1 Development View

1.1 Module Organization

The EduPal quiz integration feature is organized into several interconnected modules, each responsible for specific aspects of the system:

1.1.1 User Interface Module

- **QuizPage**: The main page where users can start the quiz.
- **QuestionPage**: The page that displays quiz questions to the users.
- **ResultPage**: The page that shows the results after the quiz is completed.

1.1.2 Quiz Session Module

- **QuizSessionController**: Manages user interactions during a quiz session.
- **QuizSessionService**: Handles the business logic associated with a quiz session.
- **QuizSessionRepository**: Manages the data storage and retrieval for quiz sessions.

1.1.3 Quiz Management Module

- **QuizController**: Manages the interactions related to quiz management.
- **QuizService**: Contains the business logic for quiz management.
- **QuizRepository**: Responsible for storing and retrieving quiz-related data.

1.1.4 Question Bank Module

- **QuestionController**: Manages interactions related to question management.
- **QuestionService**: Contains the business logic for question management.
- **QuestionRepository**: Responsible for storing and retrieving question data.

[width=]viewpoints/development/development.drawio

Figure 1: Module Organization Diagram

1.1.5 Common Processing

Data Validation: Uses React with Formik and Yup for form handling and schema-based validation.

Error Handling: Uses ASP.NET Core middleware for global exception handling, logging errors to a centralized logging service (like Serilog) and returning standardized error responses.

User Authentication: Manages user authentication and authorization using JWT tokens for securing API endpoints. Middleware checks token validity and user roles before processing quiz-related requests.

1.2 Standardization of Design

1.2.1 Design Patterns

1. **Factory Pattern:**
   - **Purpose:** Used for creating instances of different quiz types.
   - **Implementation:** Utilize a `QuizFactory` class responsible for encapsulating the instantiation logic based on the provided quiz type.
2. **Repository Pattern:**
   - **Purpose:** Abstracts data access and creates a clean separation between the business logic and data access layers.
   - **Implementation:** All repository classes must adhere to a common interface, named `IBaseRepository`, which defines methods for CRUD operations.
   - **Interface Requirement:** Any new repository must implement the `IBaseRepository` interface.
3. **Singleton Pattern:**
   - **Purpose:** Ensures a single instance of certain services, like `QuizSessionService`, for managing application-wide state.
   - **Implementation:** Implement the Singleton pattern to guarantee that only one instance of the service is created and accessed throughout the application's lifecycle.

1.2.2 Logging Package and Usage

- **Logging Package:** Consistently use a logging framework like Serilog.
- **Configuration:** Ensure the presence of a logging configuration file (e.g., `serilog.json`) in the application's directory. This file describes logging behavior, such as log levels and output destinations.

- **Usage in Code:** Log messages using the logger obtained from the logging framework's logging API. This logger should be declared as a private static readonly field within the class.

1.2.3 Repository Implementation

- **Interface Requirement:** All repository implementations must adhere to the common interface.
- **Consistency:** Maintain consistency in method signatures and behavior defined by the `IBaseRepository` interface.

1.2.4 Dependency Injection

Dependency Injection (DI) is heavily employed throughout our system. By facilitating loose coupling between components.

1.2.5 Layered Structure and Module Organization

To enhance organization and clarity, we advocate for a structured approach to module organization:

- **Interface Module:** This module serves external interactions and communication, providing a clear interface for external systems or users to interact with our application.
- **Core Module (Business Logic):** At the heart of our system lies the core module, responsible for orchestrating operations and enforcing business rules. This module encapsulates the essential logic driving our application's functionality.
- **Data Access Layer:** The data access layer provides an interface for interacting with data storage, ensuring separation of concerns and facilitating efficient data management.

1.2.6 Consistent User Flows

Quiz Taking Flow: Users navigate from the quiz list to the quiz page, answer questions sequentially, and then view results. Each step is guided by consistent visual cues and navigation buttons.

Quiz Management Flow: Admins follow a flow from creating a new quiz, adding questions, setting correct answers, and publishing the quiz.

1.3 Standardization of Testing

1.3.1 Unit Testing

For our backend unit testing, we use NUnit and Moq for testing ASP.NET Core services and repositories. These tests cover validation logic, business rules, and

data access methods, focusing on service methods and repository functions. They follow the Arrange-Act-Assert pattern and are organized in a dedicated test project that mirrors the main codebase structure. Mock objects are used to simulate dependencies, and cleanup functions reset the state between tests to ensure isolation and reliability of test results.

1.3.2 Integration Testing

For API endpoints, we use Postman to automate comprehensive integration tests. Postman scripts verify responses to various HTTP requests (GET, POST, PUT, DELETE). These tests are organized in collections and executed across multiple environments.

We use Cypress for detailed end-to-end tests. For example, in our upcoming quiz editor implementation, the following test scenarios are covered:

- **Quiz Editor Access:** The test runner opens the quiz management page and verifies that the quiz edit button is visible for the topic owner and not visible for non-owners.
- **Quiz Creation and Editing:** The test runner navigates to the quiz creation page, enters quiz details (e.g., title, questions, answers), and submits the form. It then checks the backend response to confirm the quiz was successfully created. The runner subsequently opens the quiz edit page, modifies quiz details, submits changes, and confirms updates were successful.
- **Quiz Deletion:** The test runner clicks the delete button for a quiz, confirms the action, and checks the backend response.

These tests are stored in a cypress directory and focus on critical workflows, such as user authentication, role-based access, form submissions, data validation, and ensuring data passes correctly between the frontend and backend.

1.4 Instrumentation

Monitoring Performance and Usage:

Logging: Incorporates Serilog for structured logging. Logs include details on user interactions with quizzes, errors, and performance metrics.

Analytics: Uses Google Analytics to track user behavior, quiz engagement rates, and completion statistics. Custom events track specific actions like quiz start, question answer, and quiz completion.

Usage Metrics: Stores data on question difficulty, user scores, and completion times in a PostgreSQL database. Regular reports are generated to identify trends and areas for improvement.

1.5 Codeline Organization

Modular Structure:

Frontend: Organized into directories for components (/components), pages (/pages), services (/services), and utilities (/utils). Each component and page related to the quiz feature has its own directory.

Backend: Follows a similar structure with directories for controllers (/Controllers), services (/Services), repositories (/Repositories), and models (/Models). The quiz feature-specific code is encapsulated within its own subdirectories.

Branching Strategy: Uses Git with a feature branch workflow.

1.6 Stakeholder Concerns

1.6.1 Developers

- Need clear module boundaries and documentation to facilitate development and maintenance.
- Require standardized testing and instrumentation practices to ensure code quality and performance.

1.6.2 Production Engineers

- Concerned with the reliability and scalability of the quiz feature.
- Need comprehensive logging and monitoring to quickly identify and resolve issues.

1.6.3 Testers

- Require detailed test plans and automated testing scripts to efficiently validate the system.
- Benefit from standardized testing approaches to ensure consistent test coverage and results.