**5.1 Description of Implementation**

**5.1.1 System Deployment and Transition**

The **QUIZ AI** platform will be deployed as a centralized web application. Since this is a greenfield project, no legacy data migration is required. The transition into an operational system will follow a **"Big Bang" deployment model**, where the system becomes fully available to students immediately upon the successful configuration of the host server.

**Deployment Workflow:**

1. **Environment Setup:** Configuration of the host PC with the necessary runtime environments (.NET for the backend and a web server for the React frontend).
2. **AI Model Integration:** Loading the fine-tuned AI model into the backend infrastructure.
3. **Site Launch:** Pointing the web domain or IP address to the host machine, making the "QUIZ AI" portal accessible to users.

**5.1.2 Major Tasks and Components**

The implementation effort is divided into the following key tasks:

* **Backend Deployment:** Hosting the **ASP.NET (C#)** API and ensuring the fine-tuned AI logic is responding to requests.
* **Frontend Integration:** Deploying the **React.js** build to the web server to provide the user interface.
* **AI Parameter Configuration:** Final calibration of the AI settings to ensure quiz generation matches user-selected settings and material accuracy.
* **Security Hardening:** Implementing authentication protocols and firewall rules on the host PC to prevent unauthorized access to the backend.

**5.1.3 Resource Requirements**

To support the implementation and ongoing operation of QUIZ AI, the following resources are required:

| Resource Category | Description |
| --- | --- |
| **Hardware** | A dedicated "Main Server" PC provided by the development team to host the database, backend, and frontend. |
| **Software** | .NET Runtime, Node.js (for build), a Database Management System, and the fine-tuned AI model files. |
| **Facilities** | A stable power supply and high-speed internet connection for the host PC to ensure website uptime. |
| **Personnel** | The original student development team will act as the primary **Support Team** for maintenance and troubleshooting. |

**5.2 Programming Language and Technology**

This section identifies the technical stack and tools used to implement the QUIZ AI system. The solution utilizes a hybrid hosting model where the frontend is deployed to a public web server, while the backend and database remain on a local secure host.

**5.2 .1Core Programming Languages & Frameworks**

| Name | Code / Acronym | Type | Description |
| --- | --- | --- | --- |
| **C#** | C# | COTS (Open Source) | Primary language for backend logic and AI orchestration. |
| **ASP.NET Core** | ASP.NET | COTS (Open Source) | Framework used to build the RESTful API and manage security middleware. |
| **React.js** | React | COTS (Open Source) | Frontend library used to build the interactive student interface. |
| **Transact-SQL** | T-SQL | COTS | Scripting language used for database queries and management. |

**5.2 .2 Solution-Specific Software & AI Models**

*These components represent the custom-built "State-specific" parts of the project.*

| Name | Acronym | Type | Role |
| --- | --- | --- | --- |
| **Fine-tuned TinyLlama** | TinyLlama-1B | **State-specific** | Small Language Model (SLM) fine-tuned on 1B tokens for specialized quiz generation. |
| **QUIZ AI DB** | DB | **State-specific** | SQL Server database containing custom schemas for user materials and quiz data. |
| **SQL Server** | MSSQL | COTS | Relational Database Management System (RDBMS) hosted on the main server PC. |

**5.2 .3 Implementation & Security Technologies**

| Technology | Category | Usage |
| --- | --- | --- |
| **JSON Web Tokens** | **JWT** | Used for secure, stateless authentication between the React frontend and C# backend. |
| **Swagger / OpenAPI** | **Testing Tool** | Integrated into the ASP.NET backend to facilitate API testing and documentation. |
| **Base Model API** | **Interface** | The bridge used by the C# backend to communicate with the locally running TinyLlama model. |

**5.2.4 Support & Development Software**

| Software Name | Purpose |
| --- | --- |
| **Visual Studio (VS)** | Primary IDE for C# backend development and SQL management. |
| **VS Code** | Used for React.js frontend development and styling. |
| **Localhost Environment** | Facilitates the execution of the backend and AI model without public cloud exposure. |
| **Production "dist" Folder** | The optimized React build folder deployed to the separate hosting server for the UI. |

**Technical Note on Implementation**

The system follows a **Decoupled Architecture**. The user interface is served via a public URL (hosting the dist folder), which securely calls the backend API hosted on the "Main Server" PC. Traffic is protected using **JWT**, ensuring that only authorized users can trigger the AI quiz generation.

**5.3 part of implementation**

**5.3.1 Data-Back Layer**

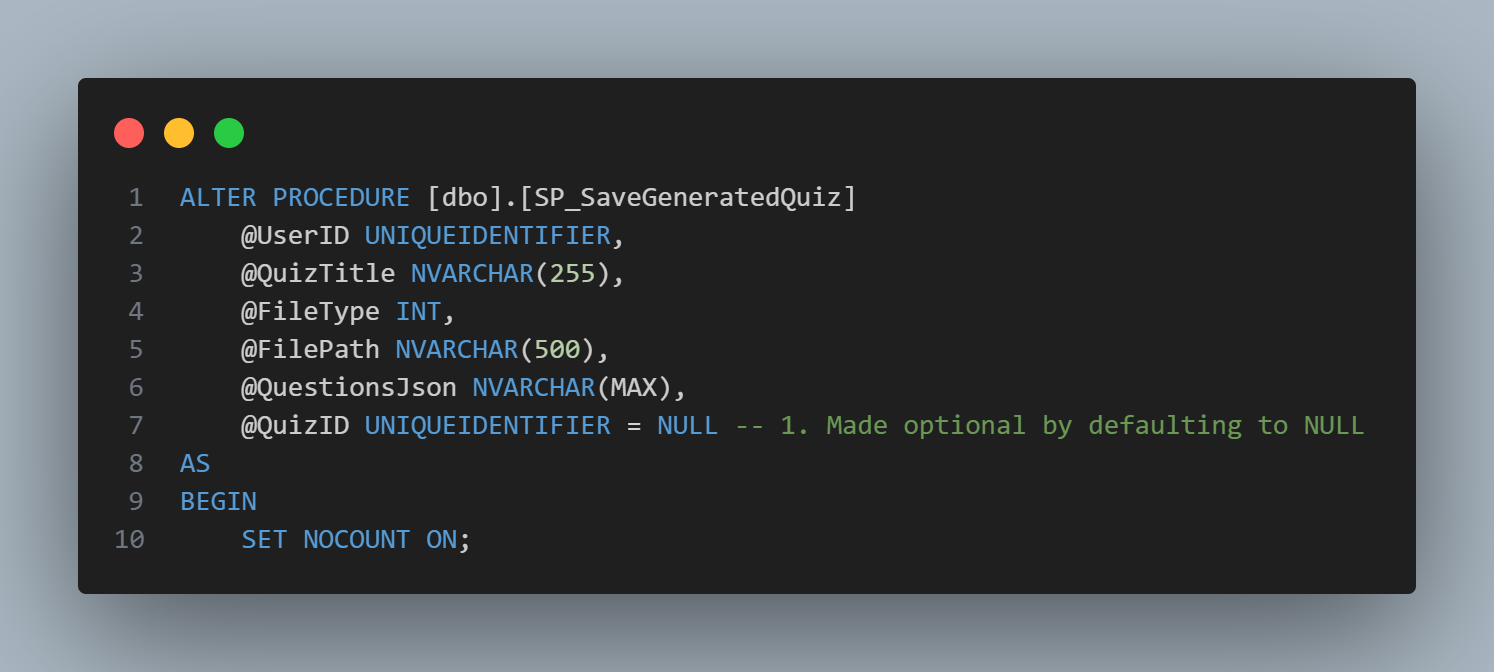
The Data-Back Layer is responsible for all interactions with the database. This layer communicates with the database exclusively through **stored procedures**, which are executed at the database level.

Using stored procedures helps ensure **database integrity and security**, as queries are processed internally within the database rather than being sent as dynamic string-based queries from the C# application. This approach reduces the risk of SQL injection attacks and enforces a controlled access pattern to the database.

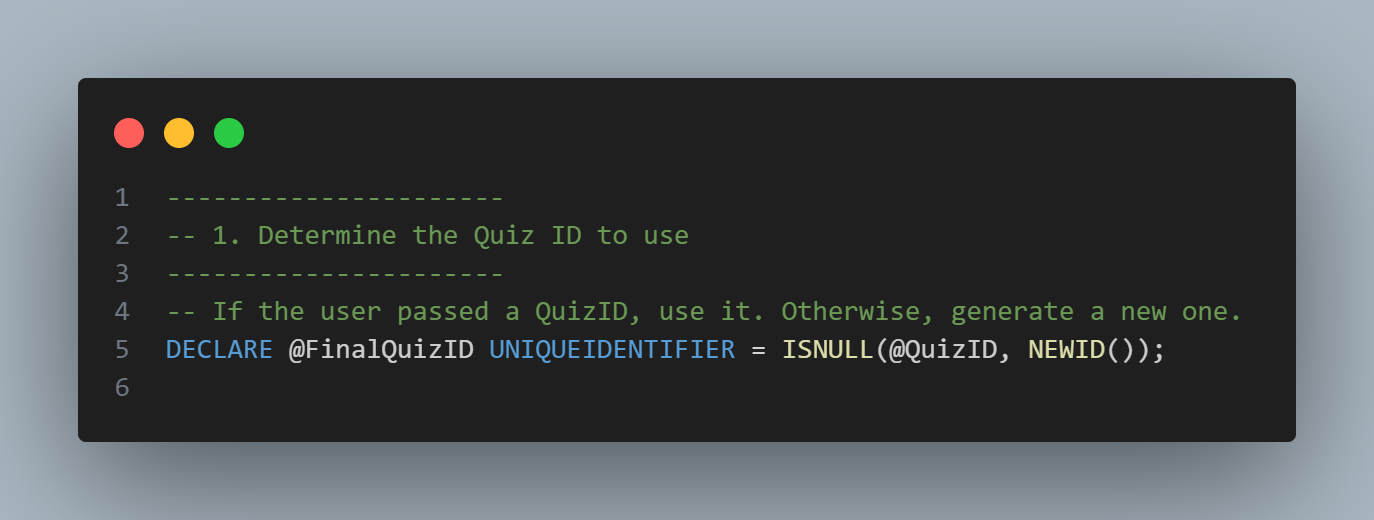
In the following section, we introduce and explain some of the stored procedures that were designed and used within this layer.

One of the most frequently used stored procedures in the system is **SaveGeneratedQuiz**. This procedure is responsible for creating a new quiz and storing it in the database. It receives several parameters, including the **User ID**, **Quiz Title**, **File Type**, **Questions JSON**, and **Quiz ID**, which together define the quiz content and its metadata.

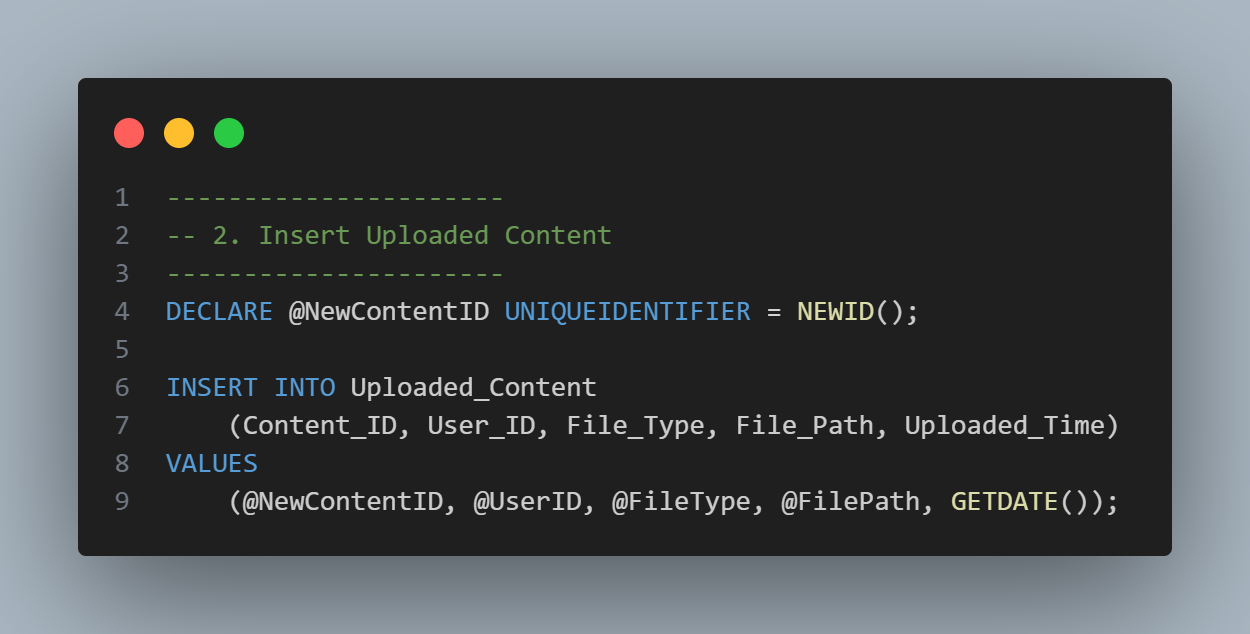
This section of the procedure is the **header**, where all required input parameters are declared. These parameters provide the necessary information for the procedure to correctly generate and persist the quiz data in the database.



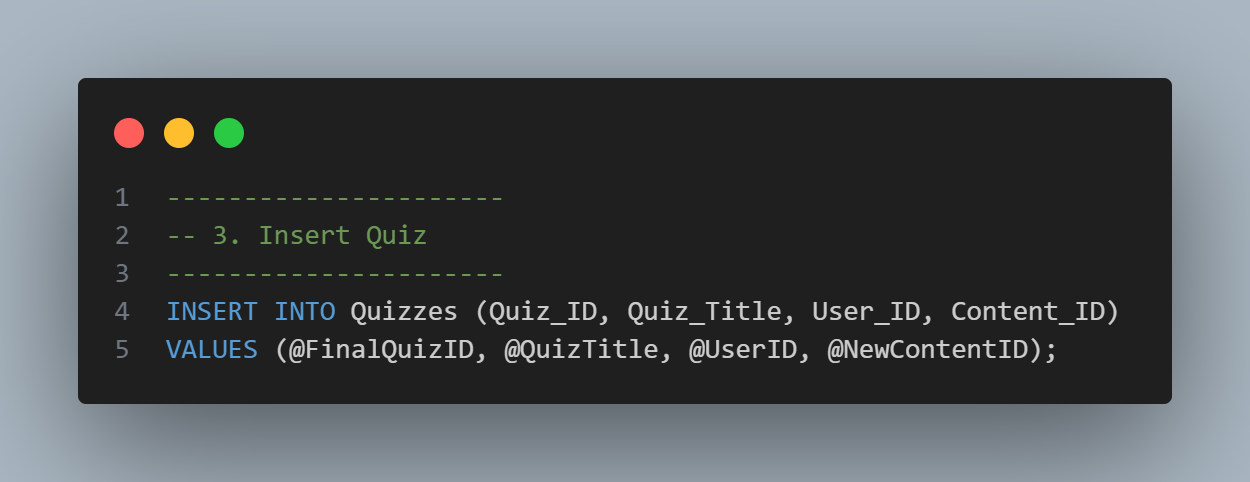
The first part of the procedure is responsible for determining the **Quiz ID**. If the quiz is being created for the first time, a new **Quiz ID** is generated. Otherwise, the procedure uses the **Quiz ID** provided as an input parameter, allowing an existing quiz to be updated or reused.



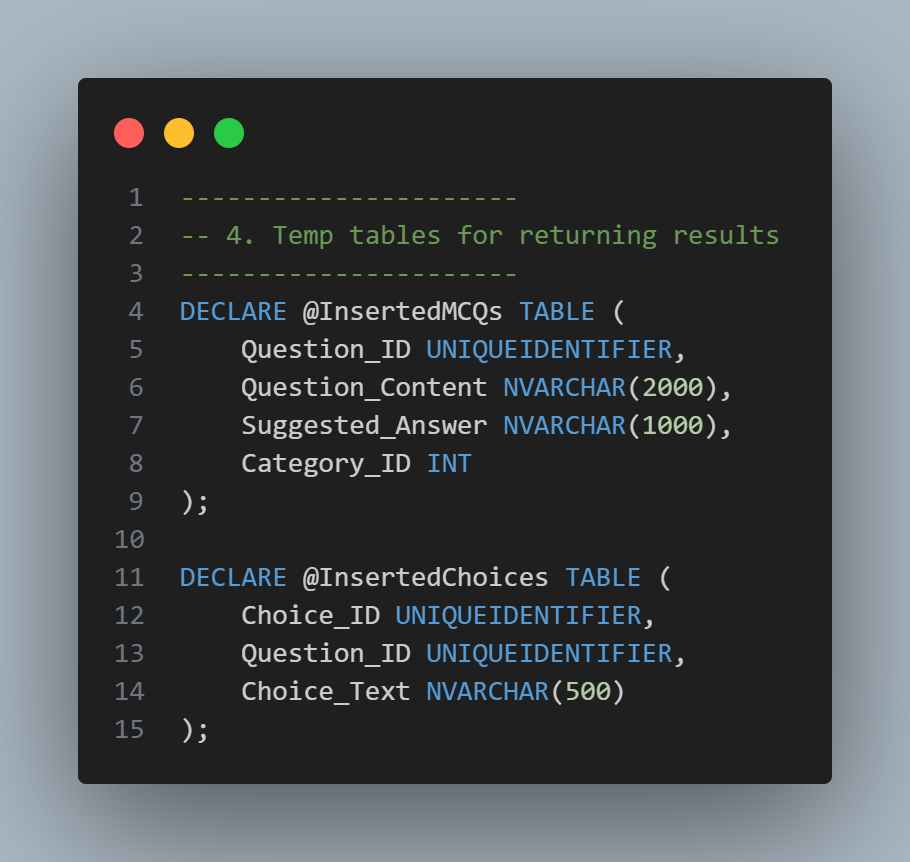
The second part of the procedure is responsible for inserting the information related to the uploaded content, which is stored on the disk. This step records the file’s metadata in the database, enabling the system to track and associate the uploaded content with the corresponding quiz.



The third part of the procedure focuses on creating a new quiz by inserting the quiz information into the **Quizzes** table. This step stores the quiz’s core details, such as its title, type, and associated identifiers, ensuring that the quiz is properly registered in the database.



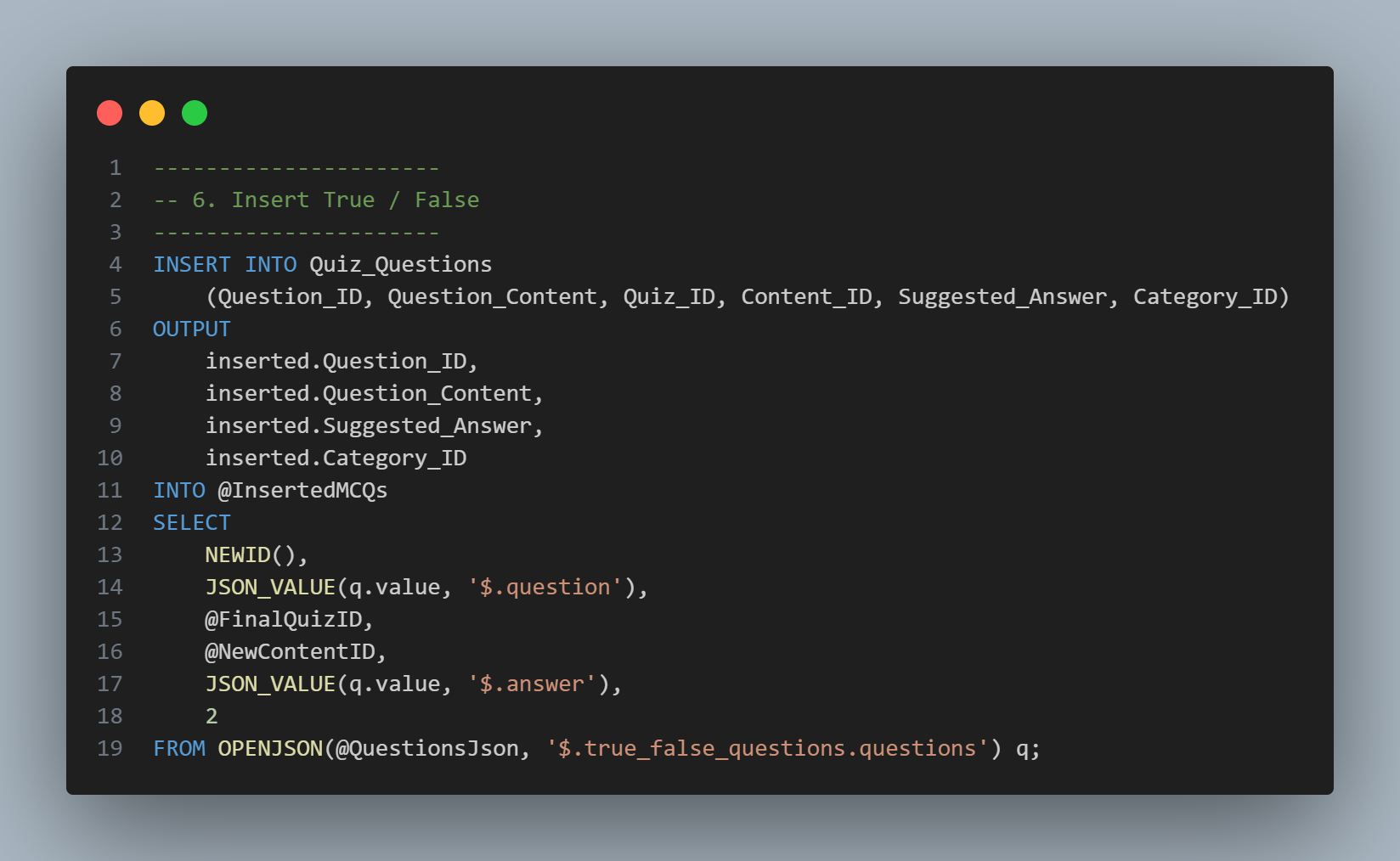
The fourth section creates a **temporary table** to store the results that will be returned to the backend (C#) and subsequently to the frontend. This temporary table holds the required **unique identifiers**, allowing the system to retrieve all necessary data without reopening an additional database connection. As a result, the entire operation is completed within a **single database connection**, improving performance and reducing overhead.



The fifth part of the procedure is responsible for handling the **JSON data** received from the backend, which originates from the AI model. In this section, the JSON content is parsed and processed to extract the quiz questions.

Based on the extracted data, the quiz questions are inserted into the relevant database tables, along with their associated answer choices. This section is specifically designed to handle **multiple-choice questions (MCQs)** only.



The sixth section is responsible for handling **true/false (T/F) questions**. In this part, the processed data is inserted into the corresponding database tables, ensuring that true/false questions are stored and linked correctly to the associated quiz. 

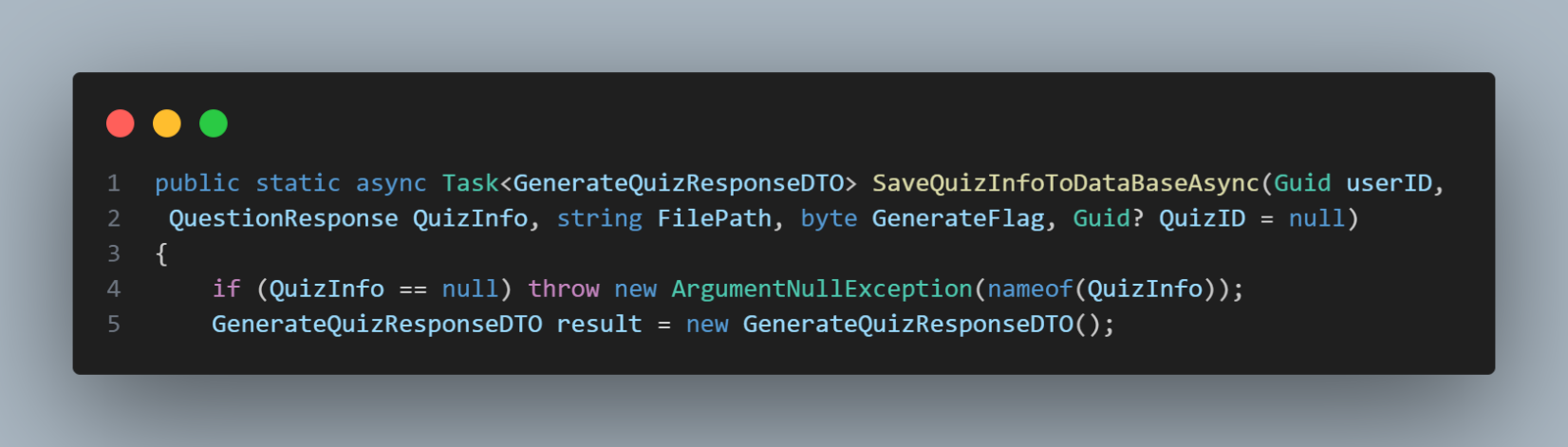
The seventh section is to return the results of the saved quiz, the questions in it, and the choices, with their id’s, in separate query parts, I know this isn’t the best practice, but when we do it in this part, we prevent opening multiple database connections, which decreases the speed.



In this section, we demonstrate how the stored procedure is handled within the backend, including the processing of its sub-queries.

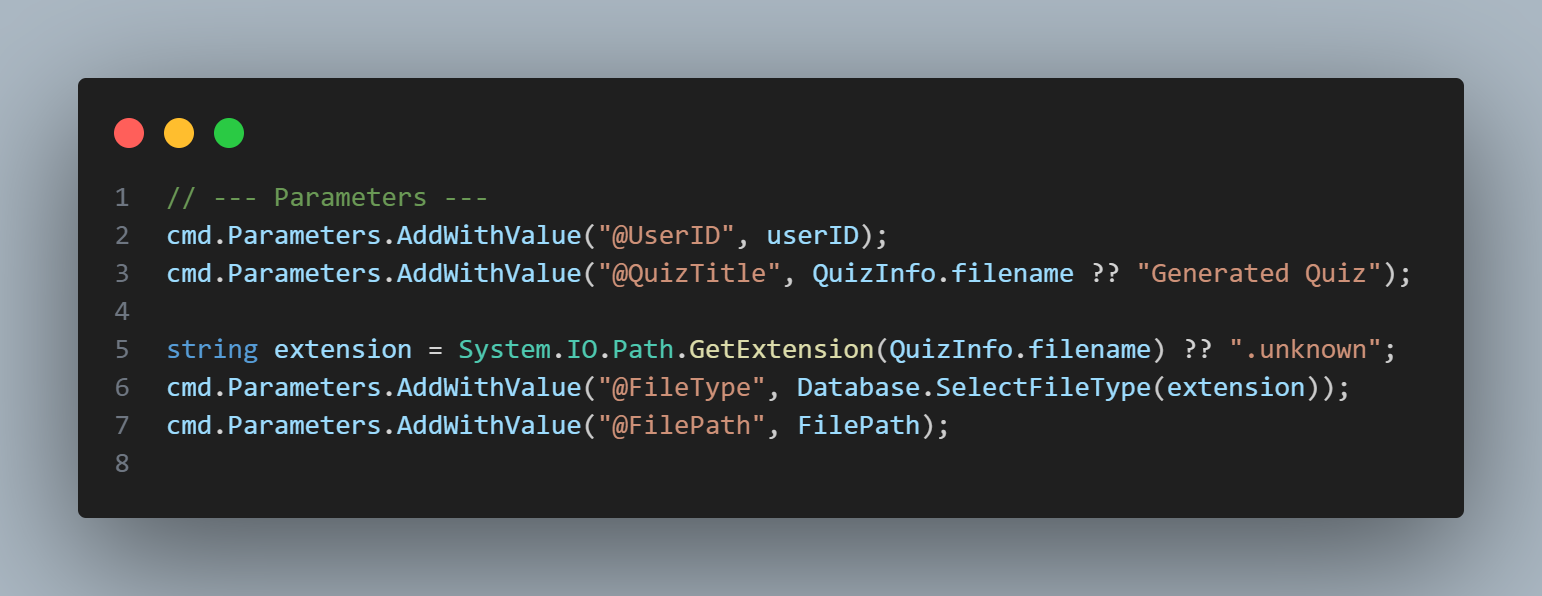
The first part presents the **function header** and the parameters required by the method. This section also includes basic **data validation** and **error handling** to ensure that invalid input is detected before interacting with the database.

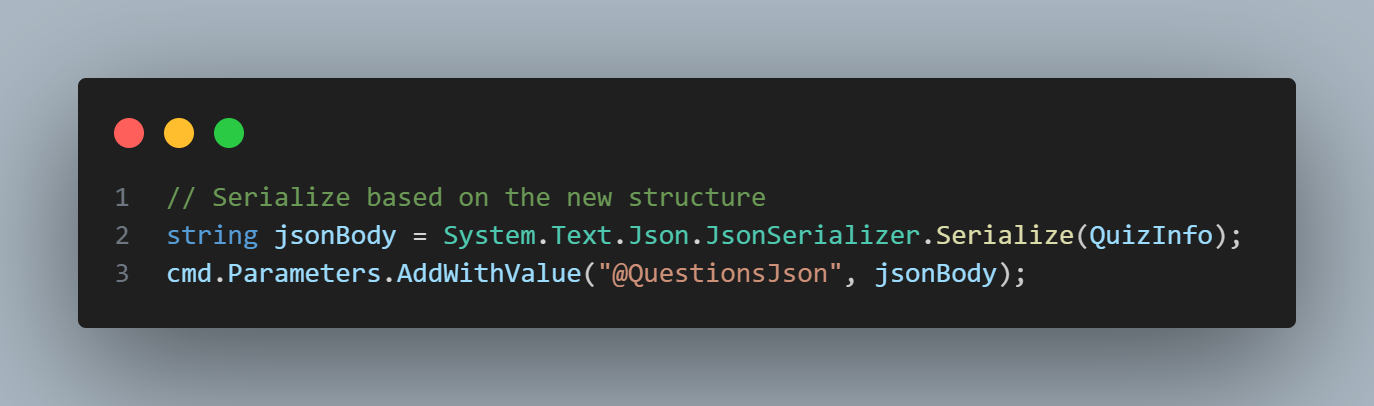
The function is implemented as an **asynchronous method**, allowing the system to continue executing other operations while waiting for the database to process and return the requested data. This approach improves system responsiveness and overall performance.

 In this section, the database connection is established using a valid connection string retrieved from the **appconfig.json** configuration file. A SQL command object is also initialized to execute the database operation. Both the connection and the command are wrapped within **using statements** to ensure proper resource management and to prevent common human errors, such as forgetting to close database connections.

Finally, the command type is explicitly set to **Stored Procedure**, ensuring that the database operation is executed at the procedure level rather than as a raw SQL query.



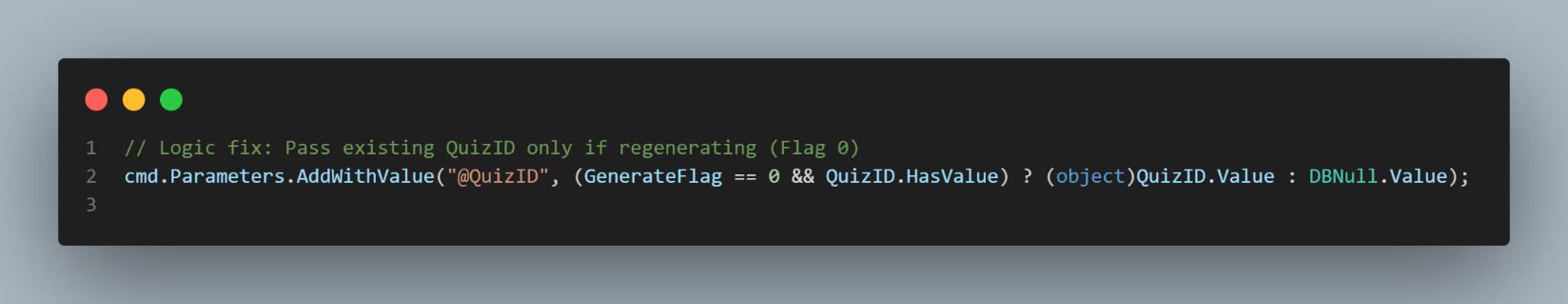
This section is dedicated to supplying the parameters required by the stored procedure. It defines and assigns all necessary input values, ensuring that the procedure receives the complete and correct data needed for execution.

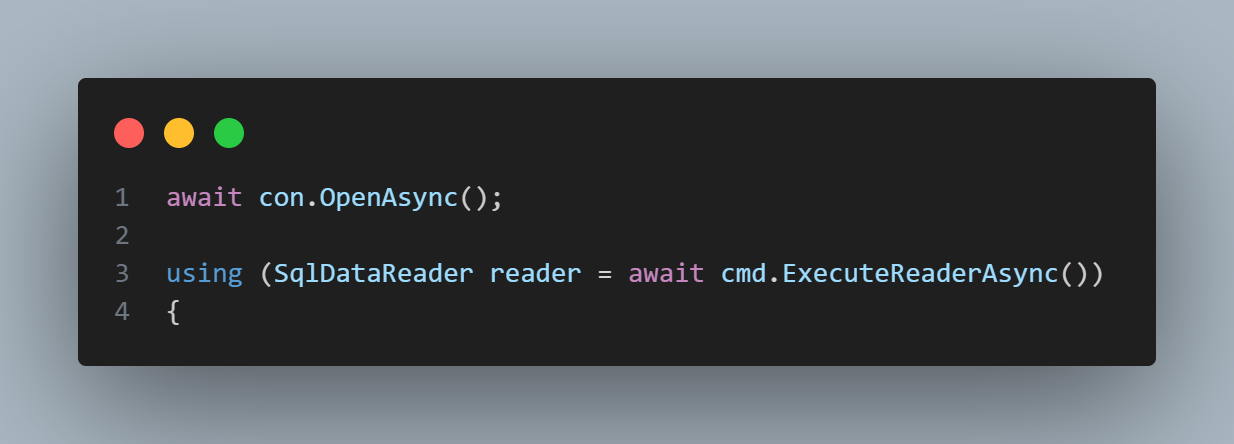
In this section, the class structures are serialized into **JSON format** before being sent to the database. This approach is necessary because **T-SQL does not support complex class or structure types**. By serializing the data into JSON, the stored procedure can parse and process the structured information within the database layer.

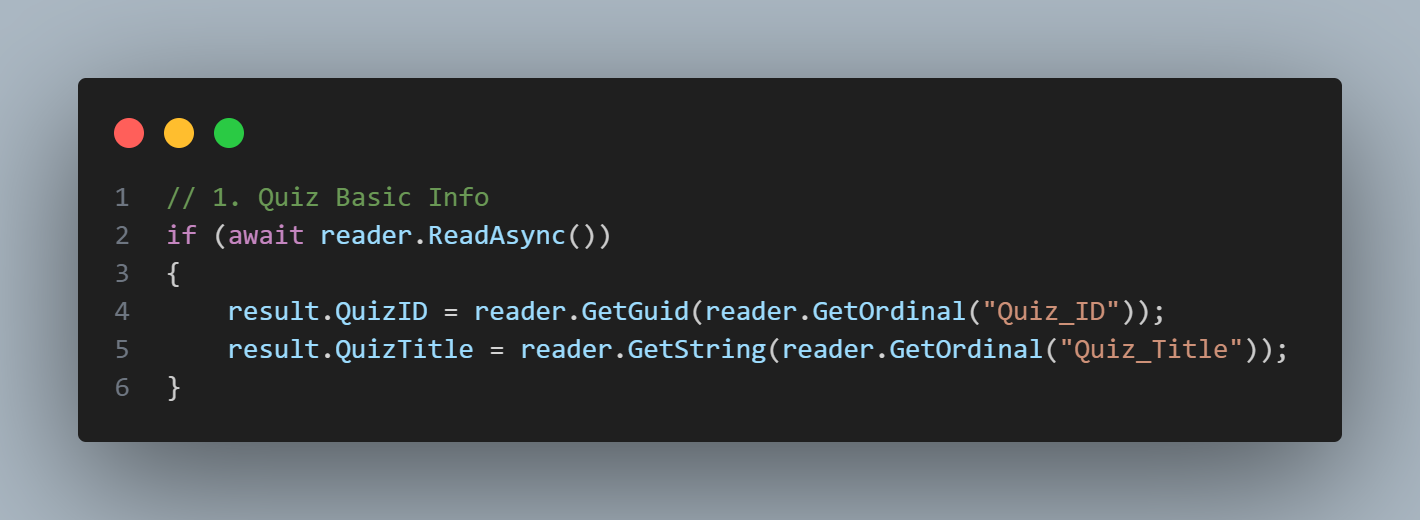
In this section, the **ternary operator** is used to determine whether to send a value or NULL for the **Quiz ID**.

* If the **Generate Flag** is 1, this indicates a **new quiz**, and NULL is sent so that the database generates a new Quiz ID.
* If the **Generate Flag** is 0, this indicates a **regenerated quiz**, and the existing Quiz ID is used to ensure that the same quiz is updated in the database.

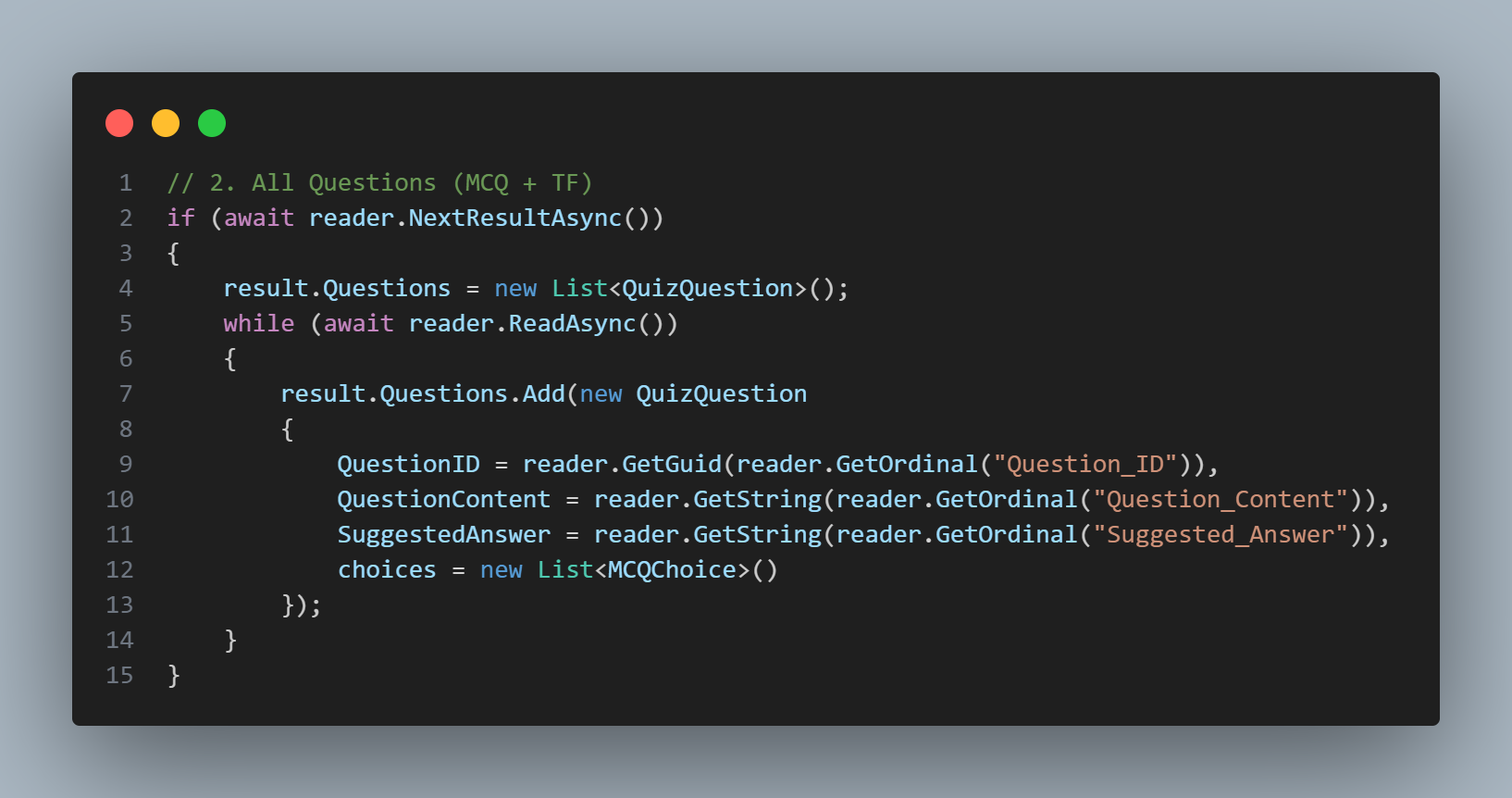
This approach allows the procedure to handle both new and regenerated quizzes efficiently while maintaining database consistency.



In this section, the method leverages **asynchronous programming** to prevent blocking the system while waiting for the database response. A **SqlDataReader** is used to efficiently handle queries that return **multiple rows**, allowing the system to read data row by row as it is retrieved from the database. This approach improves performance and resource utilization, especially when processing large result sets.

In this section, only the **basic quiz information** is retrieved from the first result set. This includes essential details such as the **Quiz ID** and **Quiz Title**, which are then stored in the response object for further processing.

In this section, all **quiz questions** are retrieved from the second result set using a **while loop** that iterates over the rows returned by the SqlDataReader. Both **multiple-choice (MCQ)** and **true/false (T/F)** questions are processed and added to the response object, preparing them for the next step of associating answer choices.



In this section, the **answer choices** for all questions are retrieved from the database. The process uses the same method as before, iterating over the result sets with **reader.Read()**. Each choice is matched to its corresponding question and added to the question’s choices collection in the response object.

Finally, we return the questions, with the choices of each question.



All **Data-Back functions** are designed to handle **direct communication with the database**. They contain **minimal logic**, focusing only on processing inputs and formatting outputs, while leaving business rules and complex operations to higher layers of the application. This ensures a clear separation of concerns and simplifies maintenance.

**Business Logic Layer:**

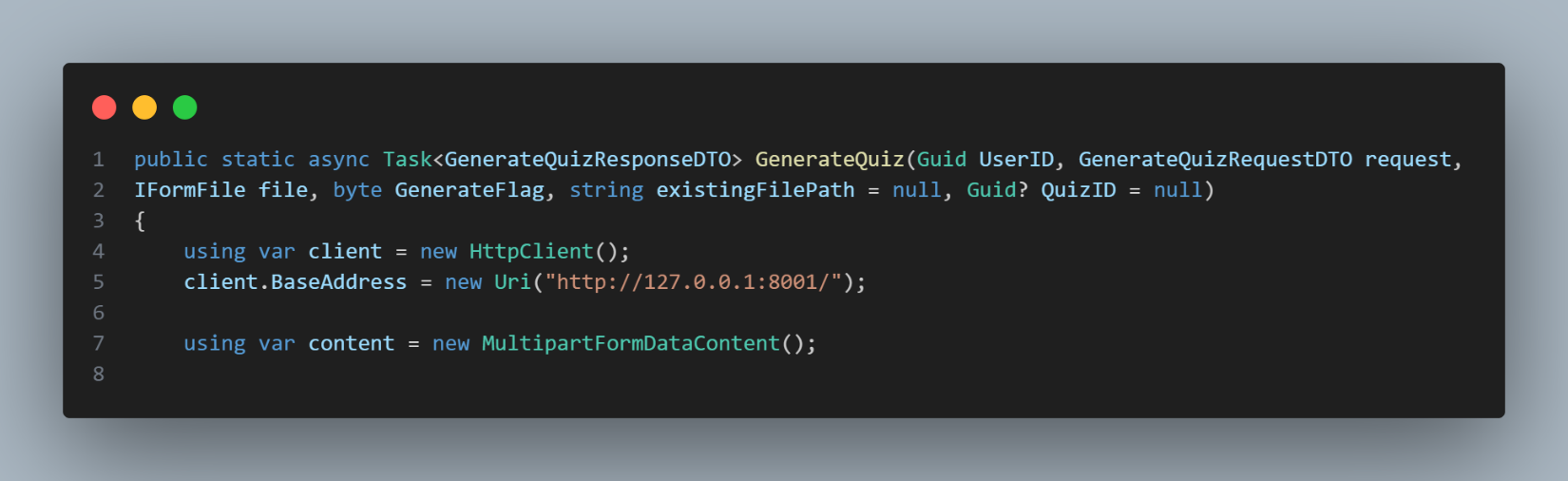
In the **Business Logic Layer**, we handle the program’s core logic, following the workflow from the stored procedure, through the Data-Back layer, and into the business logic itself. This layer manages a variety of responsibilities beyond simple data operations.

It includes functionality for **JSON Web Tokens (JWTs)** to authenticate users, such as generating tokens for account verification, password recovery, and sharing quizzes. It also handles **file management**, including saving and deleting files, and monitors **server health**, including database connectivity, AI model readiness, disk space, and email service availability.

In the sections that follow, we will focus specifically on the **Generate Quiz process**, detailing how this layer orchestrates the flow between the AI model, file storage, and database operations.

In this section, we observe that the function is implemented as an **asynchronous method**, as explained earlier. Using asynchronous execution allows the system to remain responsive while waiting for external operations, such as file handling, AI model processing, and database interactions.

A key component that determines the workflow of this function is the **Generate Flag**. This flag controls whether the user is generating a **new quiz** or **regenerating an existing quiz**, enabling the function to follow the appropriate execution path for each scenario.



In this section, the **Generate Flag** is evaluated. In this scenario, the flag value is assumed to be 1, which indicates that a **new quiz** is being generated. Under this condition, the workflow follows these steps, starting with saving the uploaded file to the local disk.



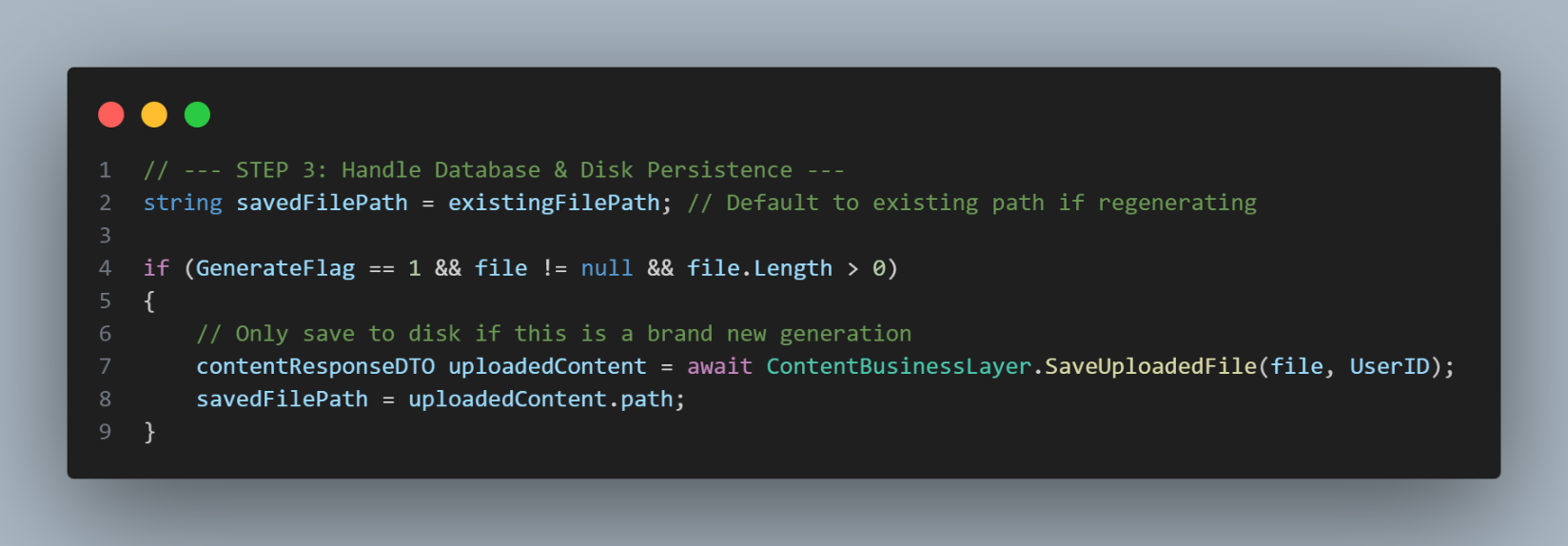
In this section, the **quiz regeneration workflow** is handled. When the **Generate Flag** is set to 0, the system loads the previously uploaded file from the local disk using the stored file path. The file is read asynchronously and attached to the request sent to the AI model, ensuring that the quiz is regenerated using the same original content. The appropriate content type is set based on the file extension to allow correct processing by the AI model. If the original file cannot be found, an exception is raised to prevent invalid regeneration.



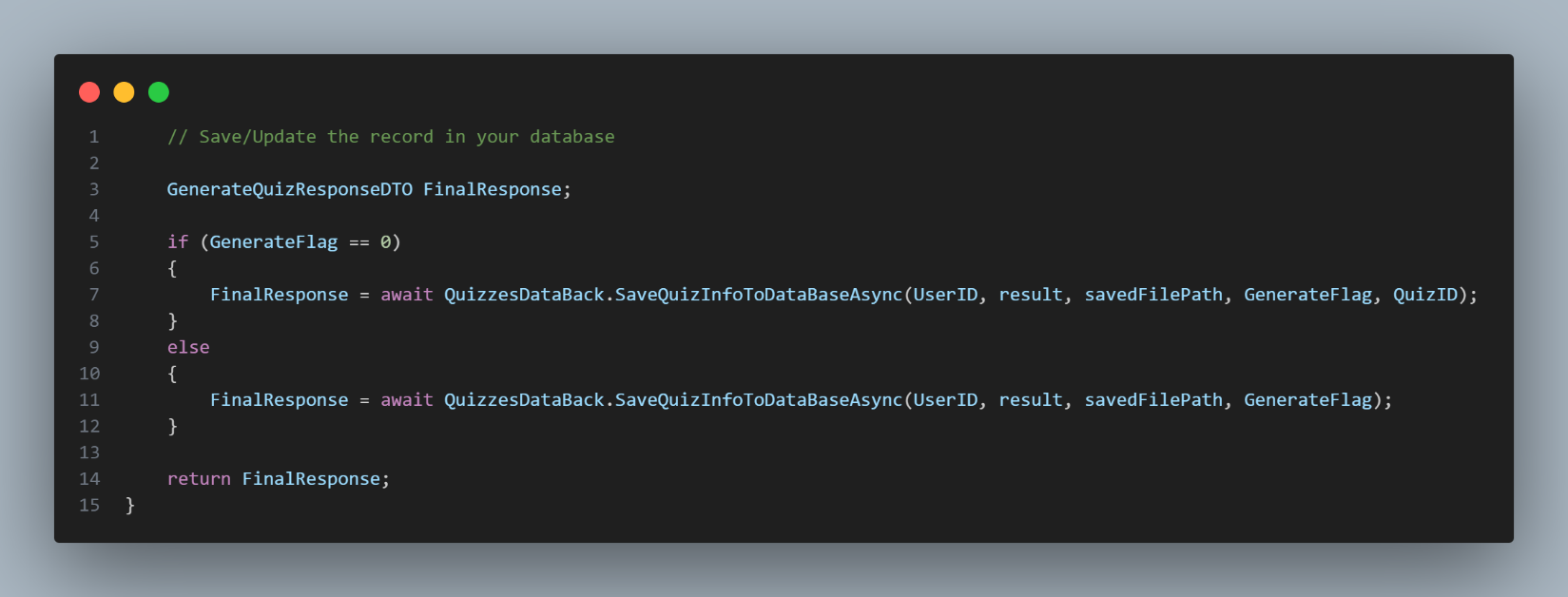
In this section, the request is sent to the **AI model**, which is exposed through a **Python FastAPI endpoint**. The model processes the provided content and generates the quiz, returning the result in **JSON format**. This response is then handled in the C# backend, where it is **deserialized into Data Transfer Objects (DTOs)** for further processing within the system.

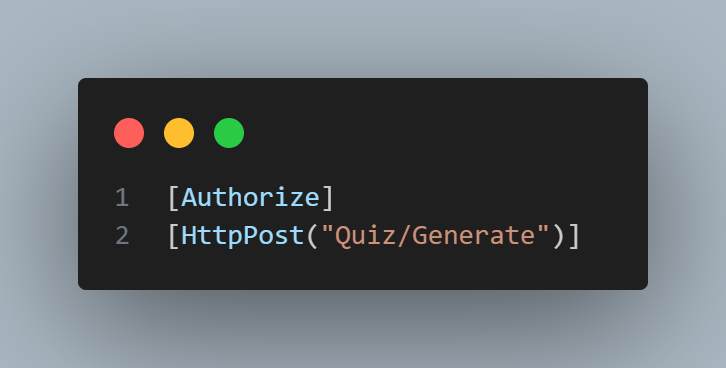


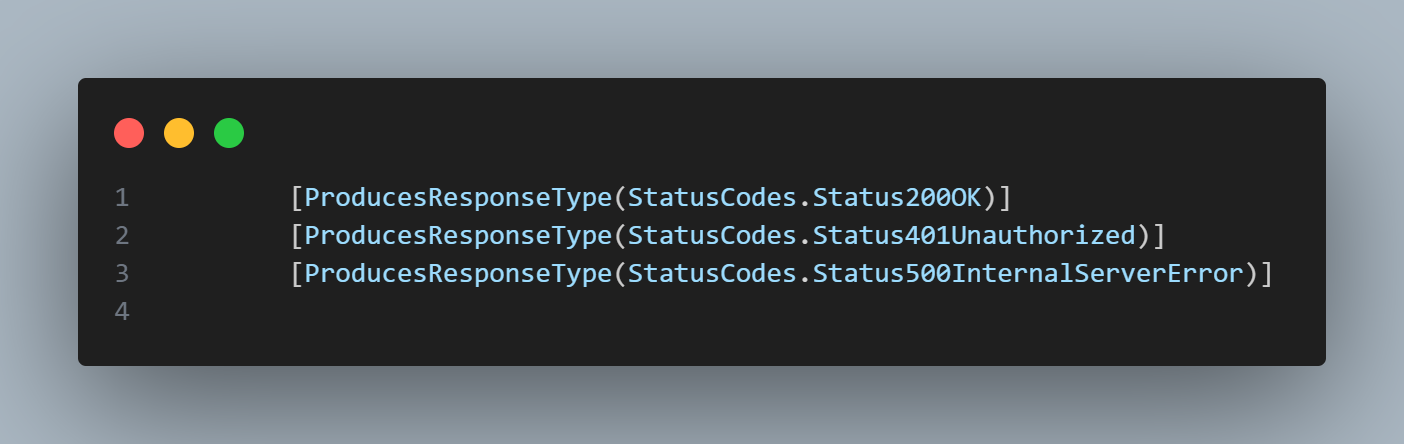
In this section, the system handles **file persistence** for the quiz generation process. If the operation represents a **new quiz generation**, the uploaded file is saved to the local disk and its path is stored for later use. In the case of quiz regeneration, the existing file path is reused, avoiding unnecessary file duplication and ensuring consistency across quiz versions.

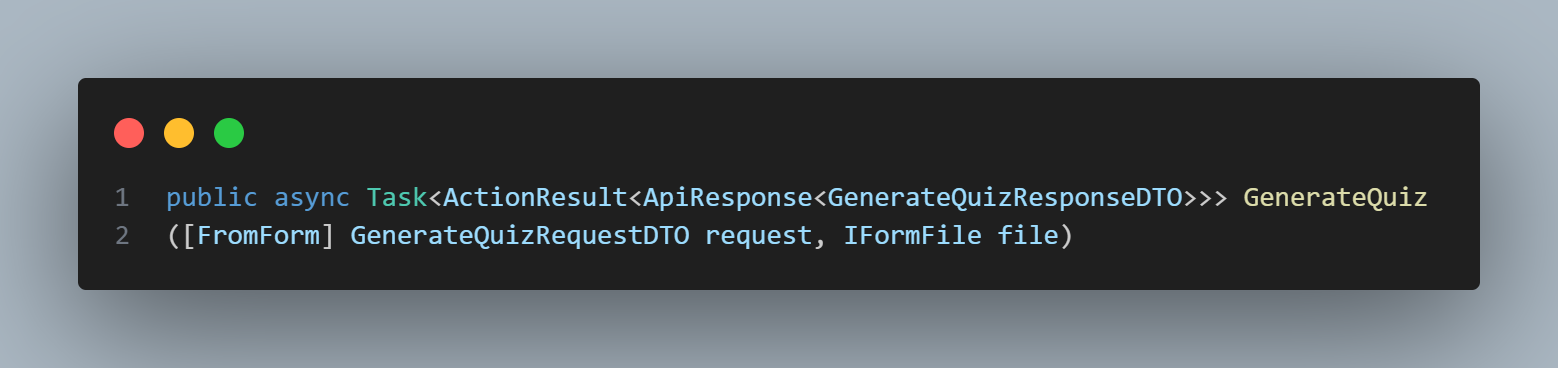


In this final section, the generated quiz data is saved or updated in the database by invoking the Data-Back layer. If the operation represents a **quiz regeneration**, the existing Quiz ID is passed to ensure that the same quiz record is updated. Otherwise, a new quiz record is created. Once the database operation is completed, the fully populated response object is returned to the caller.

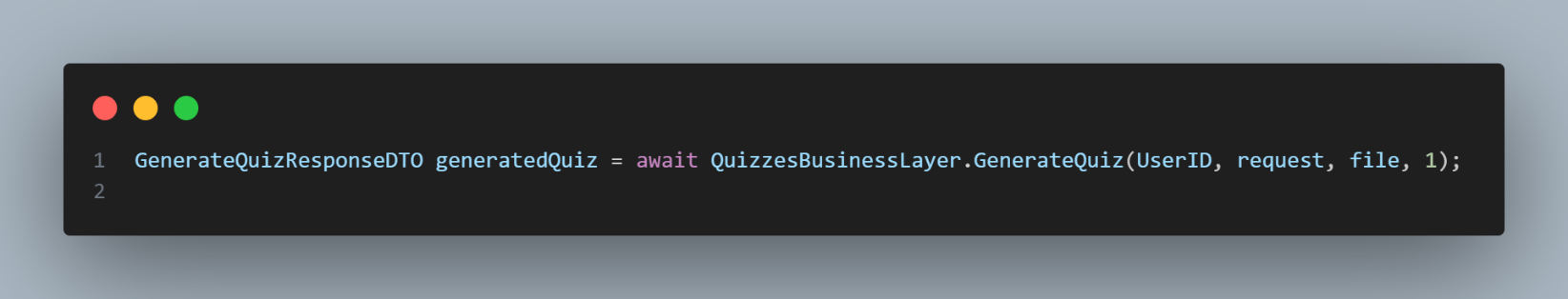


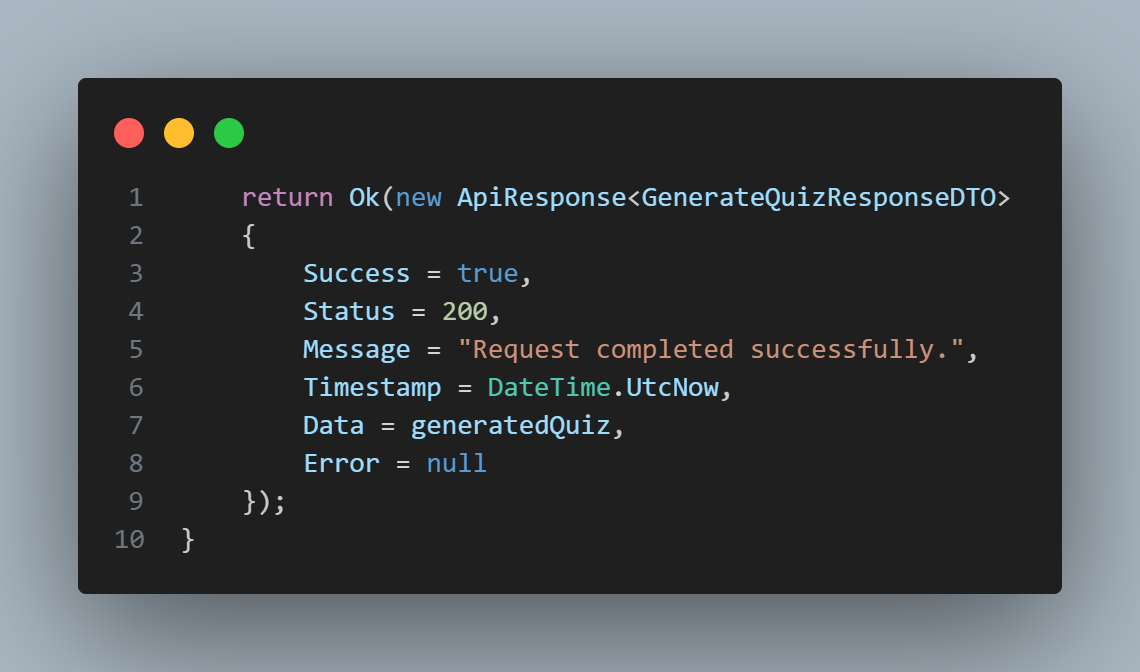
This section ensures that the user is authorized to use the function (i.e., the user is logged in and considered valid by possessing a valid JWT). Additionally, this section defines the endpoint’s name, which will be called from the front end.

In this section, we document the possible status codes that may be returned by the function, including successful responses, authorization failures, and internal server errors. 

In this section, we define the function signature, which contains the required parameters needed for the function to operate correctly. The function is also asynchronous, ensuring that its execution does not block other parts of the application while it is running.

In this section, we validate the user’s identity using the JSON Web Token (JWT). If the token is invalid or missing, the user is not allowed to interact with the system. Otherwise, the request is authorized, and the user ID is extracted from the JWT to be used in subsequent operations.

In this section, we invoke the Business Logic Layer to handle the core workflow, including saving the required information, sending the request to generate the quiz, and coordinating all related operations. This abstraction keeps the controller lightweight and delegates the application logic to the appropriate layer.

In this section, if the process completes successfully without any errors, we return a **200 OK** status code along with the generated quiz data, indicating that the request was handled correctly.

At the final level, if an unexpected error occurs during the process, we return a **500 Internal Server Error** status code to indicate a server-side failure.



This outlines the process of generating a new quiz, and the same workflow is applied across all other functions and endpoints. This approach reflects how the backend is structured, ensuring consistency, maintainability, and a clear separation of responsibilities.

This section covers how the frontend calls the endpoint and consumes its response. It explains how requests are sent, how data is passed to the backend, and how the returned quiz information is processed and displayed in the user interface.

This snippet shows how the frontend calls the backend to generate a quiz from an uploaded file: Builds a FormData payload containing the file (and optional settings like question counts). Sends an authenticated POST request to the generate endpoint (adds Bearer token). Handles non-OK responses and returns either the generated quiz data or a clear error object. This represents the project’s core feature: turning user content into an auto-generated quiz through an API call.



And now, with some random codes from the frontend:

This snippet shows how the app is wrapped with context providers so authentication state (current user, token, login/logout) and exams state (exam list, selected exam, actions) become global and accessible to any component without prop-drilling.

This snippet shows how the AuthContext.Provider exposes auth-related state and functions to the rest of the application through its value prop (for example: user, token, isAuthenticated, login, logout, loading, etc.). In other words, it’s the “public API” of your auth layer that other components consume via useContext(AuthContext).



This snippet shows how the exams provider exports exams-related state and operations (for example: exams, fetchExams, currentExam, submitAnswers, loading, etc.) to any screen/component. It represents the “shared exams/quiz state” that multiple pages can use consistently (library page, quiz page, shared exam route, etc.).

