# BrightBoard –
# A Course and Quiz System

CSCI 441 VA: Software Engineering

Jun 26, 2025

Team Members: Shadow Love-Erckert, Conner Erckert

Project URL:
GitHub URL:[https://github.com/Element713/CSCI441_BrightBoard.git](https://github.com/Element713/CSCI441_BrightBoard.git)

## Work Assignment

### a. Individual Contributions Breakdown

While both team members actively collaborated on all aspects of the project, specific responsibilities were divided to optimize workflow based on individual strengths. The following table summarizes each member's contributions across major report and development components.

| Component / Task | Shadow Love-Erckert | Conner Erckert |
|---|---|---|
| **1. Requirements Specification** | | |
| Functional Requirements (Use Cases) | A/R | A/R |
| Non-Functional Requirements (FURPS+) | A/R | A/R |
| Glossary of Terms | A/R | A/R |
| Requirements Consolidation | A/R | A/R |
| **2. Domain Modeling** | | |
| Entity Identification | R | R |
| Relationship Mapping | C | R |
| Mongoose Schema Implementation | C | A/R |
| Data Validation Rules | A/R | A/R |
| **3. Software Design** | | |
| Backend Architecture Design (Express.js) | C | A/R |
| File Structure Planning | C | A/R |
| Middleware (JWT Auth, Validation) | C | A/R |
| Routing Logic | A/R | A/R |
| API Endpoint Specification | A/R | A/R |
| **4. Report Preparation** | | |

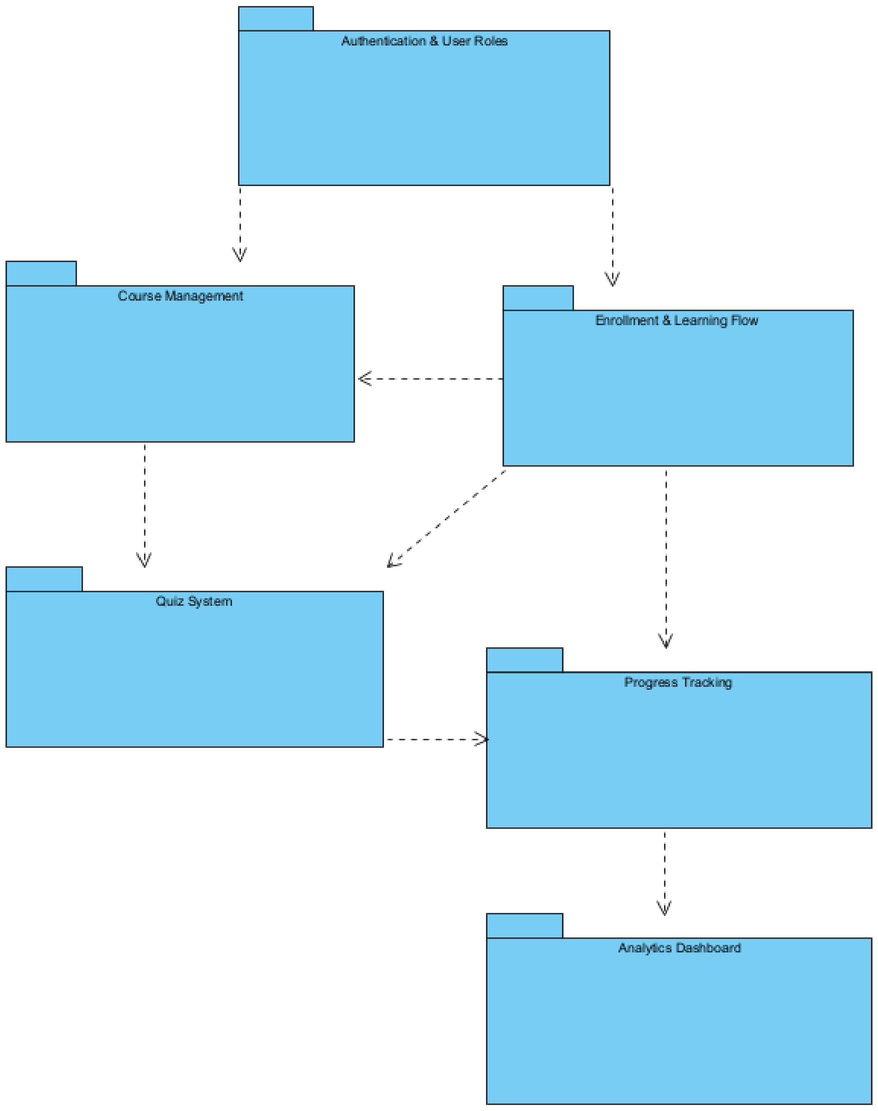
| | | |
|---|---|---|
| Cover Page, Purpose, Work Plan, TOC | A/R | C |
| Requirements Section | A/R | A/R |
| Functional & UI Specification | A/R | C |
| Analysis | A/R | A/R |
| Formatting and Final Edits | A/R | A/R |
| Diagrams and Illustrations | A/R | A/R |
| **5. Other Contributions** | | |
| UI Design & Styling | R | C |
| Frontend Development (React, CSS) | R | A |
| Backend Integration (MongoDB, Express) | C | R/A |
| Quiz System Development | A/R | A/R |
| Testing & Debugging | A/R | A/R |
| Deployment (Local & Final) | A/R | A/R |
| Integration Management | A/R | C |
| Team Communication & Weekly Check-ins | A | C |

# Table of Contents

## 6. System Architecture and System Design

### a. Identifying Subsystems

The architecture of BrightBoard was designed with simplicity, modularity, and scalability in mind, catering specifically to independent educators and small learning cohorts. The system provides a centralized platform for course delivery, quiz administration, and student progress tracking, removing the need for multiple disjointed tools.



**UML Package Diagram Description:**

While not presented as a formal diagram in the original report, the subsystems of BrightBoard can be visualized in a package-oriented UML structure. The **Authentication** module connects with both **Course Management** and **Enrollment**, setting the foundation for user access and interaction. **Course Management** interfaces with the **Quiz System**, while **Progress Tracking** and **Analytics** tie together performance data from across the platform. This modular setup supports extensibility and maintenance.

### b. Architectural Styles

BrightBoard is structured around a client-server architecture, a well-established architectural style that separates the user interface (client) from the core application logic and data handling (server). This separation of concerns ensures that the system remains maintainable, scalable, and easy to deploy in diverse environments. The frontend and backend operate independently yet communicate seamlessly via a well-defined interface, allowing for modular development and the potential for future upgrades or third-party integrations.

On the client side, BrightBoard uses React, a modern JavaScript library for building user interfaces. React was chosen for its component-based architecture, which promotes code reusability, state management, and fast rendering performance. This makes it ideal for delivering an interactive and responsive user experience across various devices, including desktops, tablets, and smartphones. Components are responsible for rendering views, managing user interactions (like course enrollment or quiz taking), and sending asynchronous requests to the backend via RESTful APIs.

The backend is implemented using Node.js with the Express.js framework. This combination allows for fast, non-blocking server-side logic and robust API creation. Express handles routing, middleware, and HTTP request management, serving as the backbone of server-side operations. Business logic, such as user authentication, lesson processing, quiz scoring, and progress tracking, is processed server-side to ensure consistency, security, and centralized control.

BrightBoard also integrates the Model-View-Controller (MVC) architectural pattern within its backend logic. This layered structure enhances clarity and separation within the codebase:

- Models are created using Mongoose, an ODM (Object Data Modeling) library for MongoDB. These schemas define the structure and constraints of data entities such as users, courses, lessons, and quizzes.

- Controllers manage the business logic and serve as intermediaries between models and views. For example, when a student submits a quiz, the controller validates the data, calculates the score, and updates the relevant records.

- Views, while traditionally rendered server-side in MVC, are managed entirely on the client side via React components in BrightBoard's architecture. This hybrid model allows for a clean API-driven interface and decouples backend logic from frontend rendering responsibilities.

This combination of client-server and MVC styles provides several advantages. It allows teams to work in parallel on frontend and backend components, improves scalability by enabling horizontal distribution (e.g., load-balancing frontend servers), and simplifies debugging by keeping concerns well-organized. Additionally, the system is well-prepared for future enhancements, such as adding real-time features via WebSockets or expanding to mobile platforms using React Native.

In summary, BrightBoard's architectural design ensures both flexibility and robustness, using tried-and-true patterns that support the project's goals of maintainability, user responsiveness, and straightforward deployment.

### c. Mapping Subsystems to Hardware

BrightBoard is designed to operate across multiple machines, following a distributed system model. The frontend runs in the user's web browser, serving both students and instructors. The backend operates on a remote server, handling all authentication, course logic, and data storage. The database, MongoDB, may be hosted either locally or on a cloud instance, depending on deployment needs. This setup ensures that user-facing interfaces remain lightweight, while processing and data management occur server-side.

### d. Connectors and Network Protocols

BrightBoard relies on a combination of well-established web communication protocols and frameworks to ensure reliable, secure, and efficient data exchange between system components. The architecture is intentionally designed around widely adopted technologies to promote compatibility, ease of deployment, and long-term scalability.

At the core of communication between the frontend (React) and the backend (Node.js/Express.js) is the HTTP/HTTPS protocol, with all client-server interactions conducted over HTTPS to ensure encrypted and secure data transmission. HTTPS (Hypertext Transfer Protocol Secure) is essential for protecting sensitive user data, such as login credentials and quiz results, against man-in-the-middle attacks and other forms of interception during transmission.

The application uses a RESTful API architecture, enabling the frontend to interact with backend resources in a stateless manner. Each API endpoint corresponds to specific actions within the system, such as creating courses, submitting quiz responses, or retrieving user progress. RESTful APIs ensure a consistent and scalable interface that allows future enhancements, such as third-party integrations or mobile clients, without major architectural changes.

For authentication and access control, BrightBoard uses JWT (JSON Web Tokens). When a user logs in, the server issues a token that is securely stored on the client side (typically in local storage or cookies). This token is included in the header of subsequent requests, allowing the server to verify the user's identity and role without the need for session persistence. JWT is well-suited for distributed systems and improves performance by avoiding frequent lookups in a session store.

The backend also uses MongoDB, a NoSQL, document-oriented database, to persist application data. Communication with MongoDB is handled through Mongoose, an Object Data Modeling (ODM) library for Node.js that defines schemas, validation rules, and query logic. This structured approach helps maintain data integrity while still benefiting from the flexibility of a NoSQL system.

In terms of connectors, the application stack is composed entirely of web-based protocols and libraries, including:

- **HTTPS** for encrypted client-server communication
- **HTTP REST** endpoints for functional API interaction
- **JWT** for stateless authentication and session management
- **MongoDB** as the database backend, connected via Mongoose ODM

There is no use of low-level socket communication (e.g., TCP sockets or WebSockets), as BrightBoard does not require real-time updates or peer-to-peer messaging. Its event-driven but non-real-time nature makes RESTful HTTP over HTTPS the ideal choice.

This architecture and protocol stack together offer a secure, scalable, and maintainable foundation that aligns well with the needs of a modern, web-based learning management system.

### e. Global Control Flow

BrightBoard operates as an event-driven system, meaning the application responds to user-generated actions such as logging in, enrolling in a course, viewing lessons, or completing quizzes. These actions do not follow a strict sequence; instead, they trigger specific server-side responses and dynamically update the user interface based on the user's role and progress. This interaction model ensures flexibility and a personalized learning experience.

- **Execution Orderliness:**

The flow of execution in BrightBoard is non-linear and entirely driven by user input. For instance, one student might choose to take a quiz immediately after enrolling in a course, while another might prefer to read through all available lessons before attempting any assessments. This flexibility supports self-paced learning and accommodates a variety of learning styles. The system is designed to allow users to navigate content in the order that best fits their needs, rather than enforcing a rigid, predetermined path.

- **Time Dependency:**

BrightBoard is not a real-time system and does not depend on timers or continuous scheduling mechanisms. However, the system does incorporate timestamps on quiz submissions to record the exact date and time each quiz was completed. This information is valuable for both students and instructors, as it provides transparency, helps track progress over time, and may be used for analytics or time-based feedback. Despite this time-aware feature, the overall system remains asynchronous and event-driven, with no hard real-time constraints or requirements for continuous uptime synchronization.

### f. Hardware Requirements

BrightBoard is designed with modest hardware requirements to ensure ease of deployment and broad accessibility for both instructors and students. The system relies on commonly available technology components without the need for specialized hardware or sensors. Below is a breakdown of the essential hardware and system resources:

Client-Side Requirements:

- Screen Display:
  Minimum resolution of 640 × 480 pixels to support responsive interface design and lesson readability.
- Web Browser:
  A modern web browser such as Chrome, Firefox, or Edge capable of rendering dynamic, JavaScript-based user interfaces.
- Communication Network:
  An active internet connection with a recommended minimum bandwidth of 56 Kbps (modern broadband is preferred for optimal performance and loading times).

- Input Devices:
  Standard keyboard and mouse or touchscreen interface for navigating lessons, quizzes, and dashboard interactions.

Server-Side Requirements:

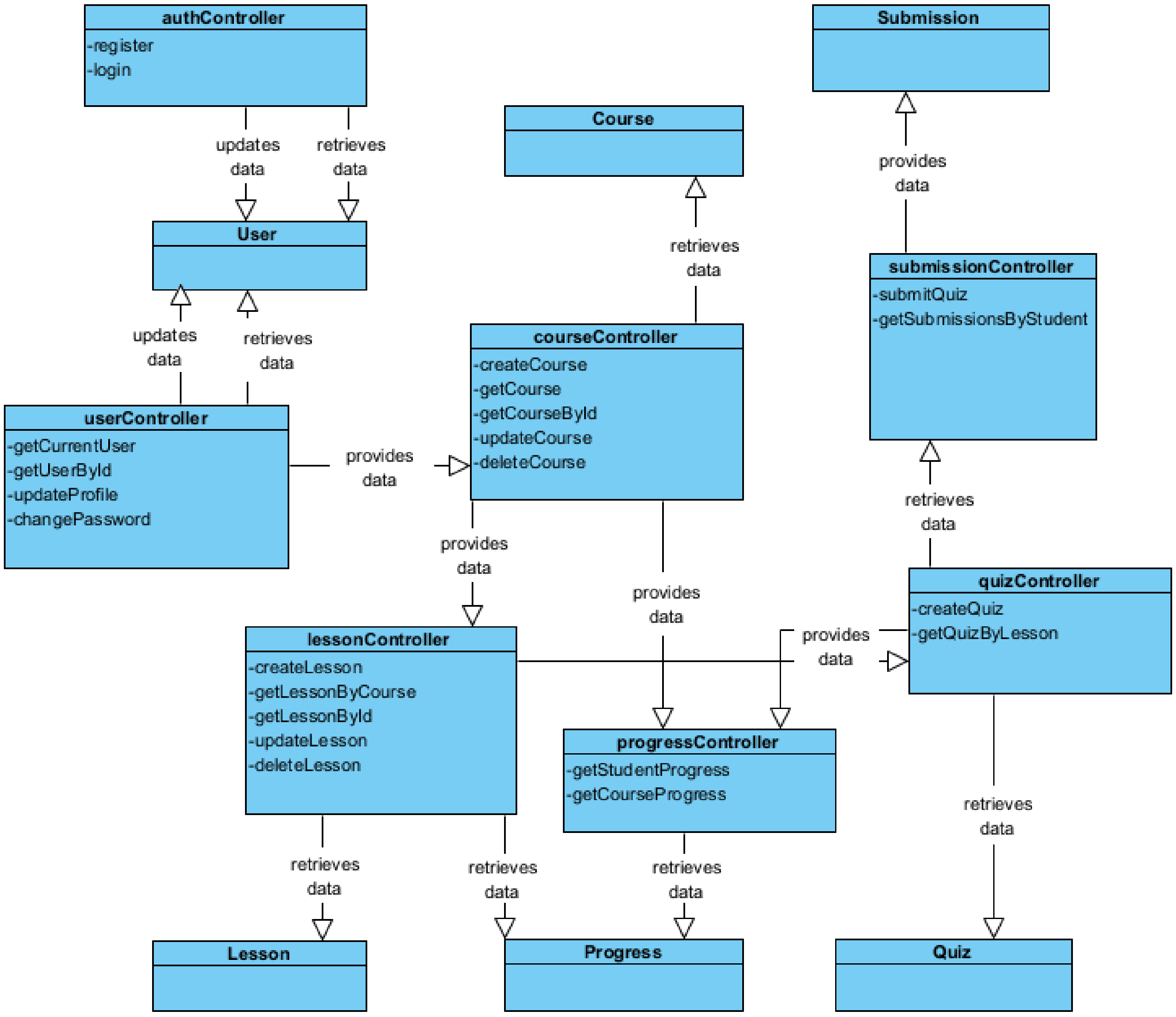
- Processing Environment:
  A hosting environment capable of running a Node.js backend and connecting to a MongoDB database.
- Disk Storage:
  Minimum of 2 GB storage capacity, sufficient for handling uploaded PDF files, user data, and database growth.
- Network Availability:
  Continuous uptime with at least 99.9% availability to ensure user access and minimal downtime.
- Communication Protocols:
  Support for secure HTTPS connections to protect data transmission and comply with privacy requirements.
- Memory & CPU:
  Modest RAM and CPU resources suitable for handling lightweight requests, expected user concurrency, and real-time feedback (e.g., 2 GB RAM and a dual-core processor or cloud equivalent)

## 7. Analysis and Domain Modeling

### a. Conceptual Model

In this part we have designed a conceptual model for our BrightBoard LMS. This is meant to display the structure that our application will operate on. The conceptual model is meant to convey concepts used, the association between them, attributes

**Domain Model**



## i. Concept definitions

| Concept Name | Type (D=Doing, K=Knowing) | Responsibility |
|---|---|---|
| authController | D | Handles user registration, login, token generation, and authentication flows. |
| userController | D | Manages user profile access, including fetching current user and user by ID. |
| User | K | Represents platform users (instructors or students), stores login and role data. |
| courseController | D | Handles course creation, deletion, listing, and fetching by ID. |
| Course | K | Represents a course with title, description, and associated instructor. |
| lessonController | D | Manages creation, listing, and retrieval of lessons tied to courses. |
| Lesson | K | Represents instructional content (text or PDF) linked to a course. |
| quizController | D | Creates and fetches quizzes associated with lessons. |
| Quiz | K | Represents a multiple-choice assessment linked to a lesson. |
| submissionController | D | Handles quiz submission, scoring, and result storage. |
| Submission | K | Handles quiz submission, scoring, and result storage. |
| progressController | D | Tracks student progress through courses, lessons, and quizzes. |
| Progress | K | Tracks student progress through courses, lessons, and quizzes. |

### ii. Association definitions

| Concept Pair | Association Description | Association Name |
|---|---|---|
| User ↔ Course | A User with role instructor can create multiple Courses. | teaches / isTaughtBy |
| Course ↔ Lesson | Each Course contains multiple Lessons. | hasLessons / belongsToCourse |
| Course ↔ Quiz | Quiz is indirectly linked to Course through Lesson. | hasQuizzes / belongsToCourse |
| Lesson ↔ Quiz | Each Lesson can have one associated Quiz. | hasQuiz / forLesson |
| User ↔ Submission | A User (student) creates a Submission when taking a Quiz. | submits / submittedBy |
| Quiz ↔ Submission | Each Submission is linked to one Quiz. | submittedFor / hasSubmissions |
| User ↔ Progress | A User (student) has a Progress tracker per Course. | tracksProgress / belongsToStudent |
| Progress ↔ Lesson | Progress tracks Lessons completed by a student in a course. | lessonsCompleted |
| Progress ↔ Quiz | Progress tracks Quiz completions per student per course. | quizzesCompleted |
| courseController ↔ Course | courseController performs create, fetch, and update actions on Course. | manages |
| lessonController ↔ Lesson | lessonController handles creation and retrieval of Lessons. | manages |
| quizController ↔ Quiz | quizController handles creation and fetching of Quiz data. | manages |
| submissionController ↔ Submission | submissionController handles quiz submission and storage of Submission. | manages |

|  |  |  |
|---|---|---|
| progressController ↔ Progress | progressController retrieves or updates Progress based on user/course actions. | manages |
| authController ↔ User | authController handles registration, login, and authentication of User. | authenticates |
| userController ↔ User | userController provides access to user profile and role information. | managesProfile |
| User ↔ Course | A User with role instructor can create multiple Courses. | teaches / isTaughtBy |

### iii. Attribute definitions

| Concept | Attributes | Attribute Description |
|---|---|---|
| courseController | createCourse | Creates a new course by an instructor. |
|  | getCourse | Retrieves a list of all courses available. |
|  | getCourseById | Fetches detailed information for a single course using its ID. |
|  | updateCourse | Updates course title or description. |
|  | deleteCourse | Deletes a course (typically by instructor or admin). |
| lessonController | createLesson | Adds a new lesson (text or PDF) under a specific course. |
|  | getLessonByCourse | Retrieves all lessons linked to a specific course. |
|  | getLessonById | Fetches a specific lesson using its unique ID. |
|  | updateLesson | Updates content or metadata of a lesson. |
|  | deleteLesson | Deletes a specific lesson. |

| | | |
|---|---|---|
| progressController | getStudentProgress | Retrieves quiz/lesson completion data for a specific student. |
| | getCourseProgress | Shows aggregated progress data for a course's students. |
| quizController | createQuiz | Creates a quiz for a specific lesson. |
| | getQuizByLesson | Retrieves the quiz linked to a particular lesson. |
| submissionController | submitQuiz | Submits a student's answers, calculates score, and stores the submission. |
| | getSubmissionByStudent | Retrieves all submissions made by a specific student. |
| userController | getCurrentUser | Fetches data of the currently authenticated user. |
| | getUserById | Returns profile data for a user by ID. |
| | updateProfile | Allows users to update their name, email, or role. |
| | changePassword | Lets users securely change their password. |
| authController | register | Registers a new user with hashed password and assigned role. |
| | login | Authenticates user credentials and returns a JWT token. |

### iv. Traceability matrix (2) - Use Cases to Domain Concept Objects

| **Domain Concepts** | **Use Case** | **UC1** | **UC2** | **UC3** | **UC4** | **UC5** | **UC6** | **UC7** | **UC8** |
|---|---|---|---|---|---|---|---|---|---|
| | **PW** | 25 | 15 | 9 | 13 | 11 | 14 | 8 | 5 |
| authController | | X | | | | | | | |
| userController | | X | | | | X | | | |
| User | | X | X | | | | X | X | X |
| courseController | | | X | | | | | | X |
| Course | | | X | X | X | X | X | | |
| lessonController | | | | X | | | X | | |
| Lesson | | | | X | X | | X | | X |
| quizController | | | | | X | | | | |
| Quiz | | | | | X | | | X | X |
| submissionController | | | | | | | | X | |
| Submission | | | | | | | | X | X |
| progressController | | | | | | | X | X | X |
| Progress | | | | | | X | X | X | X |

### b. System Operation Contracts

**UC-1:**

- *Preconditions:* User is not logged in.
- *Postconditions:* User is authenticated.
- *Flow:* Submit credentials ~ Validate ~ Redirect based on role.
- *Extensions:* Invalid login or email already used ~ Error prompts.

| Use Case ID | Use Case Name | Contract | Inputs | Outputs | Exceptions |
|---|---|---|---|---|---|
| **UC-1** | Register/Login | When a user enters login or registration details, the system validates the information and authenticates them. | Email,password, role:(on registration) | User is logged in and redirected to their dashboard | If credentials are invalid or already registered, show an error message. |

**UC-2:**

- *Preconditions:* Instructor must be logged in.
- *Postconditions:* Course list is modified.
- *Flow:* Create/Edit/Delete course ~ Save changes ~ Update dashboard.
- *Extensions:* Update or delete fails ~ Error prompt shown.

| Use Case ID | Use Case Name | Contract | Inputs | Outputs | Exceptions |
|---|---|---|---|---|---|
| **UC-2** | Course Management | When an instructor creates, edits, or deletes a course, the system processes the changes and updates the course list. | Course title, description, instructor ID | Course is created, updated, or deleted accordingly | If course data is invalid or save/delete fails, display an error message. |

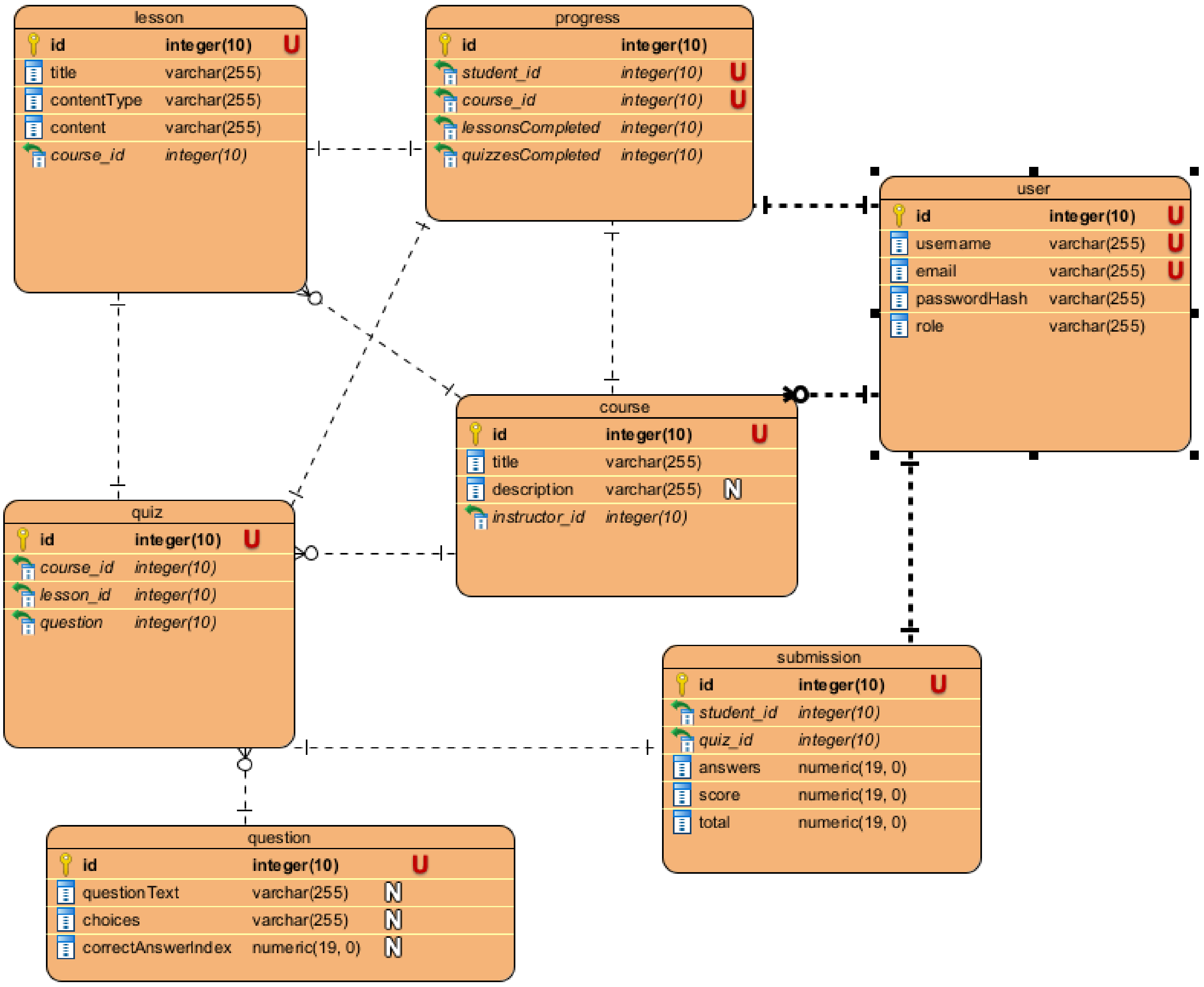
## c. Data Model and Persistent Data Storage

BrightBoard requires persistent data storage, as it supports long-term user interactions, course management, quiz tracking, and progress analytics.

The application uses a document-oriented NoSQL database management system to store user profiles, course data, lesson content, quiz information, and student progress. We selected MongoDB as our database solution because it integrates seamlessly with our backend (built on Node.js with Express) and supports flexible, schema-based modeling via Mongoose, an ODM tool. MongoDB allows us to efficiently handle the hierarchical and nested data structures inherent in educational content, such as lessons within courses and questions within quizzes.

Given that BrightBoard is designed to be lightweight and scalable, MongoDB's ability to scale horizontally and accommodate varying document structures made it a natural fit. Additionally, our team has significant prior experience with JavaScript and JSON-based data modeling, which made MongoDB both a practical and efficient choice. Data is organized across a focused set of collections that mirror the system’s core entities: users, courses, lessons, quizzes, and progress records.

## Database Schema



**User Table**

The user table, which stores information for all platform users, including both students and instructors. Each user has a unique ID, username, email, encrypted password (passwordHash), and a role that designates their permission level. This structure allows BrightBoard to enforce role-based access and navigation, enabling different capabilities depending on whether the user is an instructor or a student.

- Instructors are associated with courses they create through a one-to-many relationship from the user table to the course table via the instructor_id foreign key. Students, meanwhile, are linked to their quiz submissions and progress through foreign keys in the submission and progress tables, respectively.

**Course Table**

The course table represents instructional units created by instructors. It includes a title, a description, and an instructor_id that references the creator. Each course may contain multiple lessons and quizzes, which are managed through one-to-many relationships with the lesson and quiz tables.

**Lesson Table**

Lessons are the instructional materials associated with courses. Each entry in the lesson table is linked to a specific course via course_id and contains metadata such as a title, contentType (e.g., text or PDF), and the actual content. Lessons may also serve as containers for quizzes, supporting structured progression through course material.

**Quiz and Question Tables**

The quiz table stores assessment modules used to evaluate students' understanding. Each quiz is tied to both a course and a lesson using foreign keys. Though the current structure uses a question field within the quiz, it's complemented by the question table, which holds reusable question entries. Each question includes a prompt (questionText), multiple choice choices, and the correctAnswerIndex.

- This arrangement allows quizzes to be composed of multiple questions, promoting flexibility in quiz creation and question reuse. The implied many-to-many relationship between quiz and question supports dynamic quiz building.

**Submission Table**

Student interactions with quizzes are captured in the submission table. Each submission links a student_id and a quiz_id, recording the student's answers, the calculated score, and the total possible points. This structure supports detailed analytics and feedback on student performance.

**Progress Table**

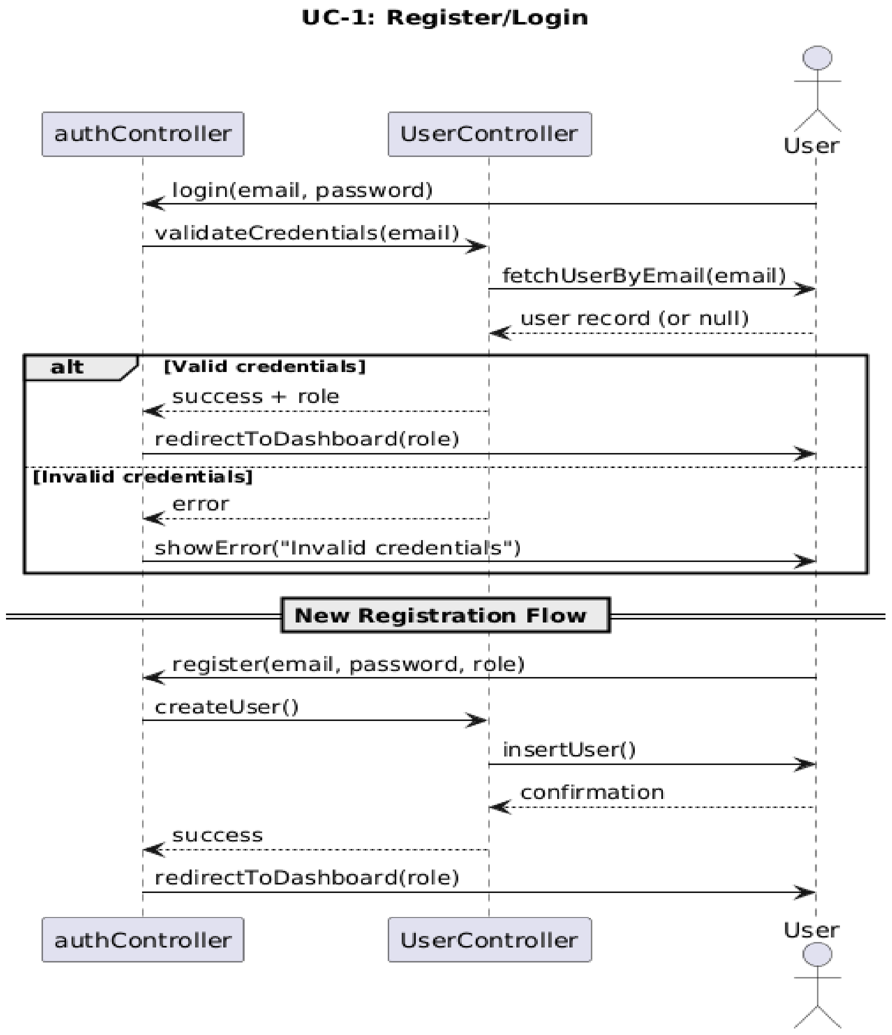
The progress table tracks each student's advancement through individual courses. It uses a composite key of student_id and course_id to uniquely identify records. For each course a student is enrolled in, the system tracks how many lessonsCompleted and quizzesCompleted, providing the foundation for progress bars and  completion percentages.

**Entity Relationships Summary**

- Users (instructors) create multiple courses
- Each course contains many lessons and quizzes
- Lessons may be linked to one or more quizzes
- Quizzes are composed of multiple questions
- Students (users) complete submissions and accumulate progress through courses

## 8. Interaction Diagrams

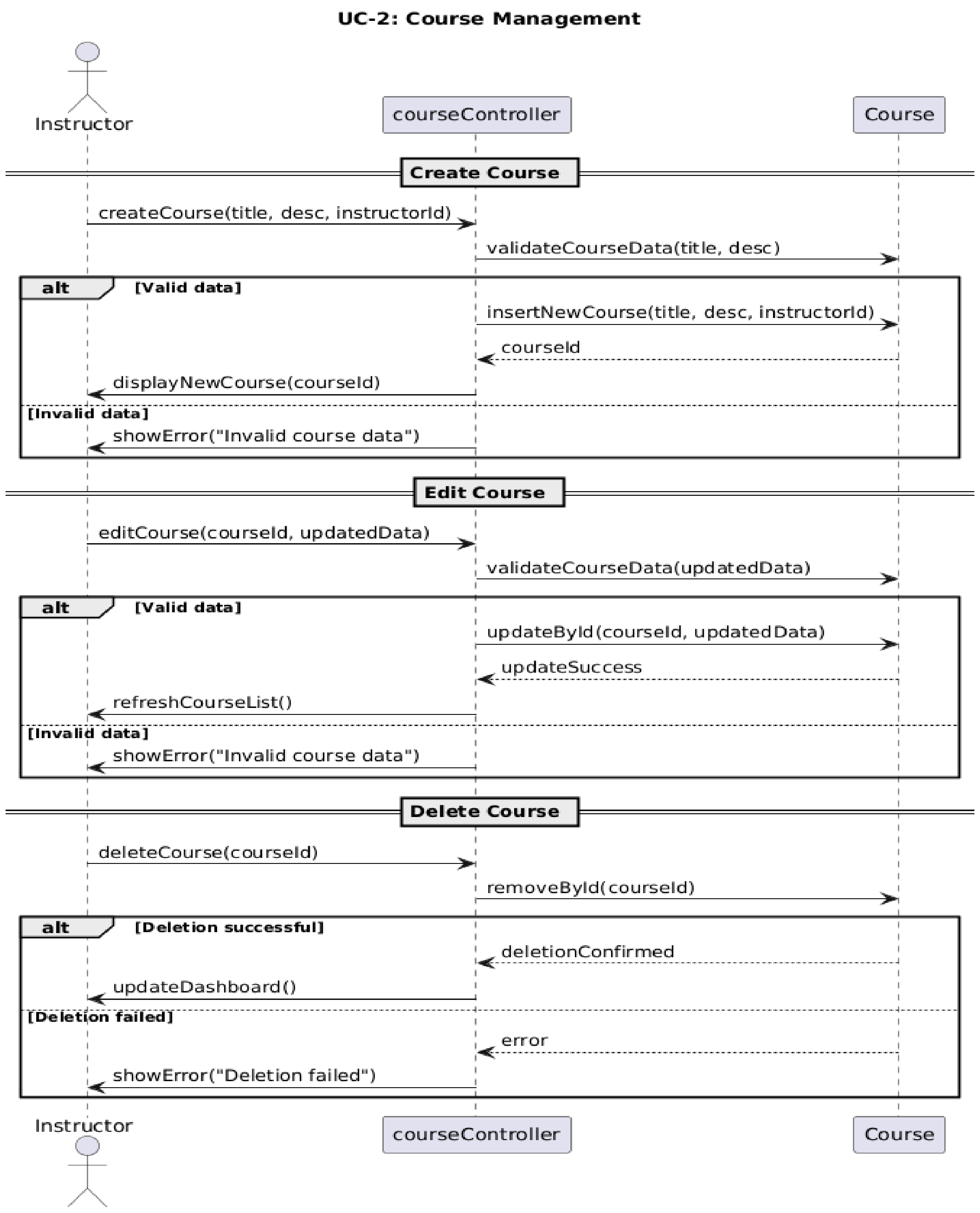
---



In the Register/Login use case, responsibilities are assigned using the Single Responsibility Principle: authController handles authentication logic, while UserController manages user data operations. This separation supports modularity and independent evolution of each component.

The Information Expert principle places credential validation and user creation within UserController, which directly accesses the User model. To maintain low coupling and high cohesion, authController delegates tasks without manipulating user data directly.

Applying the Controller Pattern, authController acts as a coordinator between user requests and domain logic. Finally, following the Law of Demeter, each controller only interacts with objects it directly depends on—ensuring encapsulated, maintainable code.


UC-2: Course Management
Instructor
courseController
Course
Create Course
createCourse(title, desc, instructorId)
validateCourseData(title, desc)
alt [Valid data]
insertNewCourse(title, desc, instructorId)
courseId
displayNewCourse(courseId)
[Invalid data]
showError("Invalid course data")
Edit Course
editCourse(courseId, updatedData)
validateCourseData(updatedData)
alt [Valid data]
updateById(courseId, updatedData)
updateSuccess
refreshCourseList()
[Invalid data]
showError("Invalid course data")
Delete Course
deleteCourse(courseId)
removeById(courseId)
alt [Deletion successful]
deletionConfirmed
updateDashboard()
[Deletion failed]
error
showError("Deletion failed")
Instructor
courseController
Course

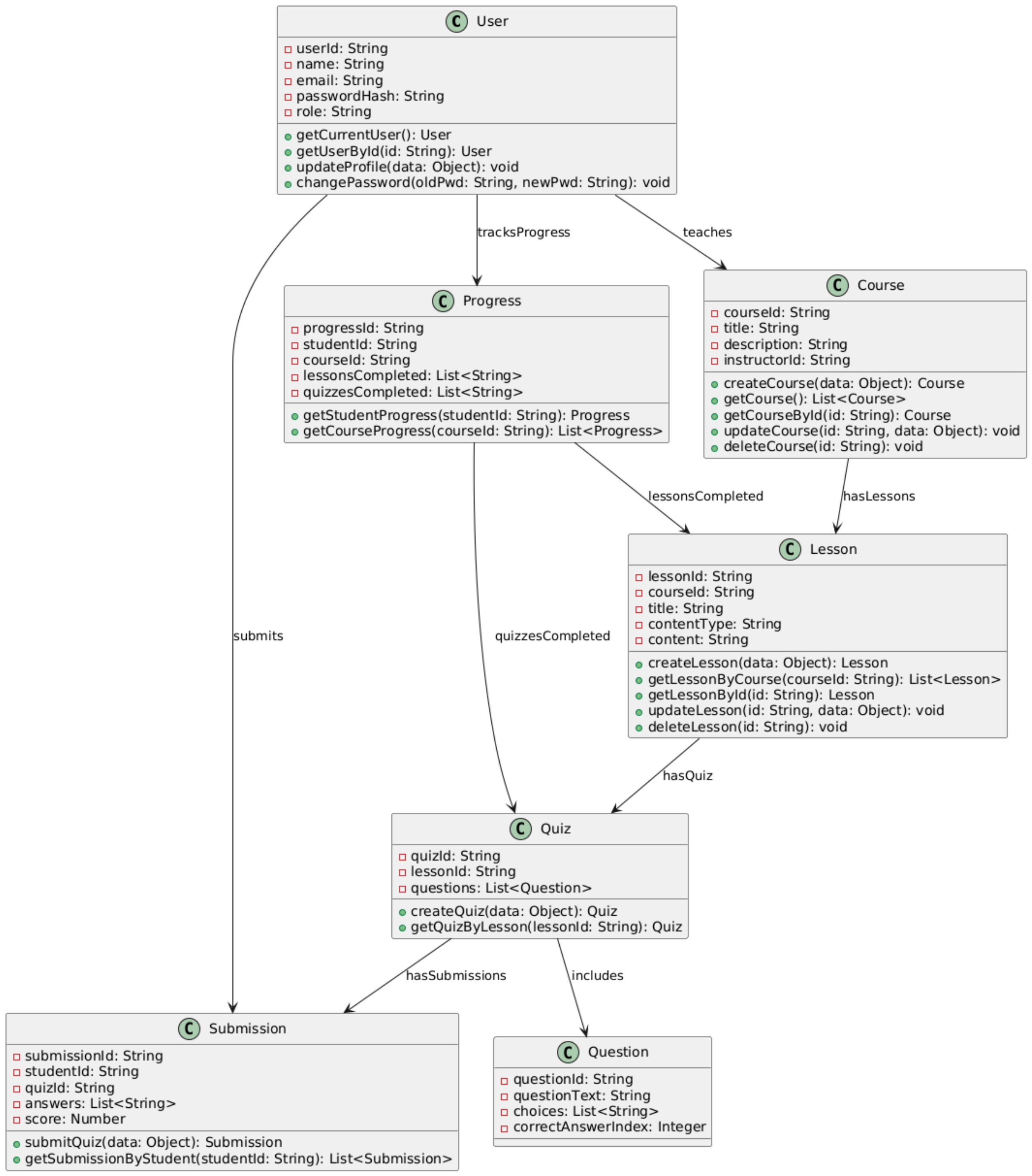
In Course Management, the Single Responsibility Principle guides the design by assigning courseController to handle user interactions and Course to manage data and business logic. This separation improves focus and simplifies maintenance.

Following Information Expert, the Course model owns operations like create, update, and delete since it contains the necessary data. Low coupling and high cohesion are achieved by having the controller delegate all data tasks to the model.

Using the Controller Pattern, courseController orchestrates workflows without containing domain logic. The Law of Demeter ensures each controller communicates only with its immediate collaborators, reinforcing modularity and minimizing dependencies.

## 9. Class Diagram and Interface Specification

### a. Class Diagram


User
userId: String
name: String
email: String
passwordHash: String
role: String
getCurrentUser(): User
getUserById(id: String): User
updateProfile(data: Object): void
changePassword(oldPwd: String, newPwd: String): void
tracksProgress
teaches
Progress
progressId: String
studentId: String
courseId: String
lessonsCompleted: List<String>
quizzesCompleted: List<String>
getStudentProgress(studentId: String): Progress
getCourseProgress(courseId: String): List<Progress>
Course
courseId: String
title: String
description: String
instructorId: String
createCourse(data: Object): Course
getCourse(): List<Course>
getCourseById(id: String): Course
updateCourse(id: String, data: Object): void
deleteCourse(id: String): void
lessonsCompleted
hasLessons
Lesson
lessonId: String
courseId: String
title: String
contentType: String
content: String
createLesson(data: Object): Lesson
getLessonByCourse(courseId: String): List<Lesson>
getLessonById(id: String): Lesson
updateLesson(id: String, data: Object): void
deleteLesson(id: String): void
submits
quizzesCompleted
hasQuiz
Quiz
quizId: String
lessonId: String
questions: List<Question>
createQuiz(data: Object): Quiz
getQuizByLesson(lessonId: String): Quiz
hasSubmissions
includes
Submission
submissionId: String
studentId: String
quizId: String
answers: List<String>
score: Number
submitQuiz(data: Object): Submission
getSubmissionByStudent(studentId: String): List<Submission>
Question
questionId: String
questionText: String
choices: List<String>
correctAnswerIndex: Integer

### b. Data Types and Operation Signatures

1.User

| **Class: User** |
|---|
| Attributes:<br>+ userId: String<br>+ name: String<br>+ email: String<br>+ passwordHash: String<br>+ role: String // e.g., "student" or "instructor" |
| Operations:<br>+ getCurrentUser(): User<br>+ getUserById(id: String): User<br>+ updateProfile(data: Object): void<br>+ changePassword(oldPwd: String, newPwd: String): void |

User Table Definitions:

- Represents a person using BrightBoard.
- userId: Unique ID for each user.
- name, email: Personal details.
- passwordHash: Securely stored password.
- role: Defines whether a user is a student or instructor.
- getCurrentUser(): Returns the user who is currently logged in.
- getUserById(id): Finds a user using their ID.
- updateProfile(data): Allows user to change their info.
- changePassword(oldPwd, newPwd): User can change password securely.

2.Course

| Course |
|---|
| Attributes:<br>+ courseId: String<br>+ title: String<br>+ description: String<br>+ instructorId: String |
| Operations:<br>+ createCourse(data: Object): Course<br>+ getCourse(): List<Course><br>+ getCourseById(id: String): Course<br>+ updateCourse(id: String, data: Object): void<br>+ deleteCourse(id: String): void |

Course Table Definitions:

- A digital learning module created by an instructor.
- courseId: Unique ID for the course.
- title, description: Course name and summary.
- instructorId: The user who created the course.
- createCourse(data): Instructor creates a course.
- getCourse(): Get all available courses.
- getCourseById(id): Fetch specific course.
- updateCourse(id, data): Edit course info.
- deleteCourse(id): Remove a course from the system.

3. Lesson

| Class Lesson |
|---|
| Attributes:<br>+ lessonId: String<br>+ courseId: String<br>+ title: String<br>+ contentType: String     // e.g., "text", "video"<br>+ content: String         // URL or text body |
| Operations:<br>+ createLesson(data: Object): Lesson<br>+ getLessonByCourse(courseId: String): List<Lesson><br>+ getLessonById(id: String): Lesson<br>+ updateLesson(id: String, data: Object): void<br>+ deleteLesson(id: String): void |

Lesson Table Definitions:

- An individual unit or topic within a course.
- lessonId: Unique ID for the lesson.
- courseId: Which course it belongs to.
- title: Lesson title.
- contentType: Format, like text or video.
- content: Actual lesson content or link.
- createLesson(), updateLesson(): For managing lesson content.
- getLessonByCourse(): Fetch all lessons in a course.
- deleteLesson(): Remove a lesson.

4. Quiz

| Class Quiz |
|---|
| Attributes:<br>+ quizId: String<br>+ lessonId: String<br>+ questions: List<Question> |
| Operations:<br>+ createQuiz(data: Object): Quiz<br>+ getQuizByLesson(lessonId: String): Quiz |

Quiz Table Definitions:

- An assessment tied to a lesson.
- quizId: Unique ID.
- lessonId: Which lesson it tests.
- questions: List of questions included.
- createQuiz(): Add a new quiz to a lesson.
- getQuizByLesson(): Retrieve quiz tied to a lesson.

5. Question

| Class Question |
|---|
| Attributes:<br>+ questionId: String<br>+ questionText: String<br>+ choices: List<String><br>+ correctAnswerIndex: Integer |
| Operations: |

Question Table Definitions:

- A multiple-choice question.
- questionId: Unique ID.
- questionText: The actual question.
- choices: The answer options.
- correctAnswerIndex: Which option is correct (e.g., 0 = first choice).

6. Submission

| Class Submission |
|---|
| Attributes:<br>+ submissionId: String<br>+ studentId: String<br>+ quizId: String<br>+ answers: List<String><br>+ score: Number |
| Operations:<br>+ submitQuiz(data: Object): Submission<br>+ getSubmissionByStudent(studentId: String): List<Submission> |

Submission Table Definitions:

- A student's answer to a quiz.
- submissionId: Unique ID.
- studentId: Who submitted.
- quizId: Which quiz it relates to.
- answers: Student's selected answers.
- score: Grade assigned after submission.
- submitQuiz(): Save the student's answers.
- getSubmissionByStudent(): Review past attempts.

7. Progress

| Class Progress |
|---|
| Attributes:<br>+ progressId: String<br>+ studentId: String<br>+ courseId: String<br>+ lessonsCompleted: List<String>     // lessonIds<br>+ quizzesCompleted: List<String>     // quizIds |
| Operations:<br>+ getStudentProgress(studentId: String): Progress<br>+ getCourseProgress(courseId: String): List<Progress> |

Progress  Table Definitions:

- Tracks what a student has finished.
- progressId: Unique ID.
- studentId, courseId: Who and what course.
- lessonsCompleted: List of lessons finished.
- quizzesCompleted: List of completed quizzes.
- getStudentProgress(): Overview for a user.
- getCourseProgress(): See how all students are doing.

## c. Traceability Matrix (3) - Domain Concept Objects to Class Objects

Some of our domain concepts did not directly translate into derived classes because they represent the logic and behavioral flow of the application rather than concrete, persistent entities. These controller-based concepts (e.g., authController, userController) are responsible for handling interactions between users and the system or coordinating data operations, but they do not represent real-world objects or require data storage. As such, no data classes were derived from them.

Below is a summary of how each domain concept maps to one or more classes and why:

| Domain Concept | Class(es) Derived | Explanation |
|---|---|---|
| authController | | Handles user authentication logic like registration and login using JWT, not tied to a data class. |
| userController | | Manages operations related to retrieving and modifying user information; interacts with the User class. |
| User | User | Core class representing users of the system (students and instructors), storing credentials and roles. |
| courseController | | Manages creation, listing, updating, and deletion of courses; operates on the Course class |
| Course | Course | Represents a structured learning module containing lessons and created by instructors. |
| lessonController | | Controls operations for creating and accessing lessons tied to specific courses |
| Lesson | Lesson | Represents individual learning units containing text or PDF content, associated with a course. |
| quizController | | Facilitates creation and retrieval of quizzes tied to lessons; operates on the Quiz class. |
| Quiz | Quiz, Question | Quiz models an assessment with multiple questions; Question defines the structure of each quiz item. |
| submissionController | | Handles quiz submissions, scoring, and stores results; updates student progress based on submissions. |
| Submission | Submission | Captures student quiz responses and calculated scores for performance tracking. |
| progressController | | Tracks how much of a course a student has completed, using lesson and quiz completion data. |
| Progress | Progress | Records completed lessons and quizzes for each student-course pair, supporting progress visualization. |

# 10. Algorithms and Data Structures

---

### a. Data Structures

Brightboard utilizes the following data structures:

Arrays (Lists)

- Storing multiple items such as:
    - Lessons in a Course
    - Questions in a Quiz
    - Choices in a Question
    - Answers in a Submission
    - Completed lessons and quizzes in Progress

Arrays are utilized due to their effectiveness in Ordered data: Lessons and questions often need to be accessed in a specific sequence. Performance: Arrays offer O(1) access by index and efficient iteration. Simplicity: Easy to use and serialize in JSON (which the system uses for API communication).

Hash Tables (Objects or Maps)

- User sessions or tokens: mapping user IDs to session data.
- Fast lookup of:
    - Users by ID
    - Courses by ID
    - Lessons or quizzes by ID

Hash Tables are utilized due to Fast access (average O(1) time complexity).Flexibility: Easy to grow or modify key-value pairs without reordering data. Scalability: Efficient even as data grows.

BrightBoard doesn't rely on more advanced structures like trees or linked lists, but it strategically uses arrays and hash tables to maintain performance, simplicity, and data organization particularly for features like course content delivery, quiz systems, and user tracking.

**b. Concurrency:**

BrightBoard does not use multiple threads explicitly.

BrightBoard is built using Node.js, which is single-threaded by design (based on the event-driven, non-blocking I/O model). Instead of spawning multiple threads, Node.js uses an event loop and asynchronous programming to handle concurrent operations like:

- API request handling
- Database queries
- File uploads
- Authentication flows
- Quiz submissions and scoring

In place of multithreading brightboard utilizes:

- **Asynchronous Callbacks / Promises / async-await:**
  These allow operations (e.g., reading a file or fetching from MongoDB) to run without blocking the main thread.

- **Express.js Route Handlers:**
  Each incoming request is handled independently and non-blocking. While it seems like parallel processing, it's actually asynchronous execution within a single thread.

There is no need for thread synchronization mechanisms (like mutexes, semaphores, or locks), because Node.js ensures that only one operation runs at a time in the main thread, while background tasks complete asynchronously.

If in the future BrightBoard needs to handle CPU-intensive tasks (e.g., real-time analytics or media processing), it could benefit from Worker Threads (Node.js module), Child Processes, Message queues (e.g., RabbitMQ, Redis) but those are not currently used.

## 11. User Interface Design and Implementation

No significant changes have been made to the initial screen mock-ups developed for Report #1, nor are any major revisions currently planned. Our original interface design emphasized clarity, simplicity, and minimal user effort. We have continued to follow those principles throughout development.

The existing layout supports intuitive navigation, with clear labeling and minimal required input per screen. By maintaining a clean and structured interface, we aim to reduce user confusion and eliminate unnecessary steps in common workflows (such as logging in, accessing content, and tracking progress). Minor visual updates may occur during implementation, but these will not affect usability or introduce additional complexity.

Overall, our design remains focused on maximizing ease-of-use by minimizing user actions and promoting intuitive interaction, in line with best practices for effective GUI design.

## 12. Test Designs

---

Note that for this report you are just *designing* your tests; you will *program and run* those tests as part of work for your first demo, see the list here.

### a. Test Cases

**Test Case List and Descriptions**

| | |
|---|---|
| Test Case Identifier: TC-1<br><br>**Input Data:** Email, Password<br><br>**Use Case Tested:** UC-2: Log in / Log out<br>**Pass/Fail Criteria:**<br>● **Pass:** If the user is able to successfully log in with valid credentials.<br>● **Fail:** If the user enters an invalid email or password. | |
| **Test Procedure:** | **Step 1:** User submits quiz with missing or malformed answers.<br><br>**Step 2:** User submits quiz with all required answers correctly formatted. |
| **Expected Result:** | ● Server rejects the submission and notifies the user to complete the quiz.<br>● Server accepts submission, evaluates the score, and returns the result. |

|  |  |
|---|---|
| Test Case Identifier: TC-2<br><br>**Use Case Tested:** UC-6: Take Quiz<br>**Pass/Fail Criteria:**<br>● **Pass:** If the user is able to submit a completed quiz and receive a score.<br>● **Fail:** If submission is rejected or scoring fails.<br><br>**Input Data:** Quiz ID, Question Answers | |
| **Test Procedure:** | **Step 1:** User enters an invalid email or password.<br><br>**Step 2:** User enters a valid email and password. |
| **Expected Result:** | ● Server denies the login attempt with a message indicating incorrect credentials.<br>● Server allows login, and depending on the user role (student or instructor), redirects to the correct dashboard. |

| Test Case Identifier: TC-3<br><br>**Use Case Tested:** UC-4: Create Course (Instructor)<br>**Pass/Fail Criteria:**<br>● **Pass:** If a course is created with a unique name and valid data.<br>● **Fail:** If required fields are missing or course already exists.<br><br>**Input Data:** Course title, description, instructor ID | |
|---|---|
| **Test Procedure:** | **Step 1**: Instructor attempts to create a course with missing title.<br><br>**Step 2**: Instructor submits course with valid data. |
| **Expected Result:** | ● Server denies creation and displays error message.<br><br>● Server creates and stores the course. |

| | |
|---|---|
| Test Case Identifier: TC-4<br><br>**Use Case Tested:** UC-7: Track Progress<br>**Pass/Fail Criteria:**<br>● **Pass:** If the progress data is correctly calculated and shown.<br>● **Fail:** If progress is not updated after completing activities.<br><br>**Input Data:** User ID, lesson/quiz completion data | |
| **Test Procedure:** | **Step 1:** User completes a quiz.<br><br>**Step 2:** User views progress tracker. |
| **Expected Result:** | ● Server updates progress and reflects changes in dashboard.<br>● If error, no update is shown and user is informed.<br>● Server creates and stores the course. |

| Test Case Identifier: TC-5<br><br>**Use Case Tested:** UC-3: Lesson View and Completion<br>**Pass/Fail Criteria:**<br>● **Pass:** If user can access lesson content and mark it as complete.<br>● **Fail:** If content fails to load or completion is not recorded.<br><br>**Input Data:** Lesson ID, user token | |
|---|---|
| **Test Procedure:** | **Step 1:** User opens a lesson that doesn't exist.<br><br>**Step 2:** User completes a valid lesson and marks it complete. |
| **Expected Result:** | ● Error message shown for missing lesson.<br>● Lesson marked complete, status updated.<br>● If error, no update is shown and user is informed.<br>● Server creates and stores the course. |

### b. Test Coverage

Our test cases currently cover core system functionality, including authentication and access control (TC-1), key user actions such as taking quizzes, creating courses, and tracking learning progress (TC-2 through TC-5), as well as validation on both the client and server sides. We ensure differentiated support for user roles(students and instructors)with each role tested for proper access and behavior.

We follow a bottom-up testing strategy, starting with individual backend components like controller logic (e.g., register, login, course creation, submission scoring), input validation, authentication/authorization middleware, and data models. As development progresses, test coverage will expand to include more advanced features such as certification generation, reminders, and analytics dashboards.

We aim to achieve at least 85% unit test coverage across all backend modules. Specific metrics tracked include:

- Line coverage
- Branch coverage
- Function coverage
- Statement coverage

Any code areas falling below 80% coverage will be flagged and revisited to maintain overall quality and reliability.

### c. Integration Testing

We use the bottom-up strategy First, test low-level modules: API endpoints, models, and individual features (quizzes, lessons). Then, test how components work together: submitting quizzes updates progress, logging in redirects based on roles, instructors see aggregated data from students.

This layered approach helps isolate bugs quickly and validate each component in the real-world context of user behavior.

Approach:

- Use Supertest to simulate HTTP requests across controller endpoints
- Use a MongoDB test instance with seeded mock data
- Focus areas:
    - User registration + course creation
    - Lesson creation + quiz linkage
    - Quiz submission + progress update
    - Authenticated routes and access control logic

Planned Integration Tests:

- POST /register → GET /me → POST /courses (registration-to-course flow)
- POST /lessons → POST /quizzes → POST /submitQuiz (learning-to-assessment flow)
- POST /submitQuiz → GET /progress (assessment-to-progress tracking)

### d. System Testing

In system testing, we will validate the system as a complete, integrated solution to ensure that all components function correctly and meet both the functional and non-functional requirements of the platform.

We will begin by validating the core algorithms that drive user interaction and content progression. This includes verifying that quiz and assignment submissions are scored accurately according to predefined rubrics or answer keys. We will also test for edge cases, such as partial submissions, retry limits, and randomized question order. Lesson completion checks will be evaluated to ensure that a lesson is marked complete only when all required actions—such as viewing embedded media or answering questions—are fulfilled. Additionally, we will test the adaptive content rendering functionality to confirm that the platform adjusts learning materials appropriately based on user performance and progress, such as unlocking the next module only when a minimum score threshold has been achieved.

We will also assess the system's non-functional requirements to confirm its robustness in real-world usage. Performance testing will include evaluating how the system responds under high load conditions, such as multiple simultaneous logins or bulk quiz submissions. We will measure latency during key actions like page transitions, content loading, and dashboard rendering to ensure a smooth user experience.

Reliability testing will involve running long-duration tests to detect potential memory leaks, improper session handling, or data persistence failures. We will simulate failover scenarios, including network interruptions and database unavailability, and verify the system's ability to recover gracefully without data loss. Security will be a major focus as well. We will confirm that all authentication and authorization mechanisms are properly enforced across the platform, preventing unauthorized access. Common web vulnerabilities such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF) will be tested against, and we will review the secure handling of sensitive data, including the use of HTTPS and proper password hashing.

The user interface will undergo thorough testing to verify usability and accessibility across all roles. We will validate that dashboards render correctly for students, instructors, and administrators, with each role having access only to its relevant features and functionality. Role-specific actions such as course creation or grade submission will be tested to confirm that they are visible and usable only by authorized users.

We will also review the effectiveness of error messages, ensuring they are clear, informative, and helpful to the user. This includes testing how the system responds to invalid input, failed submissions, or expired sessions. All user-interactive features such as links, buttons, menus, and form controls will be tested to confirm their responsiveness, functionality, and visual feedback under normal and edge-case interactions. We will further ensure the interface is fully responsive and accessible on various device types, including desktops, tablets, and mobile phones. Lastly, we will evaluate the intuitiveness of navigation flows, verifying that users can move through the system logically and efficiently, whether returning to the dashboard, editing profiles, or accessing learning progress.

## 13. Project Management

### a. Plan of Work

| Task | Developer | Est. Begin | Est.Complete | Done |
|---|---|---|---|---|
| Set up GitHub repo & task board | Both | 6/7/2025 | 6/8/2025 | X |
| Define tech stack | Both | 6/7/2025 | 6/8/2025 | X |
| Design DB schema (users, courses, etc.) | Conner | 6/7/2025 | 6/10/2025 | X |
| Set up backend project (Express.js) | Conner | 6/7/2025 | 6/10/2025 | X |
| Wireframe UI: login, dashboard, courses | Shadow | 6/7/2025 | 6/10/2025 | X |
| Setup frontend scaffolding (React, CSS) | Shadow | 6/9/2025 | 6/11/2025 | X |
| Implement auth (login, register) | Conner | 6/12/2025 | 6/15/2025 | X |
| Role-based access (Instructor/Student) | Conner | 6/12/2025 | 6/15/2025 | X |
| Build login/register UI | Shadow | 6/12/2025 | 6/14/2025 | X |
| Responsive styling + validation | Shadow | 6/13/2025 | 6/15/2025 | X |
| Create course + lesson API | Conner | 6/16/2025 | 6/19/2025 | X |
| Instructor dashboard UI | Shadow | 6/16/2025 | 6/19/2025 | X |
| Lesson upload + preview | Shadow | 6/18/2025 | 6/20/2025 | X |
| Student course listing + enroll API | Conner | 6/23/2025 | 6/25/2025 | |
| Serve lessons securely (student view) | Conner | 6/24/2025 | 6/26/2025 | |
| Student dashboard UI (browse/enroll/view) | Shadow | 6/23/2025 | 6/26/2025 | |
| Add mobile responsiveness | Shadow | 6/25/2025 | 6/27/2025 | |
| Quiz creation API (MCQs) | Conner | 6/30/2025 | 7/2/2025 | |
| Quiz-taking + feedback API | Conner | 7/2/2025 | 7/4/2025 | |
| Quiz builder UI (MCQs) | Shadow | 6/30/2025 | 7/2/2025 | |
| Quiz interface + feedback design | Shadow | 7/2/2025 | 7/4/2025 | |
| Progress tracking API | Conner | 7/7/2025 | 7/9/2025 | |
| Visual progress UI (bars, badges) | Shadow | 7/7/2025 | 7/9/2025 | |
| UI polish + accessibility | Shadow | 7/9/2025 | 7/11/2025 | |

| | | | | |
|---|---|---|---|---|
| User docs + test cases | Shadow | 7/9/2025 | 7/11/2025 | |
| Final testing + bug fixes | Both | 7/10/2025 | 7/11/2025 | |
| Deploy app to server | Both | 7/11/2025 | 7/12/2025 | |
| Prepare final demo & presentation | Both | 7/14/2025 | 7/18/2025 | |

## References

Almrashdeh, I. A., et al. "Distance Learning Management System Requirements from Student’s Perspective." *Journal of Theoretical and Applied Information Technology*, vol. 24, no. 1, 2011, pp. 17–27.

Frand, Jason L. "The Information Age Mindset: Changes in Students and Implications for Higher Education." *Educause Review*, vol. 35, no. 5, 2000, pp. 14–24.

Rhode, Jason, et al. "Understanding Faculty Use of the Learning Management System." *Online Learning*, vol. 21, no. 3, 2017, pp. 68–86. https://doi.org/10.24059/olj.v21i3.1217.

Turnbull, Deborah, Rajeev Chugh, and Jo Luck. "Learning Management Systems, An Overview." *Encyclopedia of Education and Information Technologies*, edited by Arthur Tatnall, Springer, 2020. https://doi.org/10.1007/978-3-030-10576-1_248.