Then
        expect(response.status).toBe(200);
        expect(response.body).toHaveProperty('token');
        expect(response.body).toHaveProperty('username', 'testuser');
        expect(response.body).toHaveProperty('type', 'standard_user');
    });

    test('should reject authentication with invalid username', async ()
    => {
        // Given
        const credentials = {
            username: 'nonexistentuser',
            password: 'password123'
        };

        // When
        const response = await request(app)
            .post('/api/login/user')
            .send(credentials);

        // Then
        expect(response.status).toBe(401);
        expect(response.body).toHaveProperty('error');
        expect(response.body.error).toBe('Invalid username');
    });

    test('should reject authentication with invalid password', async ()
    => {
        // Given
        const credentials = {
            username: 'testuser',  // Assume this user exists
            password: 'wrongpassword'
        };
```

```
49
50          // When
51          const response = await request(app)
52              .post('/api/login/user')
53              .send(credentials);
54
55          // Then
56          expect(response.status).toBe(401);
57          expect(response.body).toHaveProperty('error');
58          expect(response.body.error).toBe('Invalid password');
59      });
60
61      test('should reject authentication with missing fields', async ()
    => {
62          // Given
63          const credentials = {
64              username: 'testuser'
65              // Password missing
66          };
67
68          // When
69          const response = await request(app)
70              .post('/api/login/user')
71              .send(credentials);
72
73          // Then
74          expect(response.status).toBe(400);
75          expect(response.body).toHaveProperty('error');
76      });
77  });
```

Listing 10: Black Box Test for User Authentication API

2.2.3 Test Case 3: Product Search and Filtering API

This test verifies the product search and filtering functionality:

```
1  // productSearch.test.js
2  const request = require('supertest');
3  const app = require('../app');
4
5  describe('Product Search API', () => {
6      test('should return products matching search query', async () => {
7          // When
8          const response = await request(app)
9              .get('/api/products/search?query=shirt');
10
11         // Then
12         expect(response.status).toBe(200);
13         expect(response.body).toHaveProperty('products');
14         expect(Array.isArray(response.body.products)).toBe(true);
15
16         // Check if returned products match search term
17         if (response.body.products.length > 0) {
18             const containsTerm = response.body.products.some(product =>
19                 product.productDisplayName.toLowerCase().includes('
    shirt') ||
20                 product.articleType.toLowerCase().includes('shirt') ||
```

```
                product.baseColour.toLowerCase().includes('shirt')
            );
            expect(containsTerm).toBe(true);
        }
    });

    test('should filter products by category', async () => {
        // When
        const response = await request(app)
            .get('/api/products/category/shirts');

        // Then
        expect(response.status).toBe(200);
        expect(response.body).toHaveProperty('products');
        expect(Array.isArray(response.body.products)).toBe(true);

        // Check if returned products are in the correct category
        if (response.body.products.length > 0) {
            const inCategory = response.body.products.some(product =>
                product.masterCategory.toLowerCase() === 'shirts' ||
                product.subCategory.toLowerCase() === 'shirts' ||
                product.articleType.toLowerCase() === 'shirts'
            );
            expect(inCategory).toBe(true);
        }
    });

    test('should filter products by gender', async () => {
        // When
        const response = await request(app)
            .get('/api/products?gender=Men');

        // Then
        expect(response.status).toBe(200);
        expect(response.body).toHaveProperty('products');
        expect(Array.isArray(response.body.products)).toBe(true);

        // Check if returned products are for the correct gender
        if (response.body.products.length > 0) {
            const correctGender = response.body.products.every(product
=>
                product.gender === 'Men'
            );
            expect(correctGender).toBe(true);
        }
    });

    test('should paginate product results', async () => {
        // When
        const response = await request(app)
            .get('/api/products?page=2&limit=10');

        // Then
        expect(response.status).toBe(200);
        expect(response.body).toHaveProperty('products');
        expect(Array.isArray(response.body.products)).toBe(true);
        expect(response.body.products.length).toBeLessThanOrEqual(10);

```

```
78          // Check pagination metadata if present
79          if (response.body.pagination) {
80              expect(response.body.pagination).toHaveProperty('
    currentPage', 2);
81              expect(response.body.pagination).toHaveProperty('pageSize',
     10);
82          }
83      });
84  });
```

Listing 11: Black Box Test for Product Search API

3 Conclusion

In this assignment, we have successfully implemented a comprehensive Data Access Layer (DAL) for a fashion e-commerce application and designed both White Box and Black Box tests to ensure its functionality and reliability.

The DAL implementation follows a straightforward approach, directly leveraging MySQL2's connection pooling capabilities to interact with the database. This design provides an efficient abstraction layer that separates the database operations from the application logic, making the system more maintainable and testable.

The testing approach combines:

- **White Box Testing**: To verify internal logic, code paths, and error handling by targeting specific code paths and internal functionality
- **Black Box Testing**: To validate functional requirements and external behavior by testing the API endpoints from the client perspective