HMS T&L System

Technical Documentation
Group 15

JAVA CAN'T SEE SHARP

Front to Back, We're on Track

Deliverable Two: Back-End

(CMPG323)

Group Members:

41425626 – Z. Hansa – (Group Leader)

37833707 – S. Ndobela

42304954 – S. Nhlapo

35928654 – T Gumbi

Submission date: 27/09/2024

Table of Contents

Technology Stack

Front-end	React
Back-end	Node.Js with Express
Database	Azure SQL Database
File Management System	Azure Blob Storage
File conversion and compression	ffmeg library
Authentication and Authorization	JWT
API Documentation and Testing	Swagger
Containerization and Deployment	Docker

Why for React for Frontend?

- **Performance**: React's virtual DOM efficiently updates only the parts of the page that have changed, which optimizes rendering and enhances performance.
- **Scalability**: React's component-based architecture allows for building scalable applications, making it easier to manage and maintain as the application grows.
- **Data Model Flexibility**: React is flexible in terms of integrating with various state management solutions (e.g., Redux, Context API), which gives us control over how data flows and is managed in our app.
- **Ecosystem and Tooling**: React has a vast ecosystem with numerous libraries and tools available. Its strong community support provides access to cutting-edge features and comprehensive documentation.
- **Cost**: As an open-source library, React is free to use, with no licensing costs.
- **Use Case Suitability**: React is ideal for building dynamic user interfaces with a focus on responsive and interactive design, which is critical for the web app where lecturers and students interact with video content.
- **Analytics & Data Analysis:** React can be integrated with analytics tools (e.g., Google Analytics, Segment) for tracking user interactions.

Why Node.js with Express for Backend?

- **Performance**: Node.js is built on Chrome's V8 engine, which makes it fast and efficient for I/O-bound operations. This is crucial for handling large amounts of video uploads and real-time interactions.
- **Scalability:** Node.js supports horizontal scaling by spawning multiple processes or running instances across different servers. It's ideal for building scalable applications that need to handle multiple concurrent requests.
- **Consistency vs Availability:** Node.js is a good fit for systems where availability is a priority, especially when combined with Express, as it handles large-scale traffic with non-blocking I/O.
- **Ecosystem and Tooling:** With Express, Node.js benefits from a huge ecosystem of modules available via npm, enabling fast development and integration with a wide range of services (e.g., Swagger, JWT).
- **Cost:** Both Node.js and Express are open-source, with no associated costs, making it a cost-effective choice.

- **Use Case Suitability**: Node.js excels at handling real-time features like video uploads, compression, and notifications, which align well with our project's requirements.

Why Azure SQL Database?

- **Performance:** Azure SQL Database provides high-performance capabilities, such as in-memory processing, which enhances the speed of complex queries.
- **Scalability:** It scales dynamically, with automatic scaling options that can handle increasing workloads, making it ideal for growing applications.
- **Consistency vs Availability:** As a relational database, Azure SQL focuses on strong consistency, ensuring that data is always accurate and up to date, which is important when handling sensitive video-related data.
- **Data Model Flexibility:** While relational databases are more rigid compared to NoSQL, Azure SQL supports flexible schema changes without major disruptions to the application.
- **Query Complexity and Support:** SQL is well-suited for complex queries, joins, and transactional integrity, which will be essential when querying video metadata, user data, and assignment records.
- **Transaction Support:** Azure SQL provides full ACID (Atomicity, Consistency, Isolation, Durability) compliance, ensuring reliable transaction processing, which is vital for maintaining the integrity of the video uploads and user interactions.
- **Ecosystem and Tooling:** Azure SQL integrates seamlessly with other Azure services, including Azure Blob Storage for file management, and provides built-in analytics and monitoring tools.
- **Cost:** Azure SQL offers flexible pricing models based on usage, allowing the team to manage costs effectively.
- **Use Case Suitability:** As a relational database, Azure SQL is well-suited for systems requiring structured data storage and complex querying, which is essential for managing assignment data and user details.

Why Azure Blob Storage?

- **Performance:** Azure Blob Storage is optimized for large-scale file storage and provides fast access to video files with built-in CDN (Content Delivery Network) support for efficient content delivery.
- **Scalability:** It can easily scale to store vast amounts of video data, supporting our need for video uploads, storage, and streaming.
- **Consistency vs Availability:** Azure Blob Storage prioritizes availability, ensuring that video files can be accessed and streamed reliably even under high load.
- **Data Model Flexibility:** It supports flexible storage formats, which makes it ideal for storing videos and related metadata.
- **Ecosystem and Tooling:** It integrates smoothly with Azure services, including Azure SQL and Azure Functions, providing a unified ecosystem for file storage, processing, and analytics.
- **Cost:** Azure Blob Storage offers cost-effective pricing for large-scale data storage, with pricing tiers based on usage, allowing cost optimization.
- **Use Case Suitability:** Its support for streaming, large-file storage, and scalability make it the perfect choice for storing video submissions.
- **ETL Processes:** Azure Blob Storage integrates well with ETL tools (e.g., Azure Data Factory) for managing and transforming large data sets, which could be useful for video compression and metadata extraction.

Why ffmpeg Library

- **Performance**: ffmpeg is highly efficient at compressing and converting video files, which is essential for reducing file sizes and optimizing storage and streaming.
- **Scalability**: It can be integrated with Node.js to handle bulk video processing tasks in a scalable way, supporting high-throughput compression tasks.
- **Use Case Suitability**: As a mature, widely-used library, ffmpeg supports a variety of formats and codecs, making it ideal for video compression and format conversion
- **Cost**: ffmpeg is open-source, making it free to use with no licensing costs.
- **Ecosystem and Tooling:** It's widely supported and has excellent community documentation, ensuring easy integration and support.

Why Swagger?

We chose Swagger because it provides an easy, interactive way to document and test our API directly in the browser. Swagger integrates smoothly with our codebase, automatically generating API documentation from comments and making it easy for developers to understand and use the endpoints. While Postman is great for manual API testing, it lacks built-in documentation generation. OpenAPI is the specification that Swagger is built on, but Swagger offers a more user-friendly interface for both testing and documentation, making it the better choice for our project.

- **Performance**: Swagger enhances development efficiency by auto-generating API documentation, reducing the time spent manually documenting APIs.
- **Scalability**: It scales well with growing projects, as it allows easy maintenance of API documentation as new endpoints are added.
- **Ecosystem and Tooling**: Swagger integrates seamlessly with Express and Node.js, offering tools for API testing, versioning, and visualization.
- **Cost**: Swagger is open-source, and it has a free version with enough functionality for our project needs.

Why Docker?

https://nwu-hms-g6fjckard7fggqar.southafricanorth-01.azurewebsites.net/api-docs/

We chose Docker because it makes deploying our app easier by packaging everything it needs into containers. This ensures the app runs the same way on any machine, avoiding issues like "it works on my computer." Docker is faster and more efficient than traditional virtual machines, using fewer resources. It's also great for testing and updating the app easily. Overall, Docker is simple, reliable, and perfect for our deployment needs.

- **Performance**: Docker containers optimize resource usage by running lightweight, isolated applications, which enhances the performance of our deployment environments.
- **Scalability**: Docker supports horizontal scaling by allowing multiple containers to run across different machines, enabling our system to scale effortlessly.
- **Ecosystem and Tooling:** Docker is highly supported with integrations into CI/CD pipelines and deployment platforms, making it ideal for automated testing and deployment.
- **Cost**: Docker is open-source, with additional enterprise features available at a cost, though the free version should be sufficient for our project.
- **Use Case Suitability**: Docker allows us to standardize our development, testing, and production environments, ensuring consistency across different platforms.

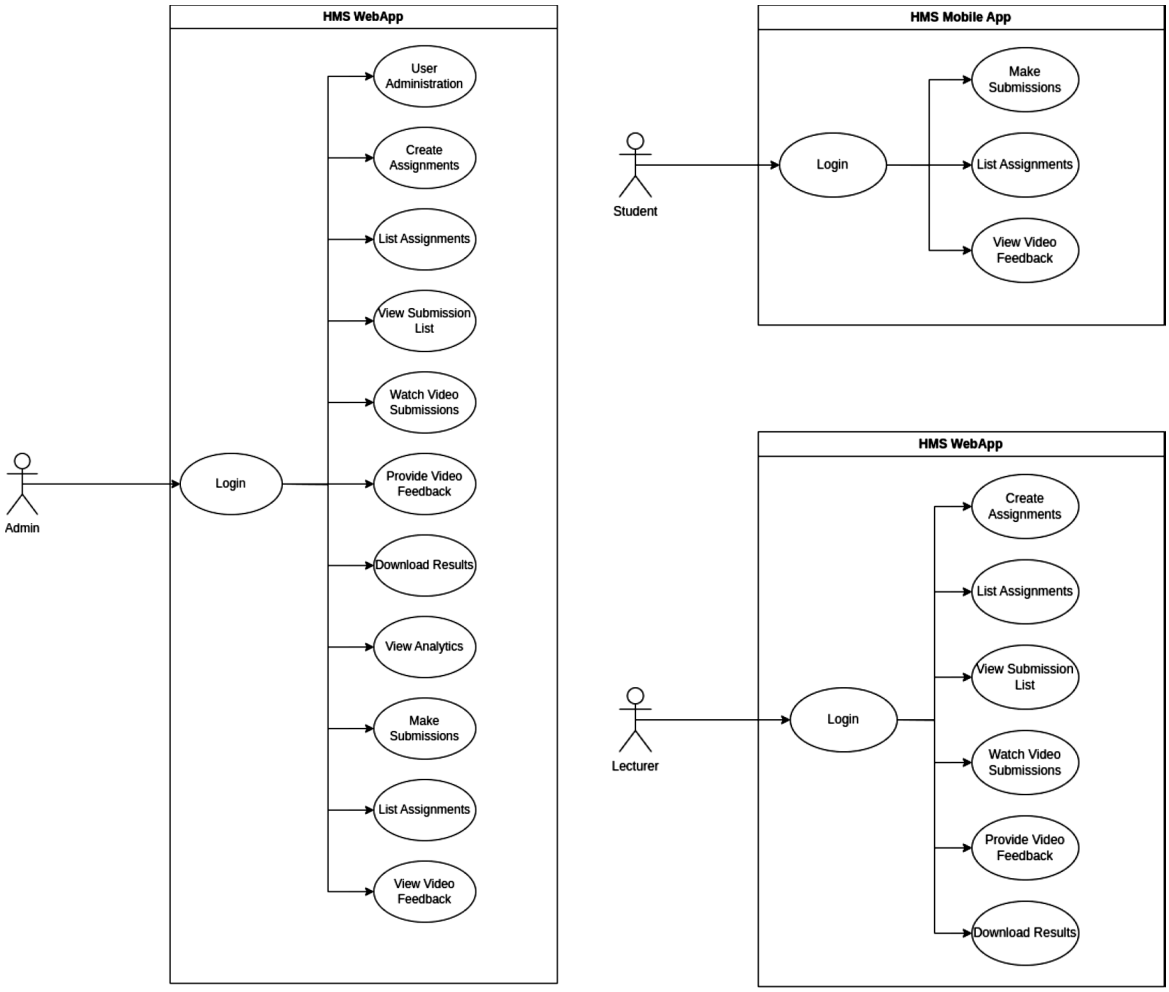
Why JWT?

We chose JWT over other authentication mechanisms like OAuth because it's simpler and more efficient for our needs. JWT allows us to authenticate users by issuing tokens that securely carry user data (like roles) without needing to store sessions on the server. OAuth, while more powerful, is complex and best for things like third-party logins (e.g., via Google or Facebook). Since our app only needs basic authentication, JWT provides a lightweight and straightforward solution

- **Performance**: JWT tokens are compact and stateless, providing efficient performance for user authentication and session management.
- **Scalability**: JWTs are stateless, meaning they don’t require server-side storage, which simplifies scaling as the system grows.
- **Security**: JWTs can be securely signed and encrypted, ensuring that sensitive information (like login credentials) is protected.
- **Ecosystem and Tooling**: JWT integrates well with Node.js and Express, providing easy-to-implement authentication mechanisms.
- **Cost**: JWT is open-source, and there are no associated costs with using it for authentication in our system.
- **Use Case Suitability**: JWT is well-suited for systems requiring secure, stateless authentication, which fits the needs of our project.

Planning

Use Case Diagram



User Roles:

1. Admin: This user role represents the project owner and/or sponsor and the developers maintaining the system. This user has unrestricted access to all components of the system.

2. Lecturer: This user role represents the lecturers within the faculty that will make use of this system.

3. Student: This user role represents the students within the faculty that will make use of this system.

Analysis of Database and Database Schema Components

1. tblCourse

Attributes:

- CourseID (Primary Key, Auto-incremented): Unique identifier for each course.
- CourseCode: A unique course code.
- CourseName: The name of the course.

- Duration: Represents the length of the course (e.g., in weeks or months).

Purpose: This table stores information related to the courses available in the system. Each course can have multiple modules associated with it, as shown in the tblModuleOnCourse table.

2. tblUser

Attributes:

- UserID (Primary Key, Auto-incremented): Unique identifier for each user.
- UserNumber: A unique identifier for each user (e.g., student or staff number).
- PasswordHash: The encrypted password for authentication.
- FirstName, LastName, Email: Basic personal information for each user.
- UserRole: This stores the role of the user (e.g., Admin, Lecturer, or Student).
- CreatedAt: Timestamp for when the user was created.
- CourseID (Foreign Key): Links each user to a course, indicating which course they are part of.

Purpose: This table stores all users of the system, including students and lecturers, and tracks which course they are associated with. It supports authentication (via password) and defines user roles for authorization purposes.

3. tblModule

Attributes:

- ModuleID (Primary Key, Auto-incremented): Unique identifier for each module.
- ModuleCode: A unique identifier for the module.
- ModuleName: The name of the module.
- Description: A brief overview of the module (optional).
- Lecturer (Foreign Key): Links each module to the lecturer responsible for teaching it.

Purpose: This table stores information about modules within a course. Each module can be linked to a course via tblModuleOnCourse and can have assignments related to it.

4. tblModuleOnCourse

Attributes:

- ID (Primary Key, Auto-incremented): Unique identifier for this table’s rows.
- CourseID (Foreign Key): Links to the relevant course.
- ModuleID (Foreign Key): Links to the relevant module.

Purpose: This table creates a many-to-many relationship between courses and modules, allowing multiple modules to be associated with multiple courses.

5. tblAssignment

Attributes:

- AssignmentID (Primary Key, Auto-incremented): Unique identifier for each assignment.
- ModuleID (Foreign Key): Links to the module the assignment belongs to.
- Title: The name of the assignment.
- Instructions: Detailed instructions for the assignment (optional).
- CreatedAt: Timestamp for when the assignment was created.
- DueDate: Deadline for the assignment submission.

Purpose: This table stores assignment information, allowing lecturers to create tasks that students will submit videos for. The table is linked to the module and is central to the submission process.

6. tblVideo

Attributes:

- VideoID (Primary Key, Auto-incremented): Unique identifier for each video.
- VideoURL: The storage location or URL of the uploaded video.
- CompressedFilePath: Path to the compressed version of the video (optional).

Purpose: This table stores metadata about the videos that students submit. Videos are uploaded as part of the submission process and may be compressed for storage efficiency, in line with the system's video management needs.

7. tblStudentModuleEnrollment

Attributes:

- EnrollmentID (Primary Key, Auto-incremented): Unique identifier for enrollment records.
- StudentID (Foreign Key): Links to the student (user) enrolled in a module.
- ModuleID (Foreign Key): Links to the module that the student is enrolled in.

Purpose: This table manages the enrollment of students in specific modules, allowing tracking of which students are part of which modules.

8. tblSubmission

Attributes:

- SubmissionID (Primary Key, Auto-incremented): Unique identifier for each submission.
- AssignmentID (Foreign Key): Links the submission to the specific assignment.
- StudentID (Foreign Key): Links the submission to the student making it.
- VideoID (Foreign Key): Links to the video file associated with the submission.
- SubmittedAt: Timestamp for when the submission was made.
- Status: The status of the submission (e.g., Submitted, Reviewed).
- SubmissionText: Additional comments or details related to the submission (optional).

Purpose: This table stores information about each assignment submission, linking students to their video submissions for specific assignments.

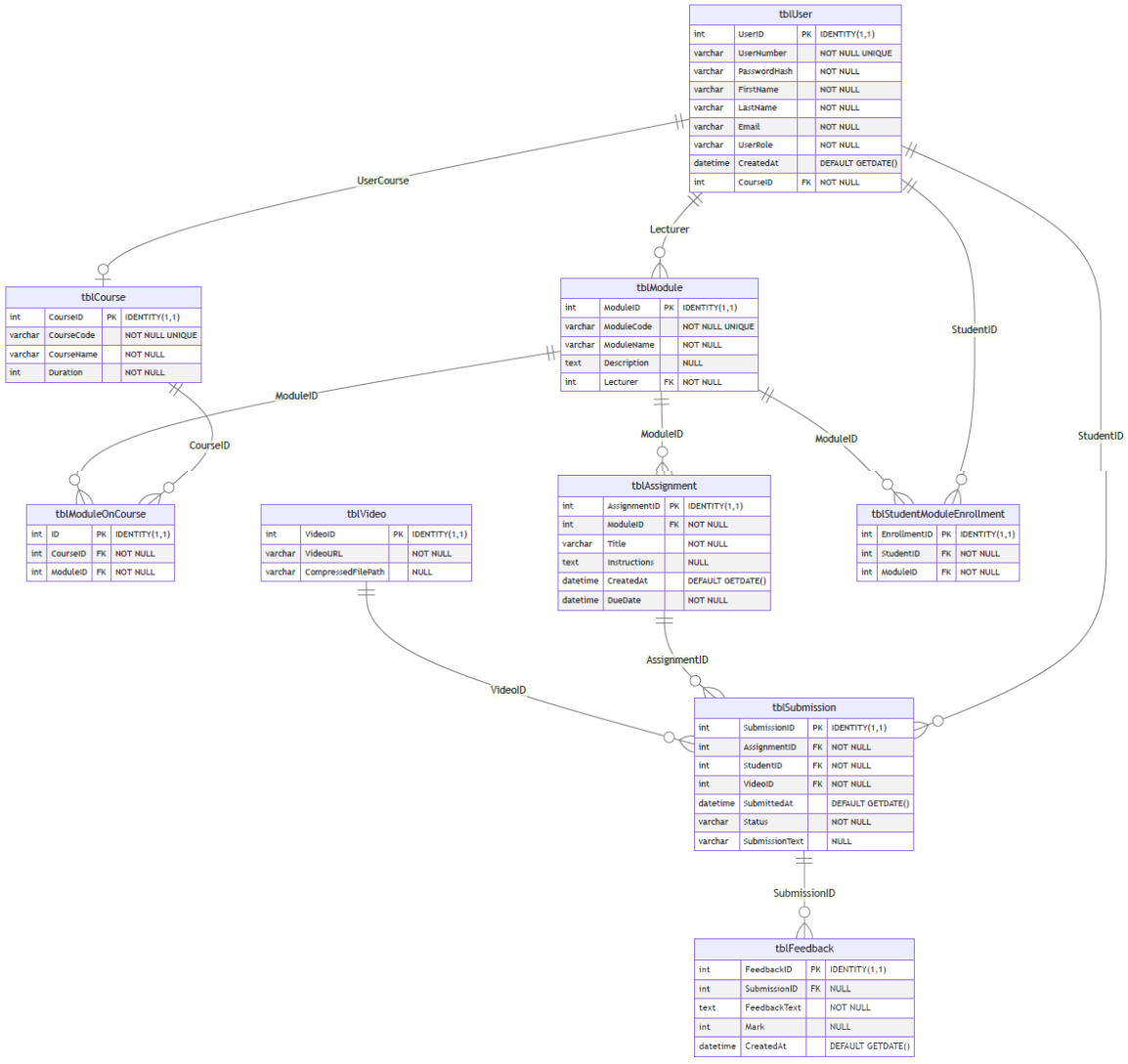
9. tblFeedback

Attributes:

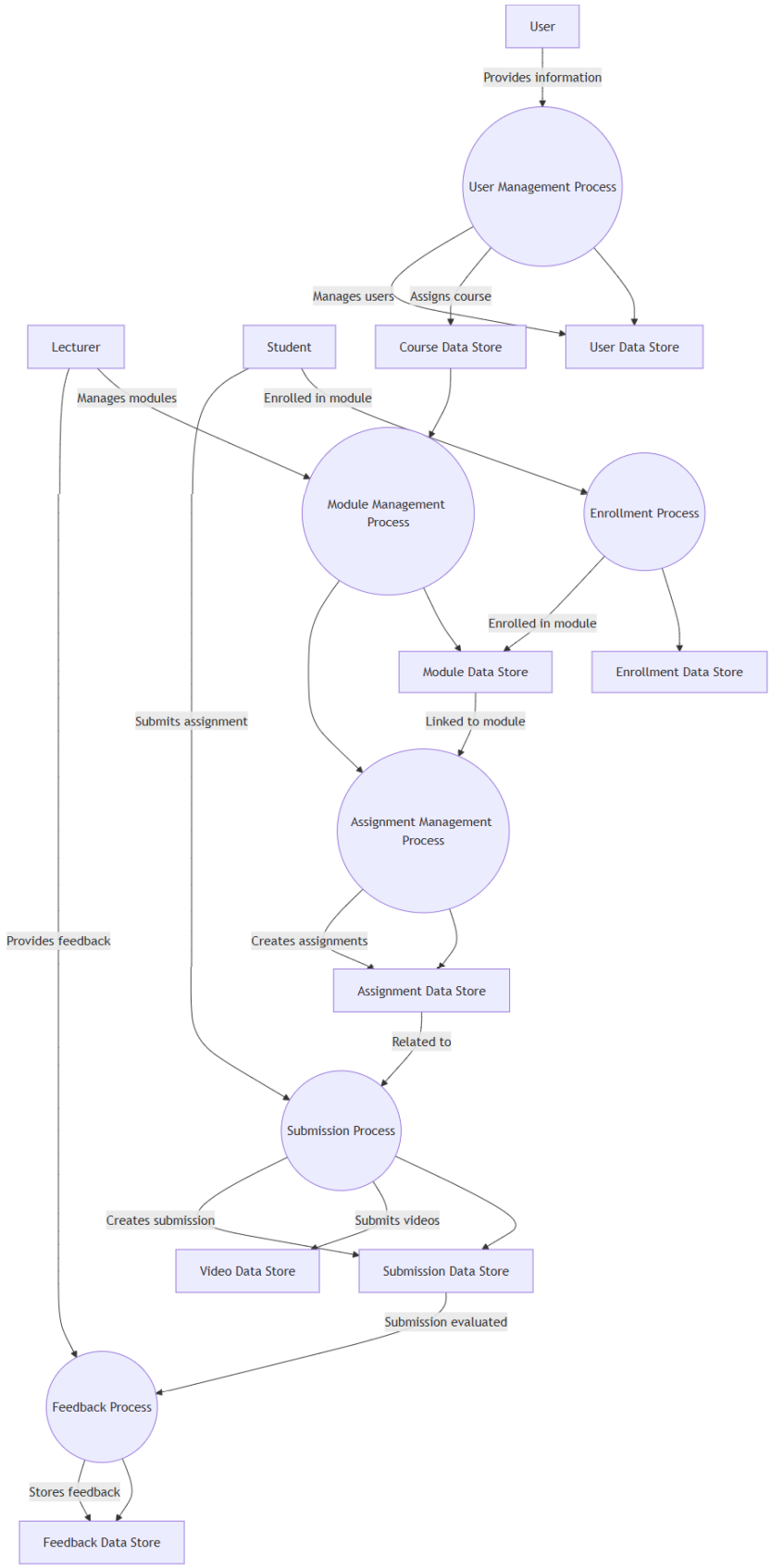
- FeedbackID (Primary Key, Auto-incremented): Unique identifier for each feedback record.
- SubmissionID (Foreign Key): Links feedback to a specific submission.
- FeedbackText: Text containing feedback from the lecturer.
- Mark: The grade or mark given for the submission (optional).
- CreatedAt: Timestamp for when the feedback was created.

Purpose: This table stores the feedback provided by lecturers for student submissions, including the mark or grade given.

ERD



Data Flow Diagram



Development Plan

Our group decided to approach development using the 40-20-40 rule.

The 40-20-40 rule states that 40% of the work is performed during feasibility, analysis, and design; 20% is during coding; and the remaining 40% is during testing and debugging

This rule is important because it enforces a comprehensive, well-structured approach to software development. It helps ensure that sufficient time is dedicated to planning, testing, and quality assurance, which ultimately leads to a more successful and reliable product. Focusing too much on coding at the expense of analysis, design, and testing can result in software that is difficult to maintain, prone to errors, and misaligned with user needs.

Development

Our application is structured as a Node.js project, with the following key components:

.env:	Environment variables configuration.
config:	Contains configuration files for the project.
controllers:	Houses the logic for handling requests and controlling the flow of data.
middlewares:	Custom or third-party middleware for processing requests and responses.
node_modules:	Directory containing all the dependencies.
package.json:	Lists dependencies and scripts, defining the project.
routes:	Defines API endpoints.
server.js:	The entry point of the application.
swagger:	Contains Swagger-related files for API documentation.
swaggerConfig.js:	Configuration for Swagger for defining API schemas

.

Implementation of Criteria

User Management

User management is essential for security and user experience.

- **User registration** allows new users to create accounts with secure password hashing.
- **Login** verifies credentials and generates a JWT for authenticated access, enabling users to access personalized features.
- **Logout** allows users to end their sessions, enhancing security by preventing unauthorized access. In all, the features create unique identities for users, ensure secure access to resources, and provide control over their accounts, making the application safer and more user-friendly.

Data Management

The API supports efficient data management by implementing structured methods for storing, retrieving, processing, and maintaining data and data integrity. It uses organized database schemas to ensure that data is stored consistently and adheres to integrity constraints. Efficient CRUD operations are facilitated through optimized queries, allowing quick data retrieval and updates. Additionally, data validation checks are integrated to prevent invalid entries, ensuring data integrity throughout its lifecycle. By managing data effectively, the API enhances reliability and performance, providing a robust framework for handling data efficiently.

Error Handling

The Error handling uses try-catch blocks to manage issues during database operations and authentication. If an error occurs, a 500-status code is returned with a descriptive error message. For login failures, a 401-status code is used for invalid credentials. The JWT authentication middleware returns 403 for unauthorized access. This approach ensures that clients receive clear feedback on errors, making it easier to understand what went wrong and aiding in debugging.

Examples of error handling for API requests in our application:

- **Authentication errors**: If the JWT token is missing or invalid, the API returns a 401 Unauthorized or 403 Forbidden error with relevant messages like "Unauthorized: Bearer token missing" or "Access denied".
- **Database errors**: If a database query fails (e.g., during user registration), we return a 500 Internal Server Error, with messages like "Error registering user" or "Error retrieving users".
- **Resource not found**: For non-existing users or routes, we return 404 Not Found errors with messages like "User not found".

Authentication

In our app, we implemented secure API endpoints using JWT (JSON Web Token) for authentication. Upon login, users are issued a JWT signed with a secret key, which they must include in the Authorization header for all protected requests. The authanticateJWT middleware extracts and verifies this token, ensuring it's valid and unexpired. Additionally, certain endpoints require role-based access control, meaning the user's role (stored in the token) is checked to authorize actions. This mechanism ensures that only authenticated and authorized users can access sensitive routes, providing both security and control over API usage.

Endpoints

CRUD Endpoints for main entities.

The main entities in our application include:

- Assignments - Post, Get, Put, Delete
- Courses - Post, Get, Put, Delete
- Feedback - Post, Get, Put, Delete
- Modules - Post, Get, Put, Delete
- Submissions - Post, Get, Put, Delete
- Users - Post, Get, Put, Delete

List of all endpoints:

POST /auth/register POST /auth/login POST /auth/logout	
POST /assignments GET /assignments GET /assignments/module/{ModuleID} GET /assignments/{id} PUT /assignments/{id} DELETE /assignments/{id}	POST /enrollments GET /enrollments DELETE /enrollments/{id}
POST /feedbacks GET /feedbacks GET /feedbacks/{id} PUT /feedbacks/{id} DELETE /feedbacks/{id} GET /feedbacks/download/csv	GET /courses POST /courses GET /courses/{id} PUT /courses/{id} DELETE /courses/{id}
POST /module-on-course DELETE /module-on-course/{id}	GET /api/files/download/{id} GET /api/files/stream/{id}
POST /modules GET /modules GET /modules/{id} PUT /modules/{id} DELETE /modules/{id}	POST /submissions GET /submissions GET /submissions/{id} PUT /submissions/{id} DELETE /submissions/{id}
GET /users GET /users/{id} PUT /users/{id} DELETE /users/{id}	POST /videos GET /videos GET /videos/{id} DELETE /videos/{id}

Security

Authentication Tokens and basic authentication checks (e.g. role-based access control).

User roles and JWT (JSON Web Token) authentication work together to control access to application resources. Each user is assigned a specific UserRole (e.g., Admin, Lecture), defining their permissions. Upon logging in, users receive a JWT containing their role information, which is sent with each request in the Authorization header.

The server checks for the token's presence and validity. If valid, it verifies the user's role against the required roles for that route. If the user's role doesn't match, access is denied with a 403 Forbidden error. This ensures users only access authorized resources, enhancing security.

JWT Authentication:

- Upon login, users receive a JWT token that contains their user details (e.g. UserNumber, UserRole).
- This token is passed in the Authorization header for future API requests.
- The token is verified using a secret key to ensure the request is authentic.

Authorization Checks:

- The system checks if the Authorization header is present and if it contains a valid token.
- If the token is missing, expired, or invalid, the user receives an error (either 401 Unauthorized or 403 Forbidden).

Role-Based Access Control (UserRole):

- The middleware checks the user’s role to restrict access to certain routes.
- Only users with the required roles (like admin) can access specific functions.
- If the user’s role doesn’t match, they receive a 403 Forbidden error.

Architecture

Clear and Structured API architecture using RESTful Principles that separates concerns between API endpoints and Business Logic.

Coding Standards

1.Consistent Naming: Endpoints have clear names that describe what they do (like /users for user information and /videos for video content), making it easy to understand their purpose.

2.Standard HTTP Methods: It uses common HTTP methods correctly—like GET for getting data, POST for creating new data, PUT for updating existing data, and DELETE for removing data—so developers know what to expect.

3.Clear Documentation: The API includes simple and easy-to-read documentation that explains how to use each endpoint, what data to send, and what responses to expect.

Testing

Unit testing checks individual parts of our app to ensure they work correctly on their own. We use testing frameworks like Jest or Mocha to create tests for functions, such as those handling user registration and authentication. Each test runs different inputs to confirm the function behaves as expected. This is important because it helps catch bugs early, improves code quality, and makes it easier to change or update code later. Overall, unit testing gives us confidence that our app works well and continues to function properly as we develop it.

Response Time

Optimisation of response times for CRUD operations.

The API enhances response times for CRUD (Create, Read, Update, Delete) operations by employing efficient database queries to quickly access and modify data. It utilizes caching for frequently accessed information, reducing redundant database calls. Batch processing allows multiple CRUD actions in one request, further speeding up operations. Asynchronous processing enables the API to handle multiple requests simultaneously, ensuring improved responsiveness. These strategies collectively ensure that users enjoy fast and efficient interactions with the API, optimizing overall performance.

Optimisation

Efficient Database Queries and Indexing.

The API optimizes response times through efficient database queries and indexing. It uses well-structured SQL queries that only fetch necessary data and filter results appropriately. By creating indexes on frequently accessed columns, the API speeds up data retrieval. It also analyses query execution plans to ensure efficient data access and reduces complex joins in the database schema. These strategies work together to enhance performance, allowing for faster data operations and improved overall efficiency of the API.

Scalability

Load Testing and considerations for handling concurrent API requests.

The API supports scalability by incorporating load testing practices that assess its ability to handle multiple concurrent requests efficiently. This involves simulating high traffic scenarios to identify performance bottlenecks and ensure that response times remain acceptable under increased loads. By optimizing code and database queries based on load testing results, the API can manage scalability better. Additionally, it can leverage techniques like caching and horizontal scaling to distribute the load across multiple servers, allowing for seamless growth and improved performance as user demand increases.

CI/CD

Automated CI/CD pipelines for building, testing, and deploying the Back-End, with version control and tagging.

The API supports Continuous Integration and Continuous Deployment (CI/CD) through automated pipelines that simplify the development process. These pipelines build and test the code whenever changes are made, ensuring that new features don’t break existing functions. Version control tools, like Git, help track changes and manage code effectively. Tagging releases allows for easy deployment of stable versions and quick rollbacks if needed. This approach improves code quality and speeds up updates, leading to a more reliable API.

https://nwu-hms-g6fjckard7fgggar.southafricanorth-01.azurewebsites.net/api-docs/

Monitoring

Monitoring of API Performance metrics (e.g. response time, error rates, etc.)

The API supports monitoring by integrating performance metrics tracking for key indicators such as response times and error rates. It uses middleware to collect data on each API request, measuring how long it takes to respond and recording any errors that occur. This information is then sent to a monitoring dashboard, where it can be visualized and analyzed. By continuously tracking these metrics, the API enables developers to identify performance

bottlenecks, assess the health of the system, and make informed decisions to optimize and improve the API's overall performance.

Logging

Implementation of logging for API requests and errors.

The API supports logging by implementing a structured logging system that records all API requests and errors. It captures essential details like request timestamps, endpoints accessed, and HTTP methods used. Additionally, it logs any errors that occur during processing, along with stack traces and relevant context. This information is stored in a centralized log management system, allowing for easier monitoring and debugging. By maintaining comprehensive logs, the API ensures better visibility into its operations, which aids in identifying issues and improving overall performance.