Group 03

Bilinguo
Software Architecture Document

Version 1.6

Revision History

Date	Version	Description	Author
06/05/2024	1.0	Initial version	Ngọc Châm
10/05/2024	1.1	Fill out section 1. Introduction	Trọng Phúc
14/05/2024	1.2	Fill out section 2. Architectural Goals and Constraints Fill out section 3. Use-case Model	Ngọc Châm
16/05/2024	1.3	Fill out section 4. Logical View	Minh Triết + Trường Sinh
16/05/2024	1.4	Fill out section 5. Deployment	Tấn Phát
16/05/2024	1.5	Fill out section 6. Implementation	Trọng Phúc
18/05/2024	1.6	Review and complete the document	All members

Table of Contents

Software Architecture Document

1. Introduction

The Software Architecture Document (SAD) serves as a comprehensive guide outlining the architecture and design decisions for Bilingou. This document is intended to provide a clear understanding of the software's structure, components, and interactions, enabling stakeholders to make informed decisions throughout the development process and beyond.

1.1 Purpose

The primary purpose of the Software Architecture Document is to:

- Communicate the Architecture: It serves as a communication tool for all stakeholders, including developers, project managers, quality assurance teams, and clients. By presenting the architecture in a structured manner, it ensures everyone involved has a common understanding of the software's design.
- Guide Development Teams: The document acts as a reference for development teams, providing them with a blueprint for the implementation of the software. It outlines the high-level design and key components to ensure consistency and maintainability during the development process.
- Basis for Decision-Making: The SAD supports decision-making processes by providing insights into the architectural trade-offs, rationale, and implications of design choices. It helps in making informed decisions about changes, upgrades, and scalability.

1.2 Scope

The scope of this Software Architecture Document encompasses the entire architecture of Bilinguo. It includes both functional and non-functional aspects of the software, highlighting key design decisions, patterns, and technologies used.

1.3 Definitions, Acronyms, and Abbreviations

- **SAD:** Software Architecture Document
- **API:** Application Programming Interface
- **UI:** User Interface
- **HTTP:** Hypertext Transfer Protocol

1.4 References

- **DJ4E - Django for Everybody**
- **Use-case Specs --rup_ucspec - Google Docs**
- **References from Moodle (Software Engineering course)**

Overview of the Software Architecture Document

The SAD is structured as follows:

- Introduction: This section provides an overview of the entire Software Architecture Document, including

its purpose, scope, definitions, acronyms, abbreviations, and references.

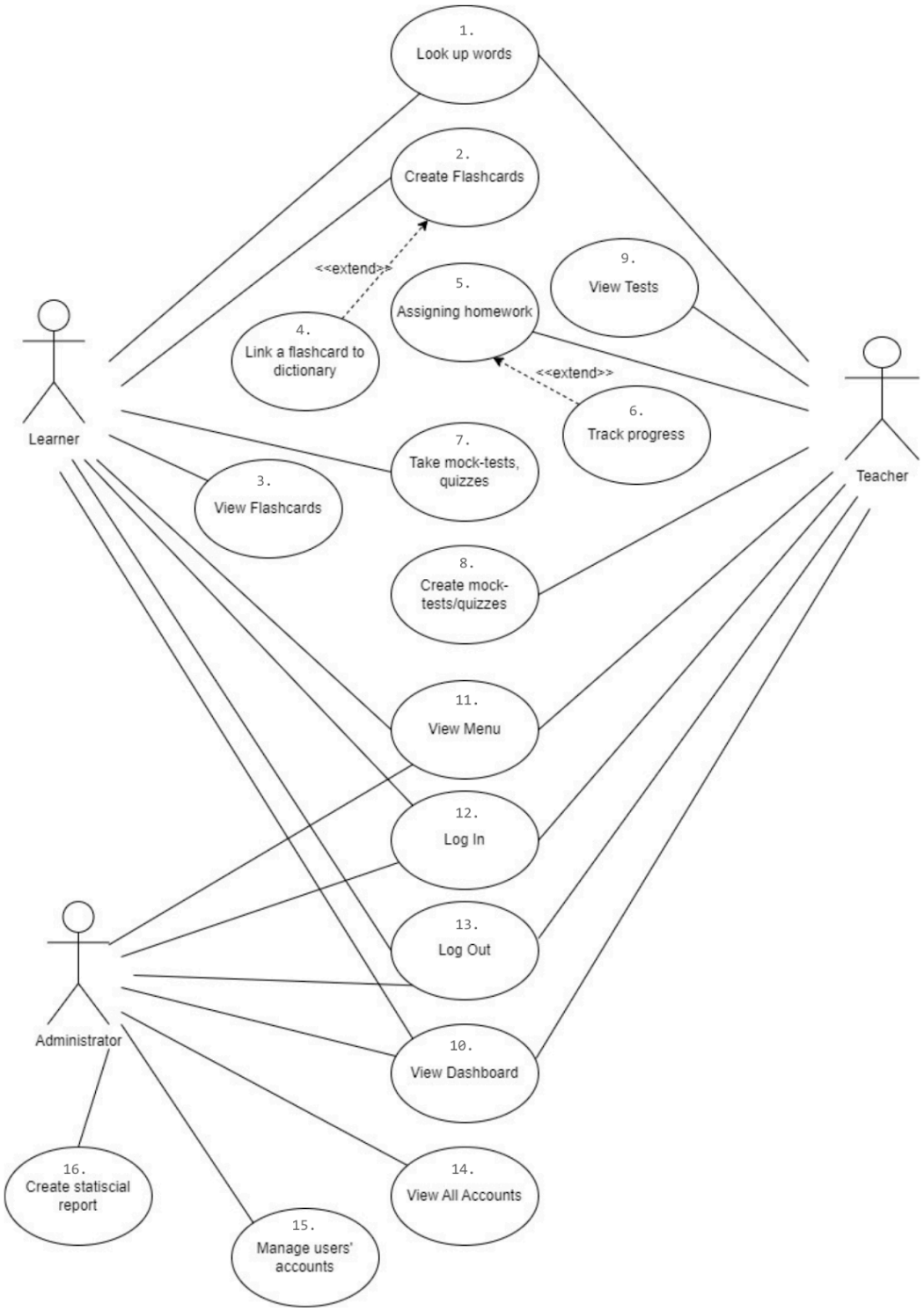
- Architectural Goals and Constraints: Here, we outline the high-level goals of the software architecture and any specific constraints that influence the design decisions.
- Use-Case Model: The Use-Case Model section presents a comprehensive overview of the software's functional requirements from the perspective of end-users. It identifies and describes the primary use-cases that the software is intended to support.
- Logical View: This section provides detailed descriptions of individual system components, their functionalities, interfaces, and interactions with other components.
- Deployment: This section outlines the deployment architecture, including hardware and software requirements, scalability considerations, and potential bottlenecks.
- Implementation View: The Implementation View delves into the technical aspects of the software architecture, providing insights into how the logical components and modules are implemented. It includes detailed information about the programming languages, frameworks, libraries, and technologies used to build the software.

By presenting this Software Architecture Document, we aim to provide a comprehensive understanding of the software's architecture, empowering all stakeholders to collaborate effectively and contribute to the success of Bilinguo.

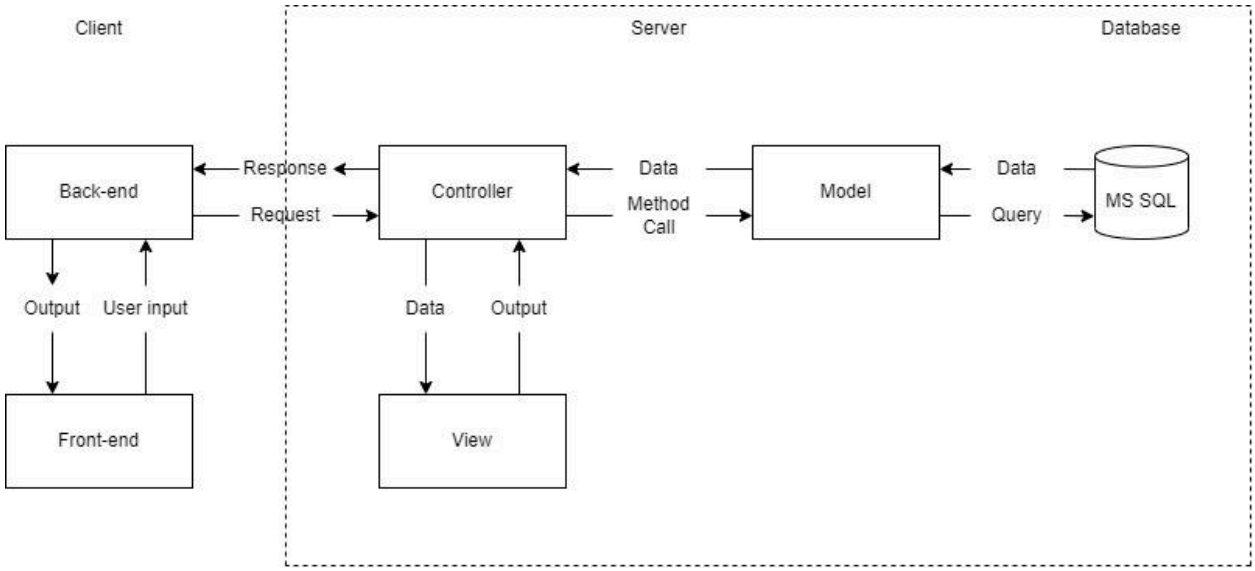
2. Architectural Goals and Constraints

- The system must have a reasonable architecture to ensure the full completion of the functions in the Use-case Specification document. Among them, managing and tracking learning progress is an important and complex function.
- The major design and implementation constraints for the application include:
    - Simplicity
    - Flexibility
    - Security
    - Maintainability: Ensuring the website's maintenance and identifying as well as resolving system issues in the future to sustain the website.
    - Scalability: The ability to develop and increase the features and functionalities of the website without affecting its performance (adding more memory, servers, or disk space to handle more transactions on the website).
    - Correctness: The extent to which the program meets the specifications and achieves the user's objectives.
    - Usability:
        - Easy to use, efficient, and accessible.
        - Appropriately assisting users in filling out mandatory fields in case of invalid input.
        - Tracking documentation, activities, and feedback.

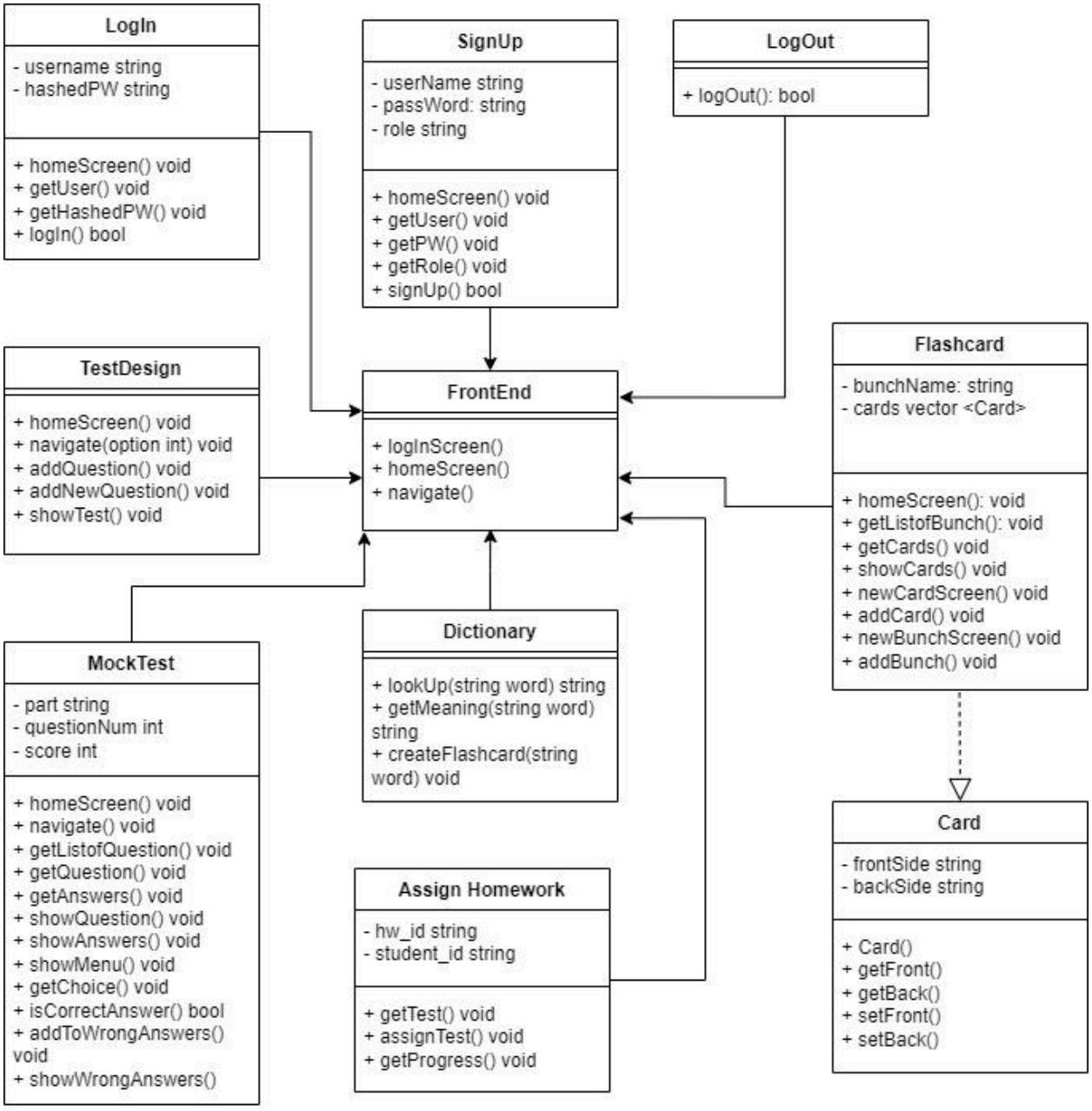
3. Use-Case Model



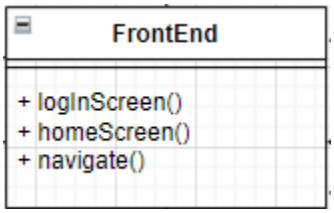
4. Logical View



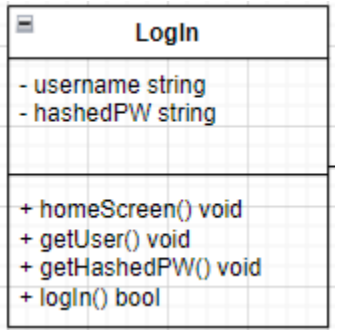
4.1 Component: Front-end



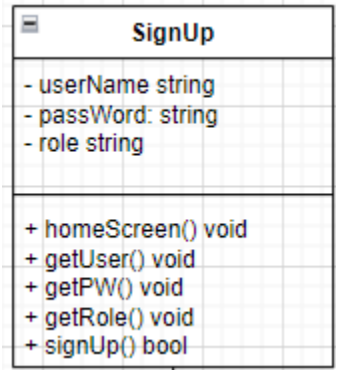
- Front-end: class is a combination of some compositions which each is responsible for a single or a couple of features.
- This class has 3 methods:
    - logInScreen: show the initial screen on opening where users can choose either log in or sign up.
    - homeScreen: show the home screen with feature buttons.
    - navigate: navigate program to a specific feature when user clicks on the corresponding button on homeScreen.



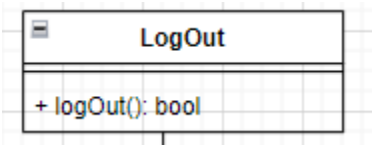
- Class **LogIn**: responsible for login feature
- Attributes: username, and hashedPW which is password stored after hashed by a hash function for security purpose.
- Methods:
    - homeScreen: show login screen.
    - getUser: store the value of username which user enters into username variable.
    - getHashedPW: store the hashed value of password which user enters into hashedPW variable.
    - logIn: if username and password match an entry in the database, return a value to allow the user to log in.



- Class **SignUp**: responsible for sign up feature
- Attribute:
    - username
    - passWord
    - role: the role user wants to sign up for, which is student or teacher.
- Methods:
    - homeScreen: show sign-up screen.
    - getUser: get and store the input username.
    - getPW: get and store the input password
    - getRole: get and store the input role
    - signUp: if user information is valid, return a value to allow the user to create a new account.

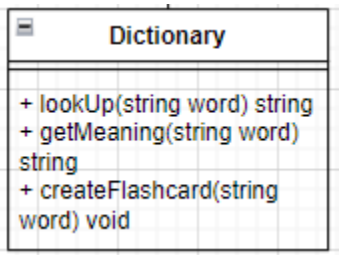


- Class **LogOut**: responsible for logout feature
- Methods:
    - logOut: when the user clicks on the log-out button, terminate user connection and set a flag value.



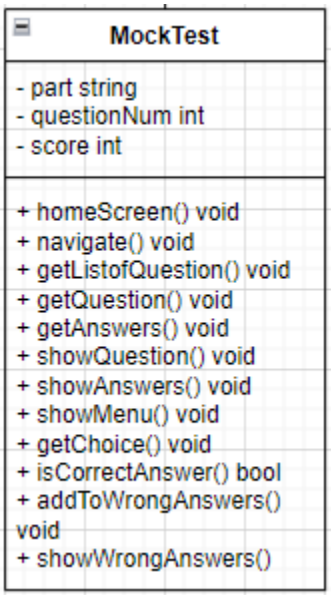
Student's Features

- Class **Dictionary**: responsible for looking up word feature
- Methods:
    - lookUp(word): look up the word in the dictionary and return the meaning of the word.
    - getMeaning(word): get the meaning of the word from the dictionary.
    - createFlashcard(word): create a flashcard directly in the dictionary (which calls another method)



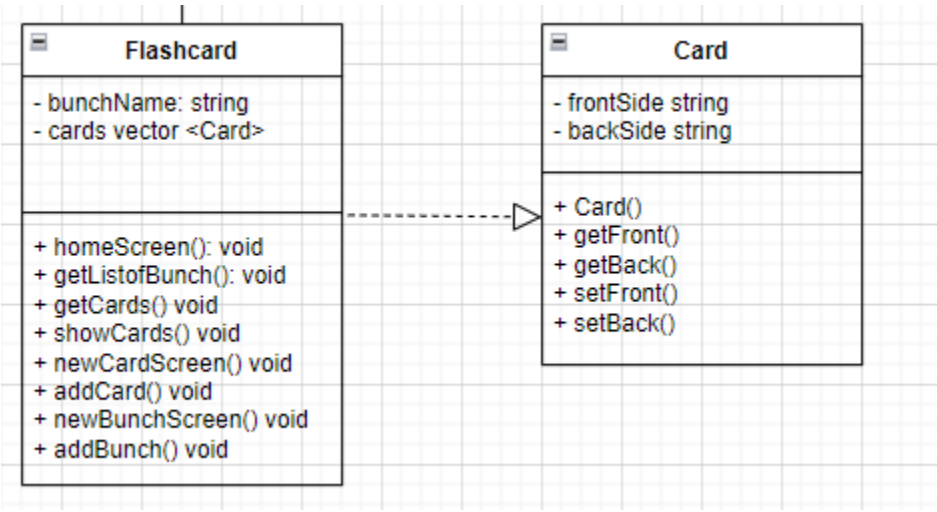
- Class **MockTest**: responsible for mock test feature
- Attributes:
    - part: part of the test: part 1, part 2 e.g.
    - questionNum: the question number
    - score: total score
- Methods:
    - homeScreen(): show the feature screen.
    - getListofQuestion: get the list of questions that belong to the chosen part of the test.
    - getQuestion: get corresponding questions from the database and show by using showQuestion.

- getAnswers: get answers of each question from the database and show by using showAnswer.
- showMenu: show all question numbers and move to a question when users click on the corresponding number.
- When users choose an answer, the getAnswer method gets the choice and checks if it is correct by using isCorrectAnswer(). If it's wrong, call addToWrongAnswers.
- showWrongAnswers: show all questions with incorrect answer choice.
- navigate: navigate to other features.



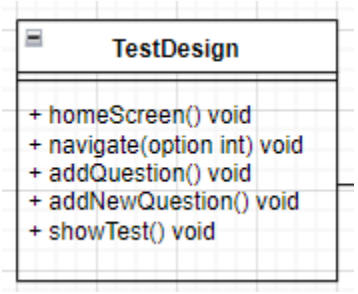
- Class **Flashcard** and **Card**: responsible for Flashcard feature
- Class **Flashcard**:
- Attribute:
  - bunchName: name of a flashcard set.
  - cards: all flashcards that belong to this bunch.
- Methods
  - homeScreen: show feature screen.
  - getListofBunch: get all existing bunches of flashcards from DB to show on homeScreen.
  - getCards: get cards in a bunch.
  - showCards: show cards on the screen.
  - newCardScreen: navigate to the screen of adding new flashcard feature.
  - addCard: add new flashcard.
  - newBunchScreen: navigate to the screen of adding new card bunch feature.
- Class **Card**: contains flashcard information and behaviors
- Attribute:
  - A flashcard has 2 sides: the front and the back. Usually, the front side is the spelling of the word, the back side is the meaning.
- Methods:
  - Card: default constructor, create a new card.
  - getFront: get the text on the front side.

- getBack: get the text on the back side.
- setFront: set value on the front side.
- setBack: set value on the back side.

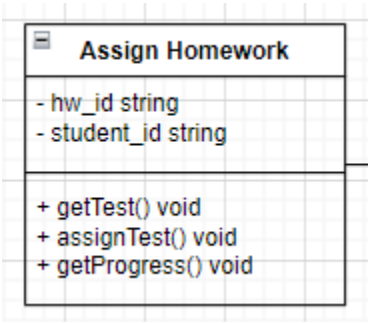


Teacher's Features:

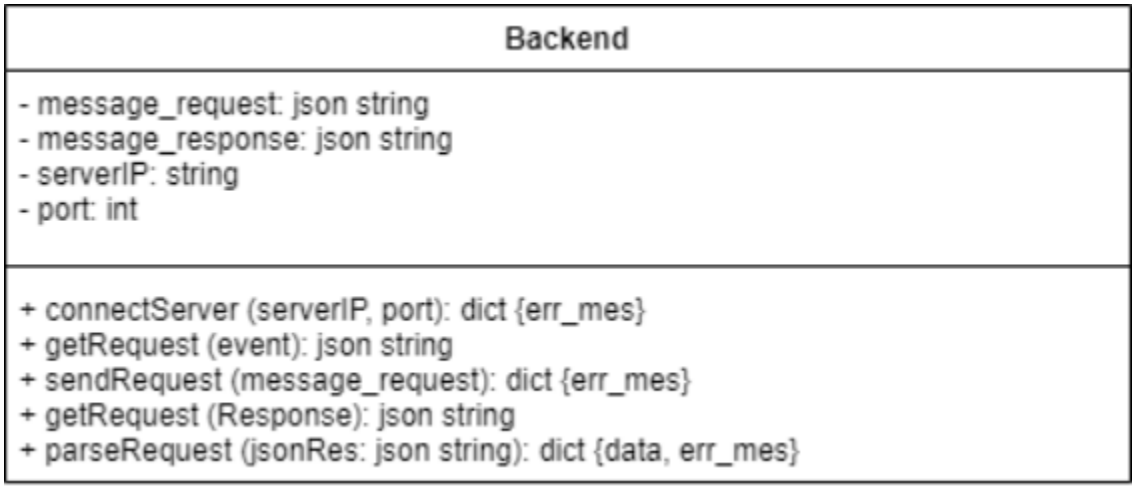
- Class **TestDesign**: responsible for the test designing feature.
- Methods:
  - homeScreen
  - navigate(int): users can add a new or an existing question to the test, depending on the parameter, navigate to the corresponding type of question.
  - addQuestion: add an existing question in the database to the test.
  - addANewQuestion: create a new question, store in the database and add to the test.
  - showTest: show all questions of the test.



- Class **AssignHomeword**: responsible for the homework assigning feature.
- Attribute:
  - hw_id: the identifier of the homework test.
  - student_id: the assigned students' identifier.
- Methods:
  - getTest: get the test to assign
  - assignTest: assign test to choose students.
  - getProgress: return the completed percentage of students.



4.2 Component: Back-end



The Back-end component of the client will handle the events that users perform on the UI/UX and send HTTP/HTTPS requests to the server for functionalities that require data provided from the Database. Additionally, the Back-end will also receive the corresponding HTTP/HTTPS responses to the requests sent and handle cases where no response is received within the allowed time frame. After receiving the response, the back-end will display the resulting data to the user through the UI/UX.

Note: “dict” is a dictionary data type; “err_mes” means error message. “dict {data, err_mes}” means a dictionary with two keys: data and err_mes.

Attributes of the class:

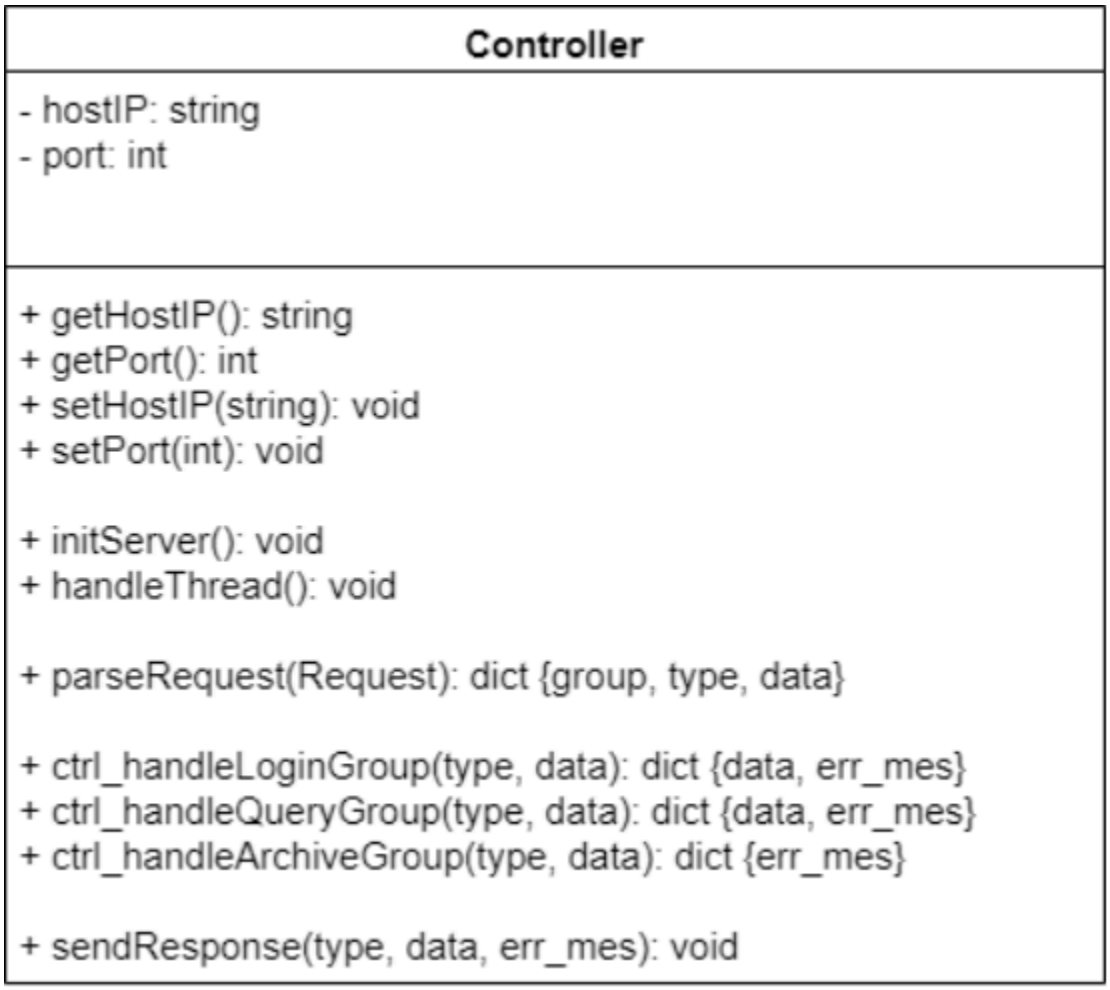
- message_request: the message sent to the server.
- message_response: JSON string after parsing the response message returned from the server.
- serverIP: IP of the server to connect to.
- port: the port used by the server.

Methods of the class:

- The connectServer method is used to connect to the server to transmit and receive data.
- The getRequest method is used to structure the message request after receiving a request from the user.
- The sendRequest method is used to send the message request to the server.

- The getResponse method receives the server's response message in JSON string format and saves it to the message_response attribute, then uses the parseResponse function to analyze the data fields and recreate a complete JSON string with a clearer structure for each data field requested by the user, and returns it for the UI/UX to display to the user.

4.3 Component: Controller



The Controller component of the server will receive HTTP/HTTPS requests from the client, analyze and extract data from the requests to serve as input data for functions in the model component. The functions used in the Model component will be determined by the results of the Controller's analysis of the requests.

In addition to receiving requests, the Controller also handles the results of those requests after they are processed. When data is received from the View, the Controller will integrate this data into an HTTP/HTTPS response and send it back to the client's Back-end. The Controller is also responsible for initializing the server and managing access flow.

Class attributes:

- hostIP: the IP address of the machine selected to act as the server.
- port: the port that the server occupies on the machine.

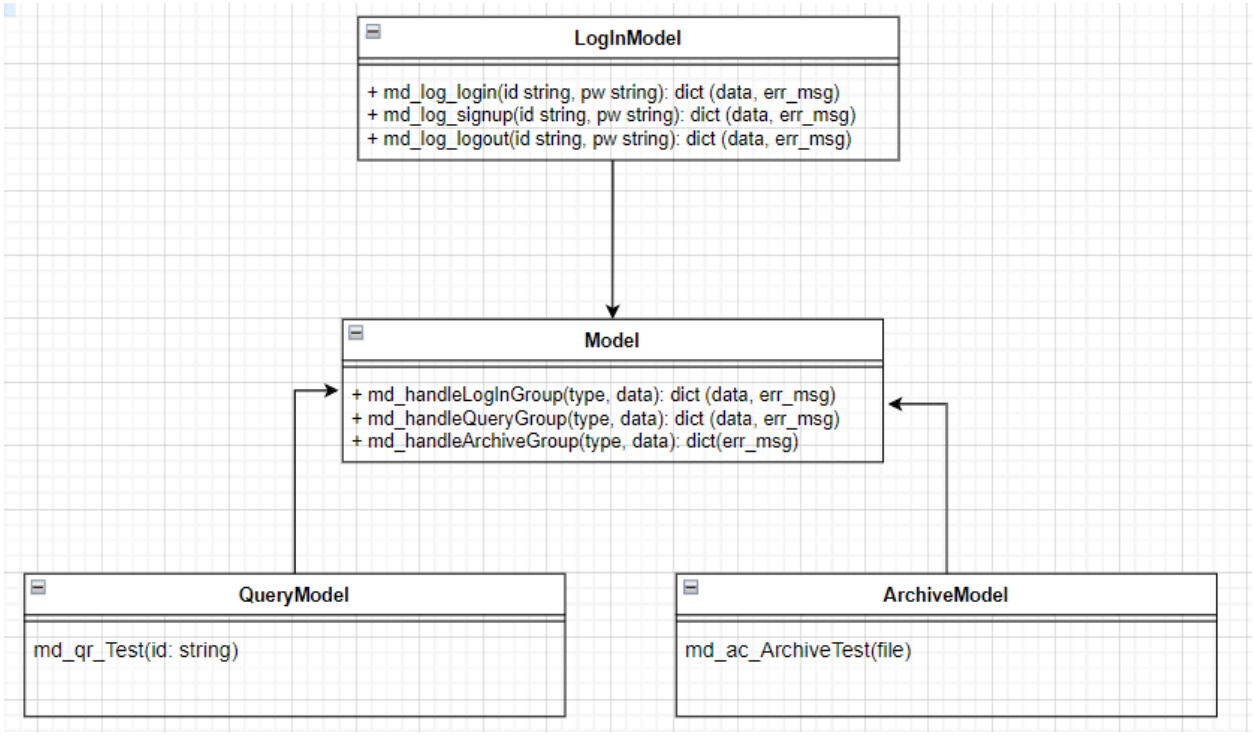
Class methods:

- getHostIP(): returns the value of the hostIP attribute.
- getPort(): returns the value of the port attribute.
- setHostIP(string): sets the value of the hostIP attribute.
- setPort(int): sets the value of the port attribute.
- initServer(): initializes the server with the hostIP and port attributes.
- handleThread(): handles multi-threading for the server.
- parseRequest(Request): analyzes the incoming Request and returns a dictionary containing information such as group name, function type, and associated data.
- ctrl_handleLoginGroup(type, data): handles Requests related to Login, Sign-in, and Log-out functions; type is the name of the function to be processed and data is the associated data.
- ctrl_handleQueryGroup(type, data): handles Requests related to functions serving Student users; type is the name of the function to be processed and data is the associated data.
- ctrl_handleArchiveGroup(type, data): handles Requests related to functions serving Teacher users; type is the name of the function to be processed and data is the associated data.
- sendResponse(type, data, err_mes): sends a Response to the client with information provided from the input parameters.

4.4 Component: Model

The Model component includes methods that interact with the Database to query data based on user requests. After processing the data, the Model sends it to the View component for further handling.

Class: Model



Methods:

- md_handleLoginGroup(type, data): Handles login, sign-up, and logout functions using methods from the LoginModel module. The type parameter specifies the function to be processed, and data contains the accompanying data.
- md_handleQueryGroup(type, data): Handles student-related functions using methods from the QueryModel module. The type parameter specifies the function to be processed, and data contains the accompanying data.
- md_handleArchiveGroup(type, data): Handles teacher-related functions using methods from the ArchiveModel module. The type parameter specifies the function to be processed, and data contains the accompanying data.

Modules and Their Methods:

LoginModel Module

- md_log_Login(id: string, pw: string): Handles the login function. Queries the database for username and password, compares them with the input data, and calls updateActivityLog().
- md_log_SignUp(id: string, pw: string): Handles the sign-up function. Queries the database to validate the account.
- md_log_Logout(id: string, pw: string): Handles the logout function and calls updateActivityLog().

QueryModel Module

- md_qr_Test(id: string): Queries the database for test data.

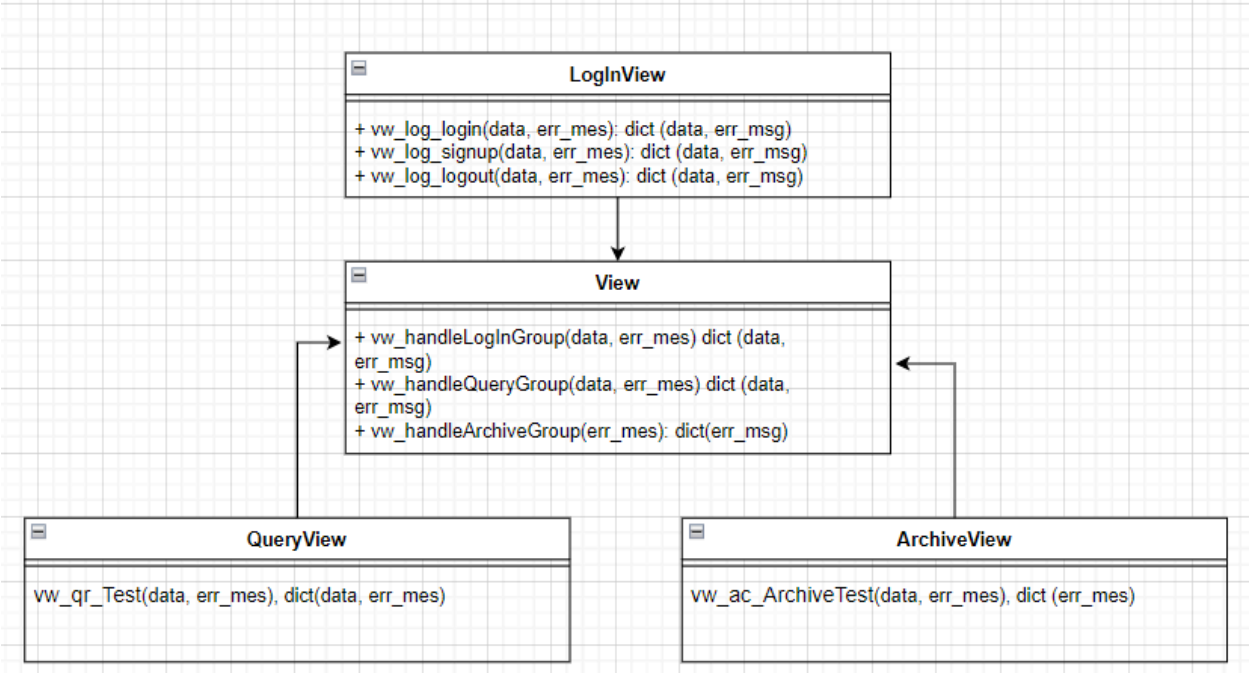
ArchiveModel Module

- md_ac_ArchiveTest(file): Updates reading tests in the database.

4.5 Component: View

The purpose of the View component is to format the processed data from the Model into predefined templates and return the results to the Controller. The View ensures that the server can control the output data and that the client always receives data as intended by the server, preventing accidental exposure of important information.

Class: View



Methods:

- vw_handleLoginGroup(data, err_mes): Formats query data and returns results for login, sign-up, and logout functions.
- vw_handleQueryGroup(data, err_mes): Formats query data and returns results for student-related functions.
- vw_handleArchiveGroup(data, err_mes): Formats query data and returns results for teacher-related functions.

Modules and Their Methods:

LoginView Module

- vw_log_Login(data, err_mes): Formats query data and returns results for the login function. Displays

an error message if err_mes exists.

- vw_log_Signup(data, err_mes): Formats query data and returns results for the sign-up function. Displays an error message if err_mes exists.
- vw_log_Logout(data, err_mes): Formats query data and returns results for the logout function. Displays an error message if err_mes exists.

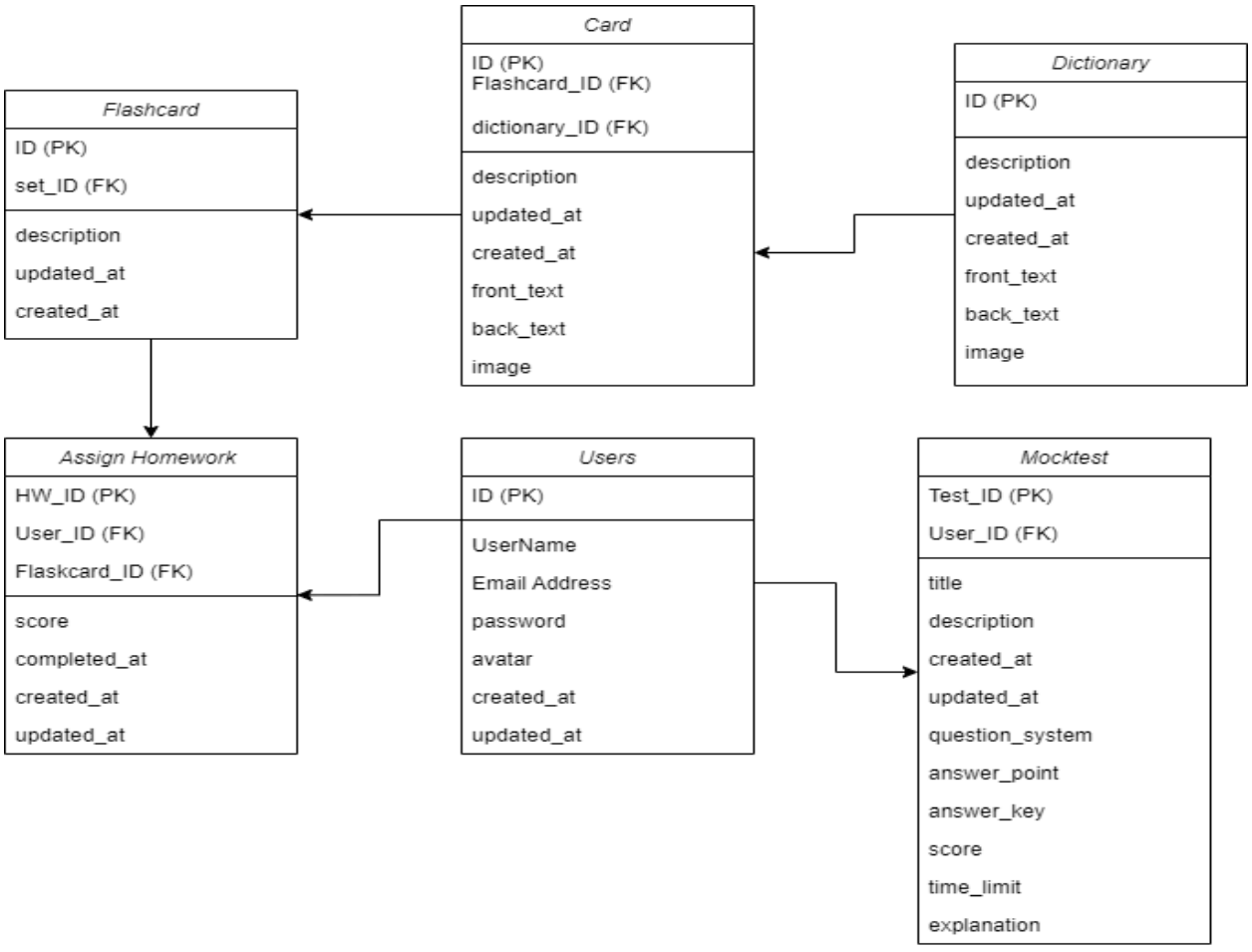
QueryView Module

- vw_qr_Test(data, err_mes): Formats query data and returns results for the test query function. Displays an error message if err_mes exists.

ArchiveView Module

- vw_ac_ArchiveTest(data, err_mes): Formats data and returns the status of the test archiving function. Displays an error message if err_mes exists.

4.6 Component: Database



Database Responsibilities:

- Acting as the central repository for all application data, it stores information about users, vocabulary, lessons, exercises, mock tests, user progress, and more.
- The database enforces data integrity by defining data types, constraints, and relationships between tables. This ensures data consistency and reduces errors.
- Unlike temporary memory used by the backend, data stored in the database persists even after the application restarts. This allows users to maintain their progress and access information across sessions.
- The database provides efficient mechanisms for querying and retrieving specific data based on user requests or backend logic.

Connections with other components:

- Backend is the server-side component that interacts with the database in order to house the application logic and business rules. It utilizes the database in several ways:
  + When a user logs in, the backend retrieves user credentials (username, password) and verifies them against the database (User Authentication).
  + The backend retrieves data from the database based on user requests (displaying vocabulary lists, presenting lessons, recording exercise results).
  + The backend modifies data in the database based on user actions (creating new flashcards, updating user profiles, storing mock test results).
- Controller acts as an intermediary between user requests and the backend. It receives user input (e.g., login form submission, button clicks) and interacts with the backend to perform the desired actions.

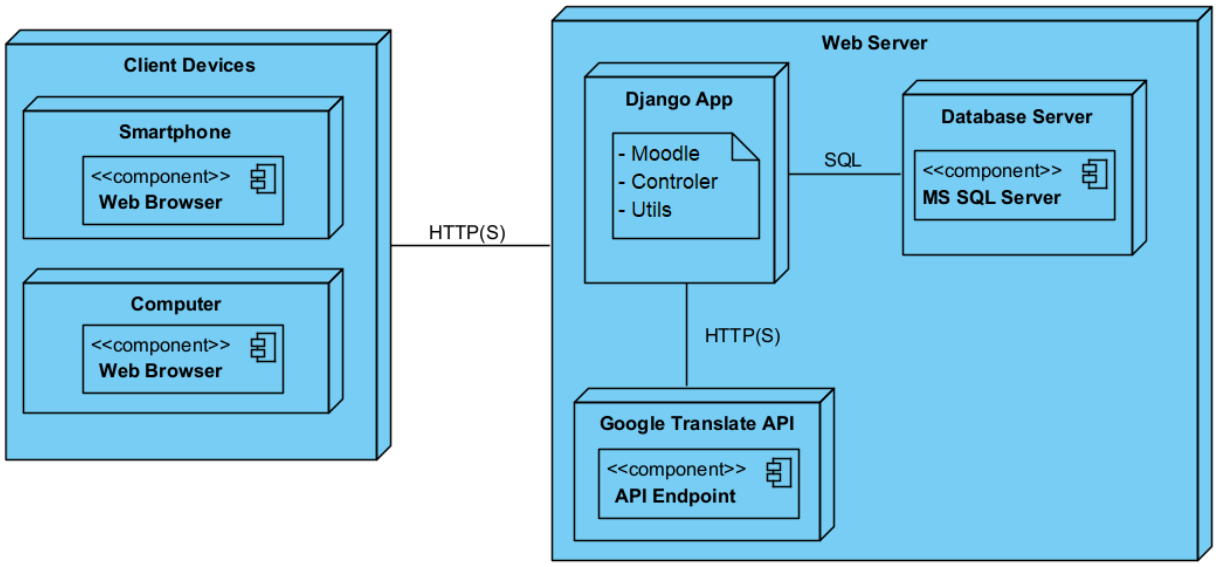
  The controller doesn't directly interact with the database. It relies on the backend to handle all database operations.

Communication Channels:

- Backend to Database: The backend server uses the database's provided API (MySQL Connector/J) to interact with the database.
- User to Controller: Users interact with the application through their web browser, sending requests and receiving responses via HTTP/HTTPS protocols.
- The communication between controller and backend typically happens within the same server environment and doesn't involve specific protocols like HTTP/HTTPS. They can use function calls, shared memory, or messaging frameworks depending on the chosen architecture.

5. Deployment

The deployment architecture of Bilinguo involves the allocation of software components to the appropriate hardware and software environments, along with the integration of external services such as the Google Translate API. This section outlines the deployment strategy, including hardware and software requirements, scalability considerations, potential bottlenecks, and the integration of the Google Translate API.



UML Diagram Deployment

5.1. Hardware Requirements:

- Web Server: A robust web server with ample processing power, memory, and storage is necessary to efficiently handle HTTP requests from users. This server should be capable of supporting concurrent connections and database operations.
- Database Server: Bilinguo relies on a relational database management system (RDBMS) for data storage. Therefore, the database server needs to be resilient and scalable to accommodate the increasing volume of data and user interactions.
- Optional Load Balancer: In high-traffic scenarios, deploying a load balancer can help distribute incoming web requests across multiple web servers, enhancing performance and reliability.

5.2. Software Requirements:

- Operating System: The web server and database server should run on a stable and secure operating system.
- Web Server Software: Bilinguo can be deployed using web server software such as Apache HTTP Server. These servers handle HTTP requests and serve static and dynamic content to users.
- Database Management System: MS SQL Server can be used as the backend database management system for storing user data, language resources, and other application-related information.
- Application Framework: Bilinguo is built using the Django framework.
- Programming Language: The backend of Bilinguo is primarily developed using Python, leveraging its simplicity, readability, and extensive ecosystem of libraries and frameworks.

5.3. Integration of Google Translate API:

- API Key Management: Securely manage and store the API keys required to access the Google Translate API. This typically involves setting environment variables on the web server to keep the

keys secure.

- API Requests: Implement functions in the Bilinguo application to send translation requests to the Google Translate API. This involves constructing appropriate HTTP requests and handling responses, including error handling and retry logic for robustness.
- Rate Limiting: Monitor and respect the rate limits imposed by the Google Translate API to avoid service disruptions. Implementing caching strategies for frequently translated phrases can reduce the number of API requests and improve response times.

5.4. Scalability Considerations:

- Bilinguo should be designed to scale horizontally to accommodate increasing user traffic and data volume. This can be achieved by adding more web server instances behind a load balancer and employing database replication or sharding techniques to distribute the database workload.
- Monitoring tools should be integrated into the deployment architecture to track system performance metrics, detect bottlenecks, and proactively scale resources as needed.

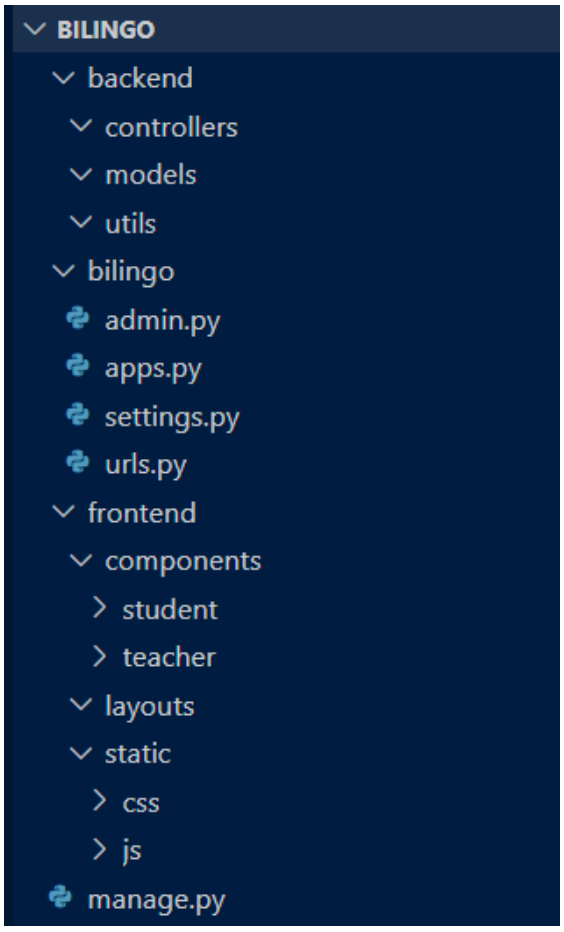
5.5. Potential Bottlenecks:

- Database Performance: High database load or inefficient queries can lead to performance bottlenecks. Optimizing database schema, indexing frequently accessed columns, and caching query results can mitigate these issues.
- Network Latency: In distributed deployments, network latency between components can impact overall system performance. Minimizing the number of network round-trips and optimizing data transfer protocols can help reduce latency.
- API Rate Limits: Exceeding the rate limits of the Google Translate API can lead to service disruptions. Implementing local caching and optimizing the frequency of API calls can help mitigate this risk.
- Single Points of Failure: Critical components like the database server or load balancer should be redundantly configured to avoid single points of failure. Implementing failover mechanisms and disaster recovery plans can ensure high availability and fault tolerance.

6. Implementation View

This section outlines the technical details of how the logical components described in the Logical View section are implemented. It includes information about the folder structures, programming languages, frameworks used to build the software.

The following is a high-level representation of the Bilinguo project directory structure:



- **backend** directory: Contains the server-side code for Bilinguo.
  - **controllers** directory: Stores the implementation files for controller classes.
  - **models** directory: Houses the implementation files for model classes.
  - **utils** directory: Includes utility functions used throughout the backend.
- **frontend** directory: Contains the client-side code for Bilinguo's user interface.
  - **components** directory: Organizes the UI components by user role (student, teacher) and functionality.
  - **layouts** directory: Stores the layout templates for the application.
  - **static** directory: Holds static assets like CSS and JavaScript files.
- **manage.py**: The Django project management script used for various tasks like running the development server and database migrations.
- **bilinguo** directory: The Django application package for Bilinguo.
  - **admin.py**: Configures the Django admin interface for Bilinguo models.
  - **apps.py**: Defines the Django application configuration for Bilinguo.
  - **settings.py**: Contains critical project settings like database credentials, secret keys, and application paths.
  - **urls.py**: Defines the URL routing patterns for the Bilinguo application.