Penn State World Campus

MTG Generative AI-Powered chatbot

Final

Cameron Beebe

8-9-2024

Table of Contents

Introduction

The Generative AI-Powered MTG Deck Building Framework is designed to assist users in creating and managing Magic: The Gathering decks using an AI-powered chatbot. This document provides a comprehensive overview of the system's design and functionality, combining use case models, domain models, and interaction diagrams.

Problem Statement

In the world of Magic: The Gathering, creating effective and competitive decks can be a daunting task for players. The primary business requirement is to provide a user-friendly platform that utilizes generative AI to assist users in building, managing, and sharing MTG decks. Non-functional requirements include system scalability, performance efficiency, data security, and a seamless user experience across different devices.

Business Requirements

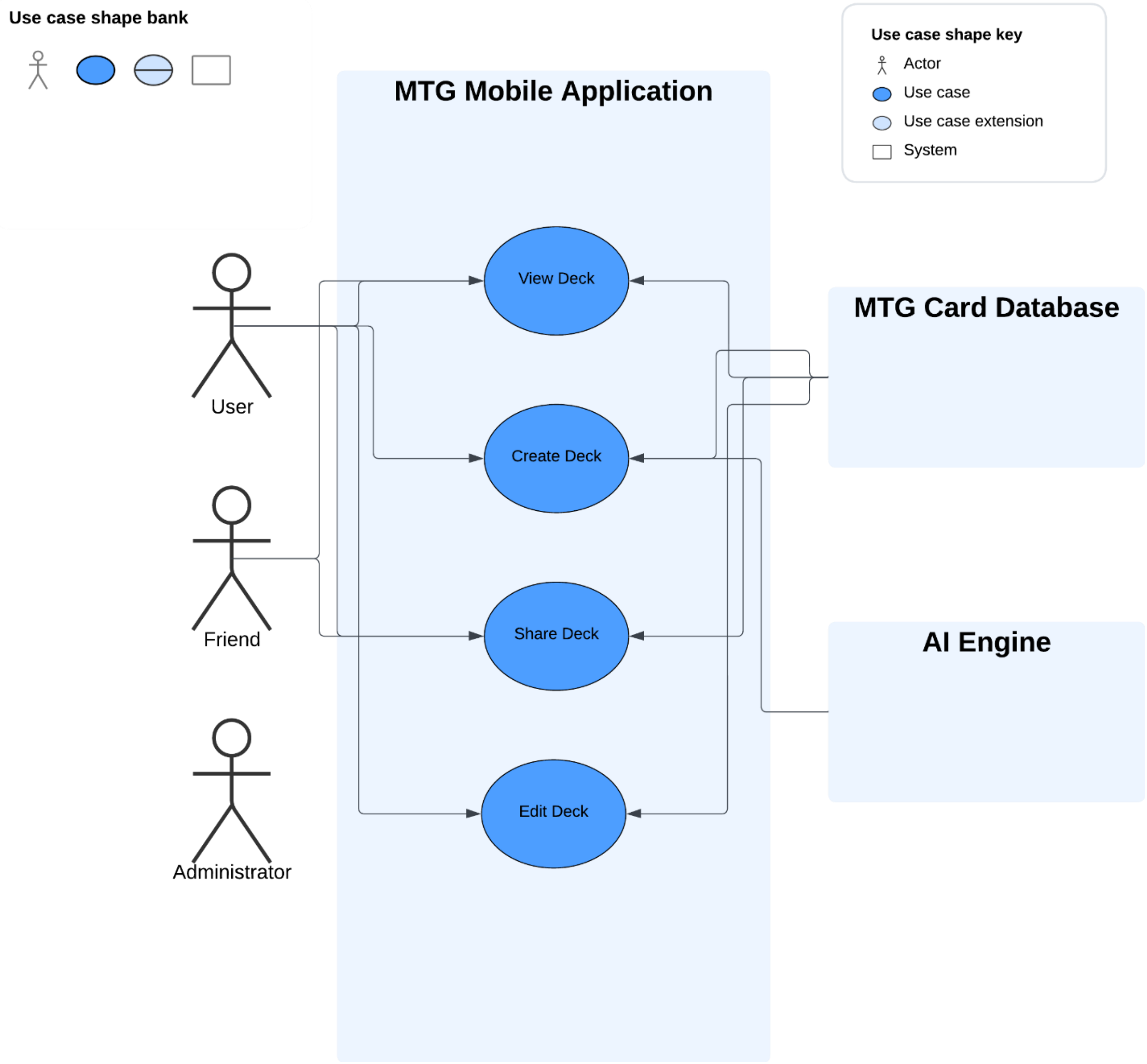
- Provide an AI-powered suggestion system for deck creation.
- Enable users to edit and manage their decks.
- Allow users to view their decks and share them with friends.
- Maintain a user profile system to personalize the experience.

Non-Functional Requirements

- **Performance:** The system should handle multiple simultaneous users efficiently.
- **Scalability:** The platform should scale to accommodate a growing number of users and data.
- **Security:** Ensure data protection through encryption and secure authentication mechanisms.
- **Reliability:** The system should have high availability and reliability to avoid downtime.
- **Usability:** Provide an intuitive and responsive interface for a seamless user experience.

Use Case Model

Use Case Diagram



The Use Case Diagram includes primary actors like Users who create and manage MTG decks and Friends who can view and share these decks. Supporting actors include Admins, who ensure the system runs smoothly, and the AI Engine, which generates deck suggestions. The Database plays a key role in storing user profiles and deck information.

Actor Table

Type	Actor	Goal Description
Primary	User	Create and manage MTG decks
Primary	Friend	View and share decks
Supporting	Admin	Ensure system uptime and security
Offstage	AI Engine	Generate deck suggestions
Supporting	Database	Store user profile and deck information

Use Case Descriptions

Create Deck

- Goal in Context: User successfully creates an MTG deck.
- Primary Actors: User
- Supporting Actors: AI Engine, Database
- Stakeholders and Interest:
  - User: Wants to create a competitive MTG deck.
  - AI Engine: Provides deck suggestions based on user input.
  - Database: Stores the created deck.
- Preconditions:
  - User must have a valid login credential.
  - Users must have an active account in the system.
- Success Guarantee:
  - Users can create and save a new MTG deck without any errors.
- Main Success Scenario:
  - User logs into the system.
  - System verifies login credentials.
  - User navigates to the deck creation section.
  - User inputs preferences and requirements.
  - AI Engine generates a deck.
  - User reviews and saves the deck.
  - System saves the deck in the Database.
- Extensions:
  - User enters invalid login credentials:
    - System displays an error message.
    - User retries login with valid credentials.
  - AI Engine fails to generate deck:
    - System displays an error message.
    - User retries the deck creation process.
  - System fails to save the deck:
    - System displays an error message.
    - User retries saving the deck.

Edit Deck

- Goal in Context: User successfully edits an existing MTG deck.
- Primary Actors: User
- Supporting Actors: Database
- Stakeholders and Interest:

    - User: Wants to update their deck for better performance.
- Preconditions:
    - User must have a valid login credential.
    - Users must have access to an existing deck.
- Success Guarantee:
    - Deck is updated and saved without errors.
- Main Success Scenario:
    - User logs into the system.
    - System verifies login credentials.
    - User selects an existing deck.
    - System saves the updated deck in the Database.
- Extensions:
    - User enters invalid login credentials:
        - System displays an error message.
        - User retries login with valid credentials.
    - System fails to save the updated deck:
        - System displays an error message.
        - User retries saving the deck.

View Deck

- Goal in Context: User views their existing MTG decks.
- Primary Actors: User
- Supporting Actors: Database
- Stakeholders and Interest:
    - User: Wants to view their created decks.
- Preconditions:
    - User must have a valid login credential.
    - Users must have one or more decks in the system.
- Success Guarantee:
    - Users can view their decks without any errors.
- Main Success Scenario:
    - User logs into the system.
    - User navigates to the deck view section.
    - System retrieves and displays the user’s decks from the Database.
- Extensions:
    - User enters invalid login credentials:
        - System displays and error message.
        - User retries login with valid credentials.
    - System fails to retrieve decks:
        - System displays and error message.
        - User contacts support for assistance.

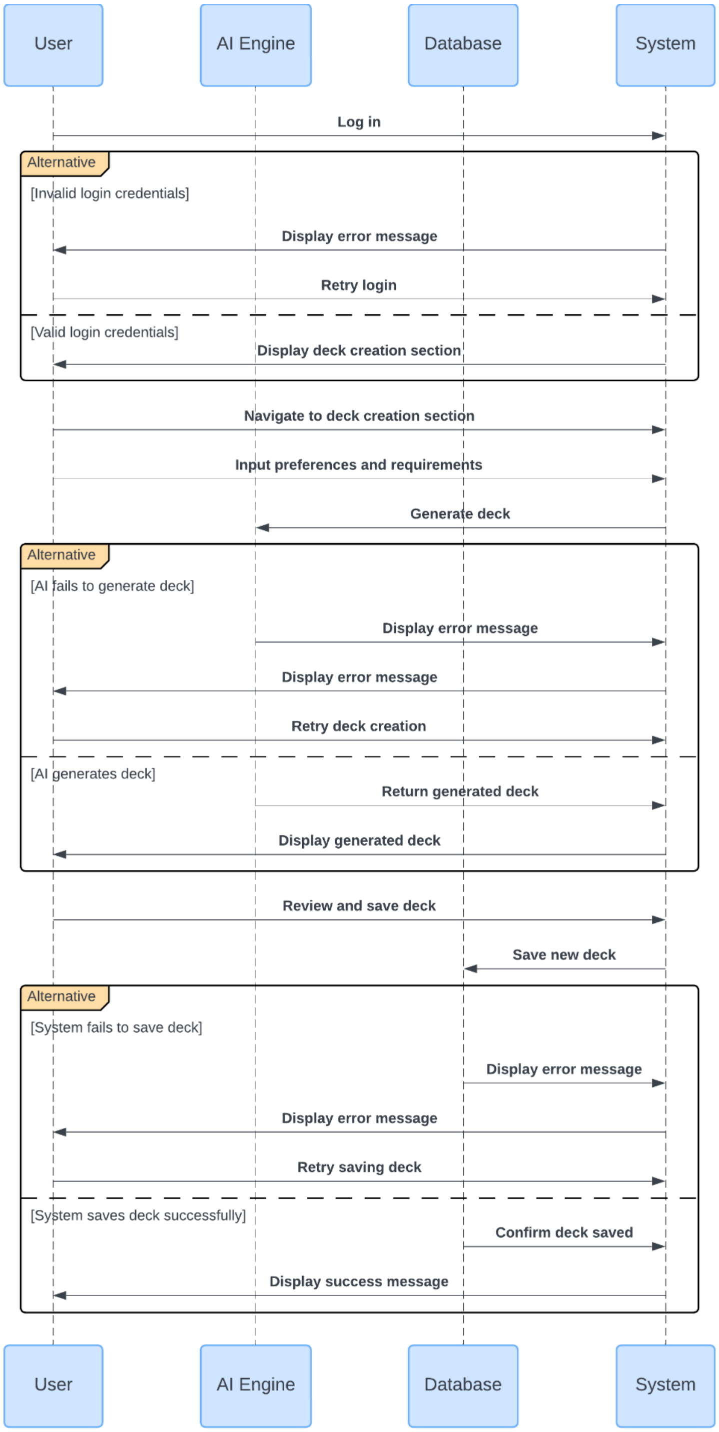
Share Deck

- Goal in Context: User successfully shares an MTG deck with a friend.
- Primary Actors: User
- Supporting Actors: Friend, Database
- Stakeholders and Interest:
    - User: Wants to share their deck with friends.
    - Friend: Wants to view the shared deck.

- Preconditions:
    - User must have a valid login credential.
    - Users must have access to an existing deck.
    - Friend must have an active account in the system.
- Success Guarantee:
    - Deck is shared without any errors.
- Main Success Scenario:
    - User logs into the system.
    - System verifies login credentials.
    - User selects a deck to share.
    - User enters friend's details.
    - System retrieves deck details from the Database and sends them to the friend.
- Extensions:
    - User enters invalid login credentials:
        - System displays an error message.
        - User retries login with valid credentials.
    - Friend does not have an active account:
        - System displays an error message.
        - User invites friend to join the system.
    - System fails to share the deck:
        - System displays and error message.
        - User retries sharing the deck.

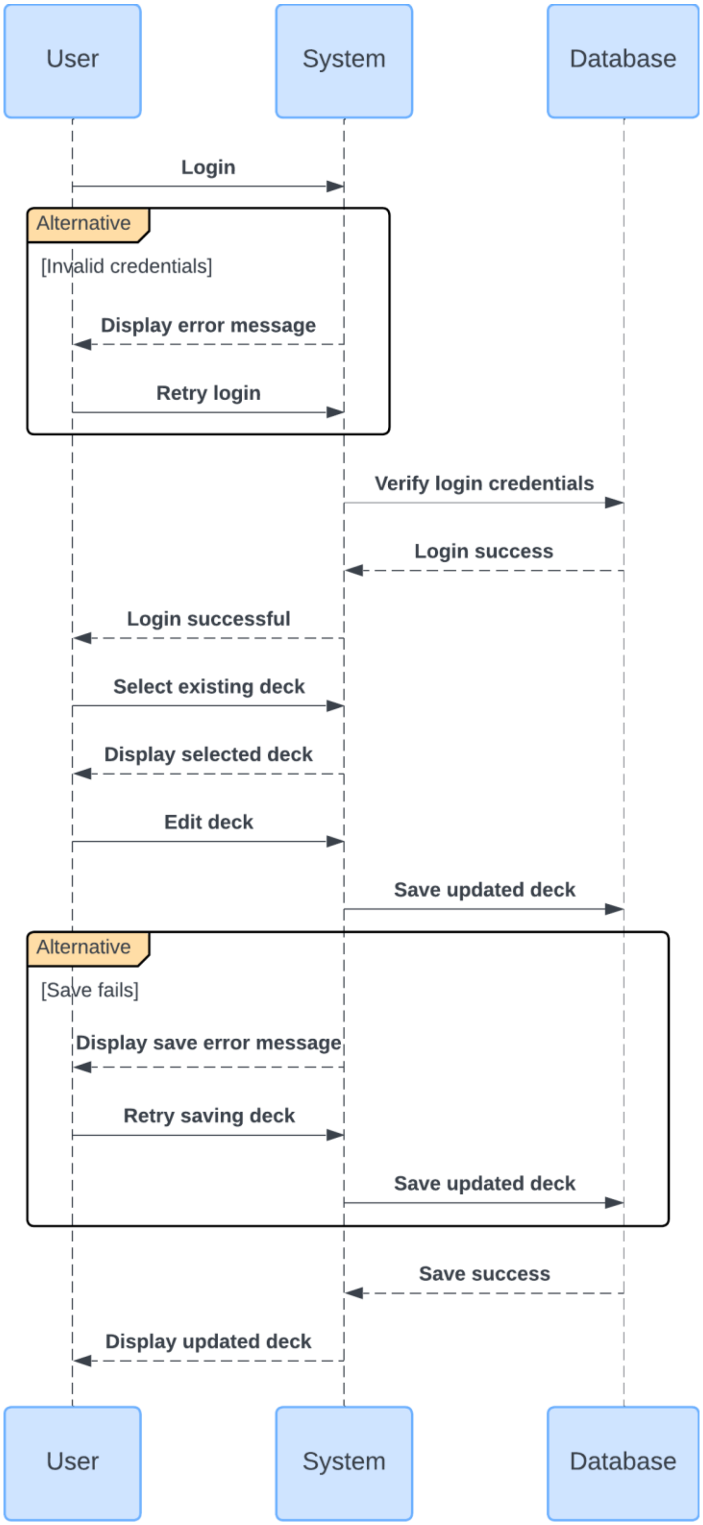
System Sequence Diagrams

Create Deck



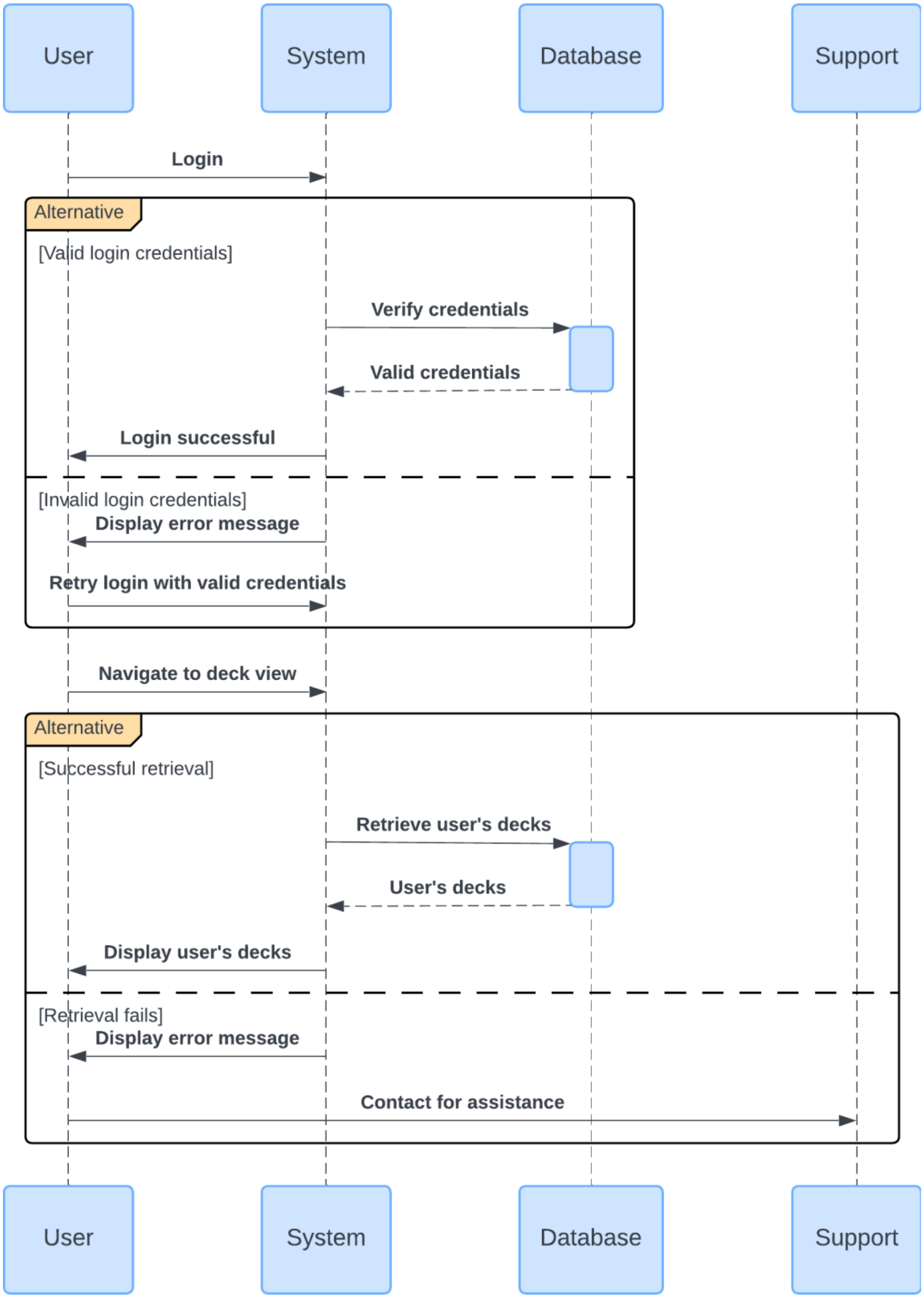
The user initiates the process by logging in and providing deck preferences. The AI Engine then generates a deck based on these preferences, and the system validates the user's credentials. Finally, the created deck is saved in the database, ensuring all data is stored securely.

Edit Deck



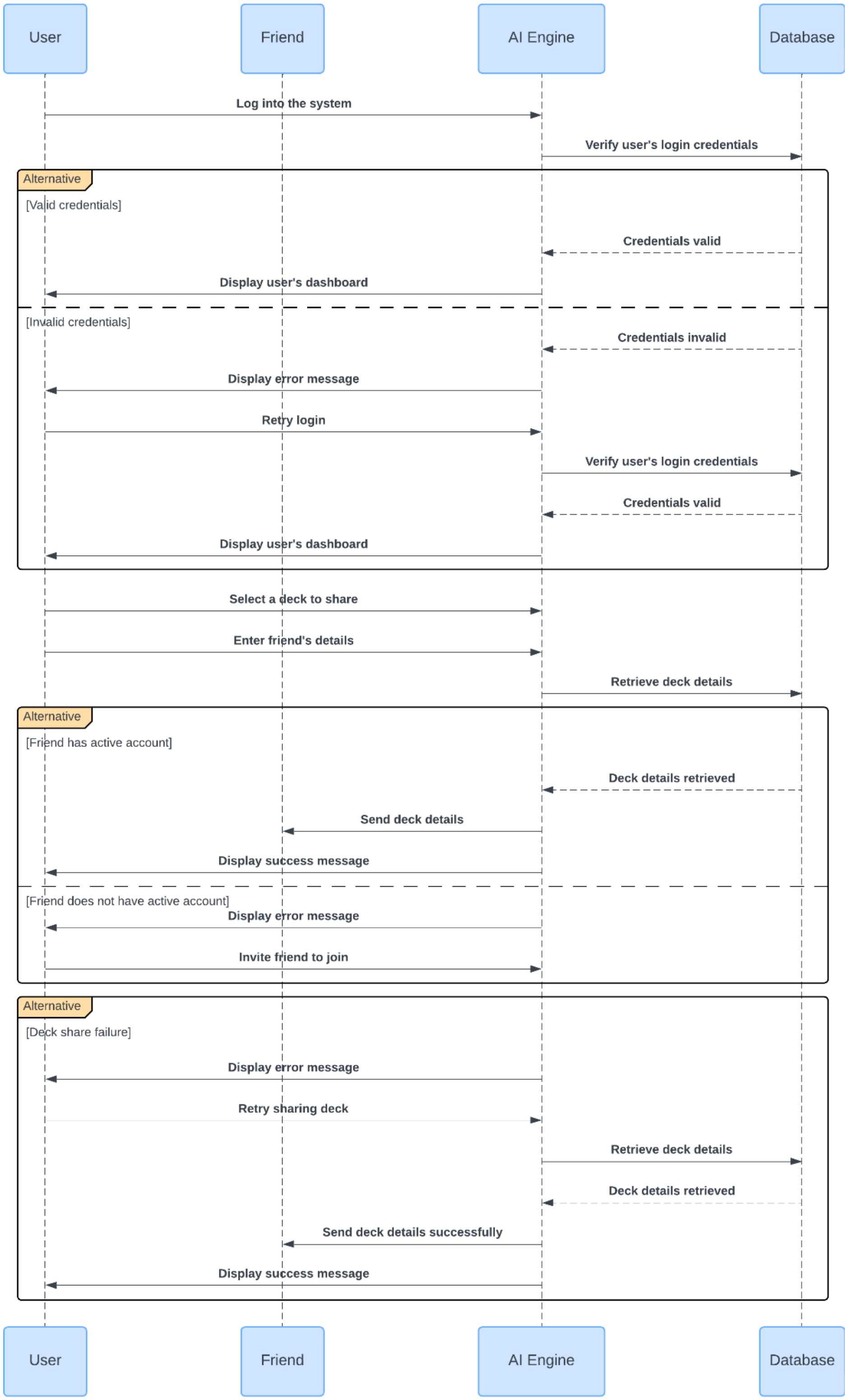
The user logs in and selects an existing deck to update. The system verifies the user's credentials and retrieves the deck from the database. After the user makes the desired changes, the updated deck information is saved back to the database. This ensures the deck is modified correctly and efficiently.

View Deck



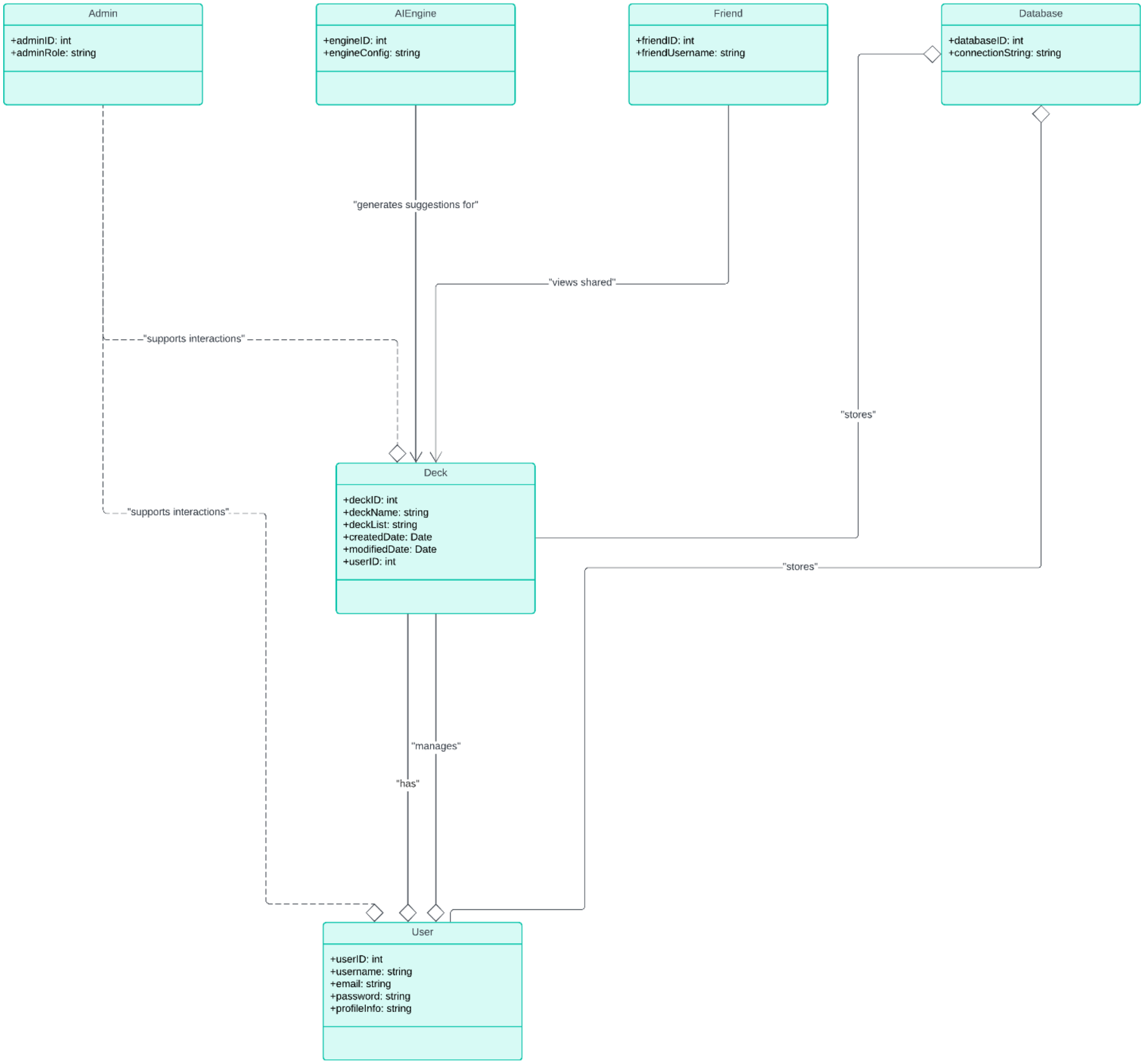
The user logs in and requests to view their decks. The system checks the user's credentials and retrieves the deck information from the database. The requested deck details are then displayed to the user. This process guarantees that users can access their decks seamlessly and accurately.

Share Deck



The user logs in and selects a deck to share. They provide the friend's details, and the system validates these details along with the user's credentials. The system retrieves the deck from the database and sends it to the specified friend. This diagram ensures that deck sharing is done securely and efficiently.

Domain Model



This class diagram represents the core components of the deck management system. At the center is the User class, with attributes like userID, username, email, password, and profileInfo. Each user can manage multiple decks, as shown by the one-to-many relationship between User and Deck. The Deck class, with its own attributes like deckID, deckName, deckList, createdDate, modifiedDate, and userID, connects back to the user. Each deck interacts with a Database, represented by a one-to-one relationship. The Database class has attributes like databaseID and connectionString. Decks can also be shared with friends, illustrated by the many-to-many relationship with the Friend class, which includes friendID and friendUsername. To enhance deck functionality, I've integrated an AIEngine that generates suggestions for decks. This relationship is shown as an aggregation, indicating that the AIEngine plays a crucial role in improving the deck but is not dependent on it. Lastly, the Admin class, with adminID and adminRole attributes, supports interactions with the decks ensuring the system runs smoothly and efficiently.

System Sequence Contracts

generateDeck(userID: String)

- **Field Example**
    - **Name of Operation:** generateDeck(userID: String)
    - **Responsibilities:** Generate a deck based on user preferences and input.
    - **Type:** System
    - **Cross Reference:** Use Cases: Create Deck
    - **Notes:** Ensure userID is validated before generating a deck.
    - **Exceptions:** If userID is not valid, indicate that it was an error.
    - **Output:** A new MTG deck.
    - **Pre-conditions:**
        - The userID exists in the system.
    - **Post-conditions:**
        - A new deck is created and associated with the user.
        - The new deck is saved in the Database.

updateDeck(userID: String, deckID: String, deckData: DeckData)

- **Field Example**
    - **Name of Operation:** updateDeck(userID: String, deckID: String, deckData: DeckData)
    - **Responsibilities:** Update an existing deck.
    - **Type:** System
    - **Cross Reference:** Use Cases: Edit Deck
    - **Notes:** Ensure deckData is valid before updating.
    - **Exceptions:** If userID or deckData is not valid, indicate that it was an error.
    - **Output:** Confirmation of deck update.
    - **Pre-conditions:**
        - The userID exists in the system.
        - The deckID exists in the system.
        - The deckData provided is valid.
    - **Post-conditions:**
        - The existing deck is updated and saved in the Database.

viewDeck(userID: String, deckID: String)

- **Field Example**
    - **Name of Operation:** viewDeck(userID: String, deckID: String)
    - **Responsibilities:** Retrieve and display the deck associated with the given deckID.
    - **Type:** System
    - **Cross Reference:** Use Cases: View Deck
    - **Notes:** Ensure the deckID is validated before attempting to retrieve the deck.
    - **Exceptions:** If deckID is not valid, indicate that it was an error.
    - **Output:** The requested deck.
    - **Pre-conditions:**
        - The userID exists in the system.

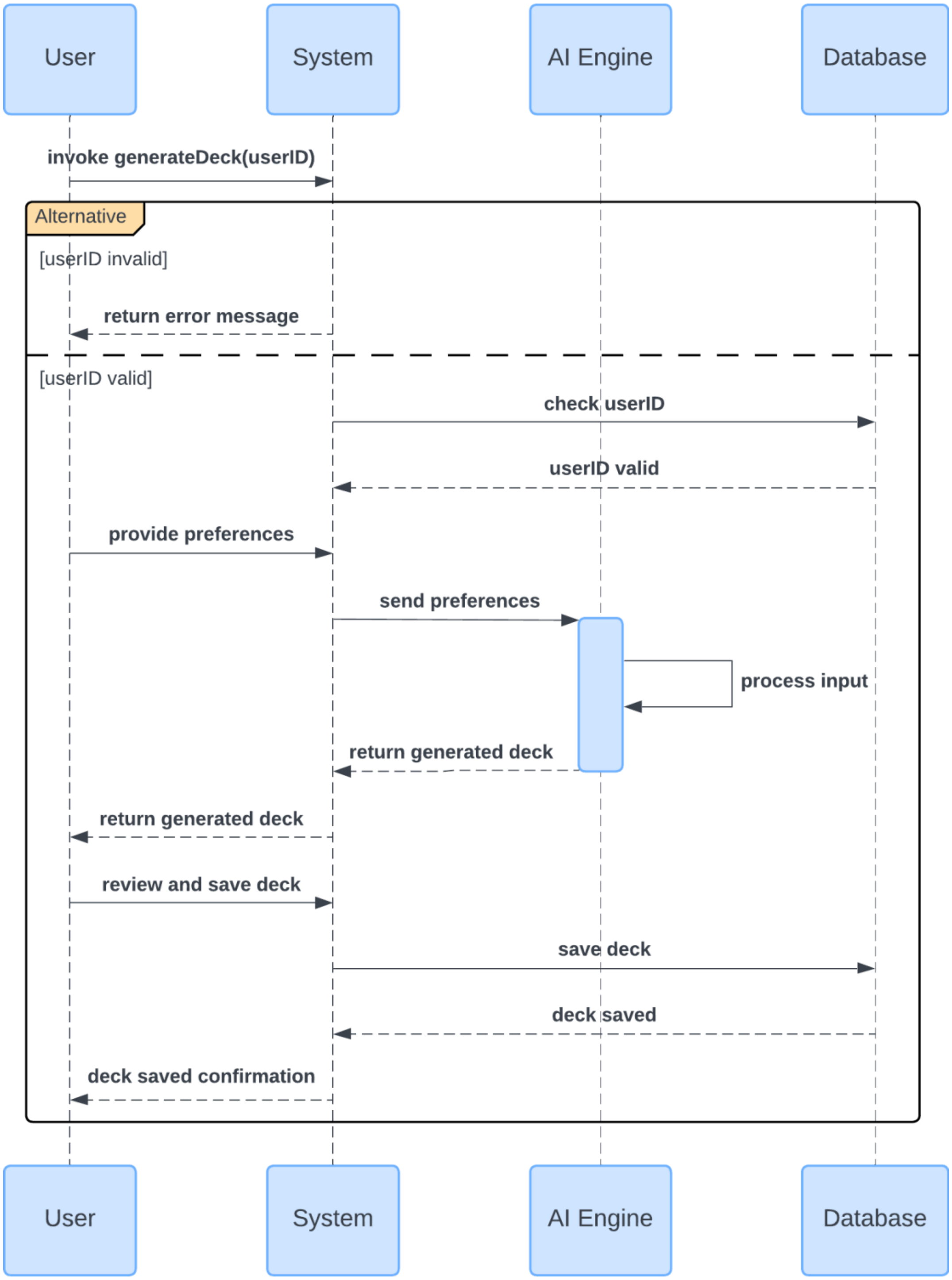
    - The deckID exists in the system.
  - **Post-conditions:**
    - The deck is retrieved from the Database and displayed to the user.

shareDeck(userID: String, deckID: String, friendID: String)

- **Field Example**
  - **Name of Operation:** shareDeck(userID: String, deckID: String, friendID: String)
  - **Responsibilities:** Share the deck with a specified friend.
  - **Type:** System
  - **Cross Reference:** Use Cases: Share Deck
  - **Notes:** Ensure friendID is valid before sharing.
  - **Exceptions:** If friendID is not valid, indicate that it was an error.
  - **Output:** Confirmation of deck sharing.
  - **Pre-conditions:**
    - The userID exists in the system.
    - The deckID exists in the system.
    - The friendID exists in the system.
  - **Post-conditions:**
    - The deck is retrieved from the Database and shared with the specified friend.

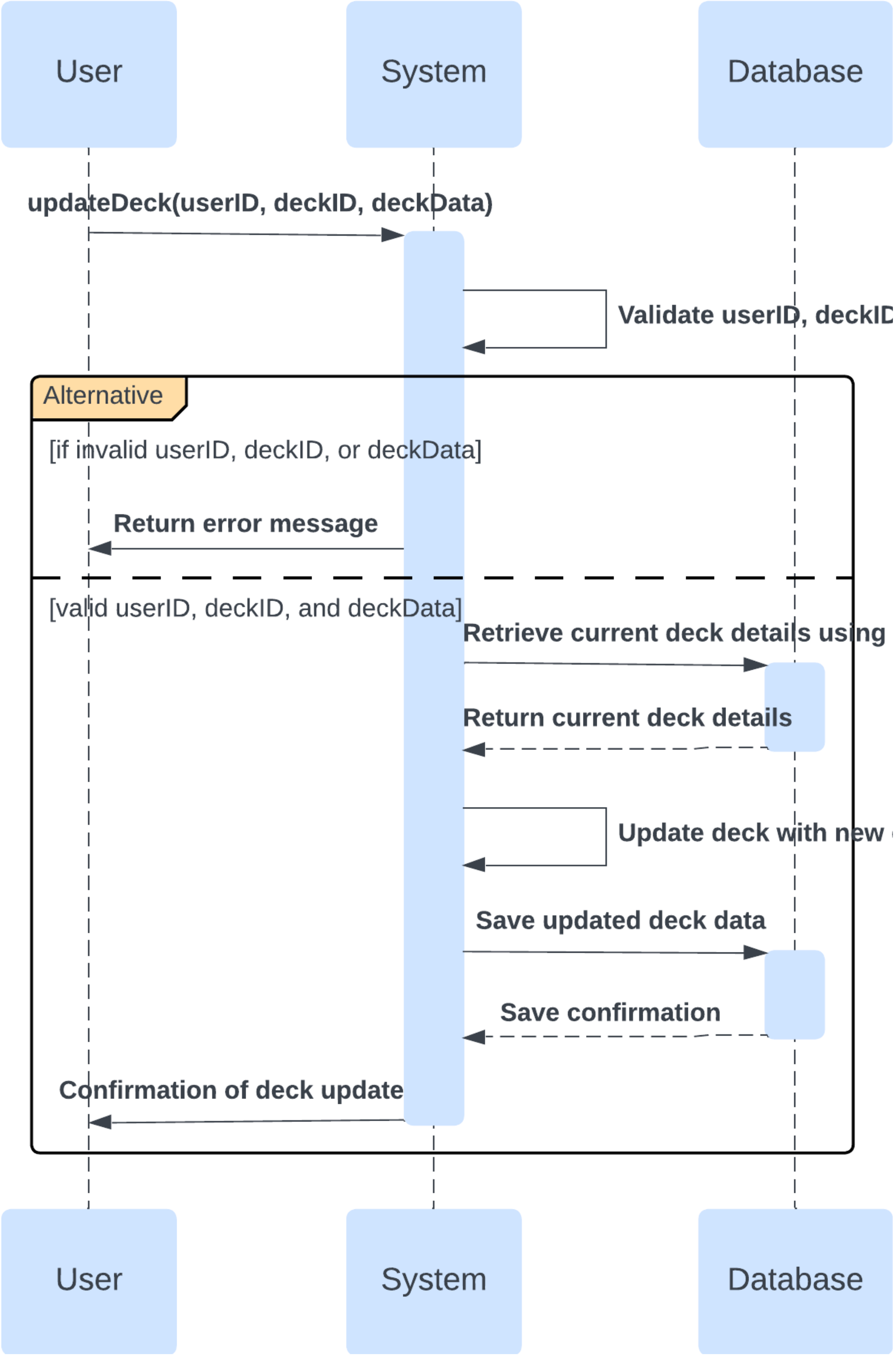
Interaction Diagrams

generateDeck(userID: String)



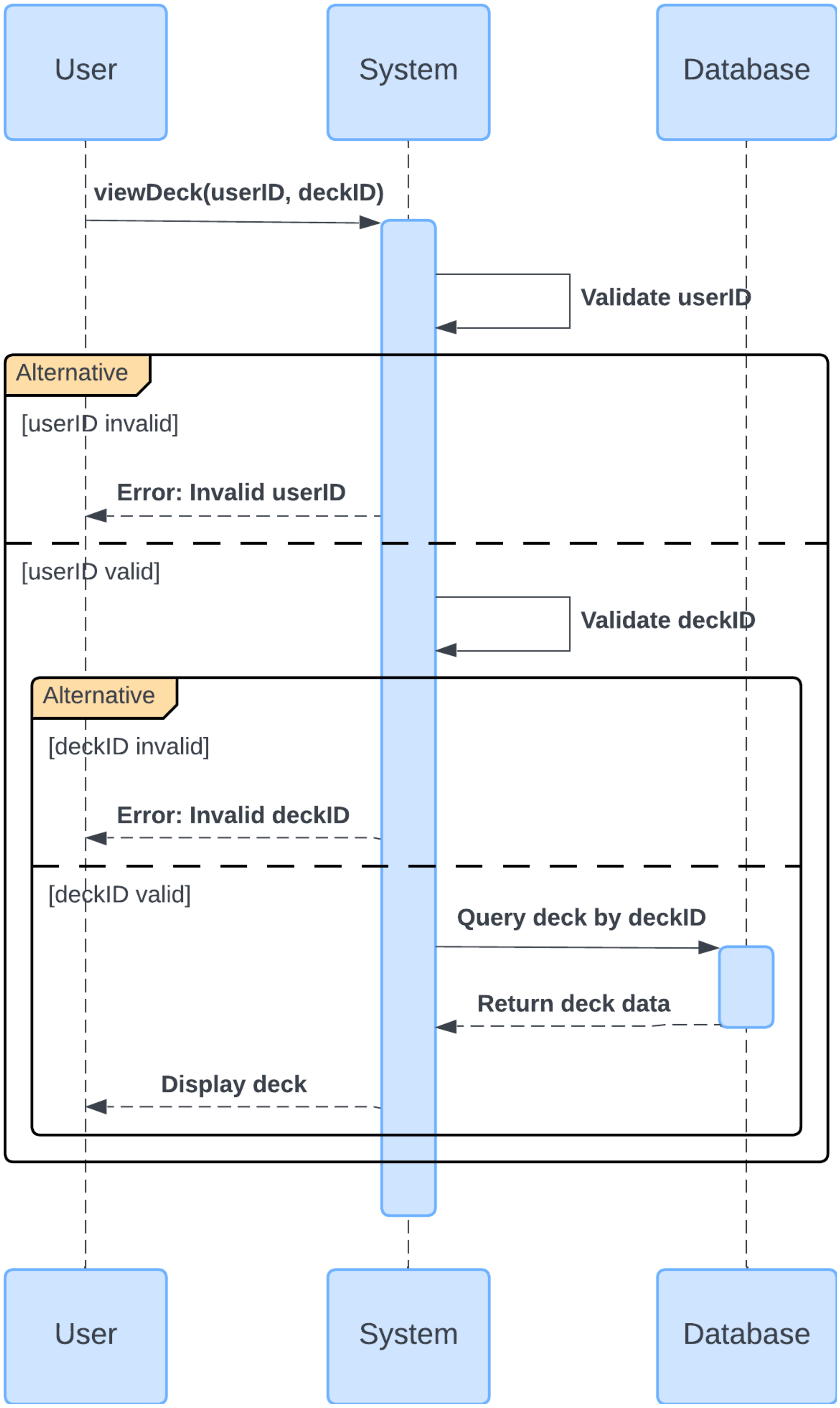
This illustrates the collaboration between the user, AI Engine, and database. The user initiates the process by providing their preferences, which the AI Engine uses to generate deck suggestions. The system then validates the user's ID and preferences before the generated deck is stored in the database.

updateDeck(userID: String, deckID: String, deckData: DeckData)


User
System
Database
updateDeck(userID, deckID, deckData)
Validate userID, deckID
Alternative
[if invalid userID, deckID, or deckData]
Return error message
[valid userID, deckID, and deckData]
Retrieve current deck details using
Return current deck details
Update deck with new
Save updated deck data
Save confirmation
Confirmation of deck update
User
System
Database

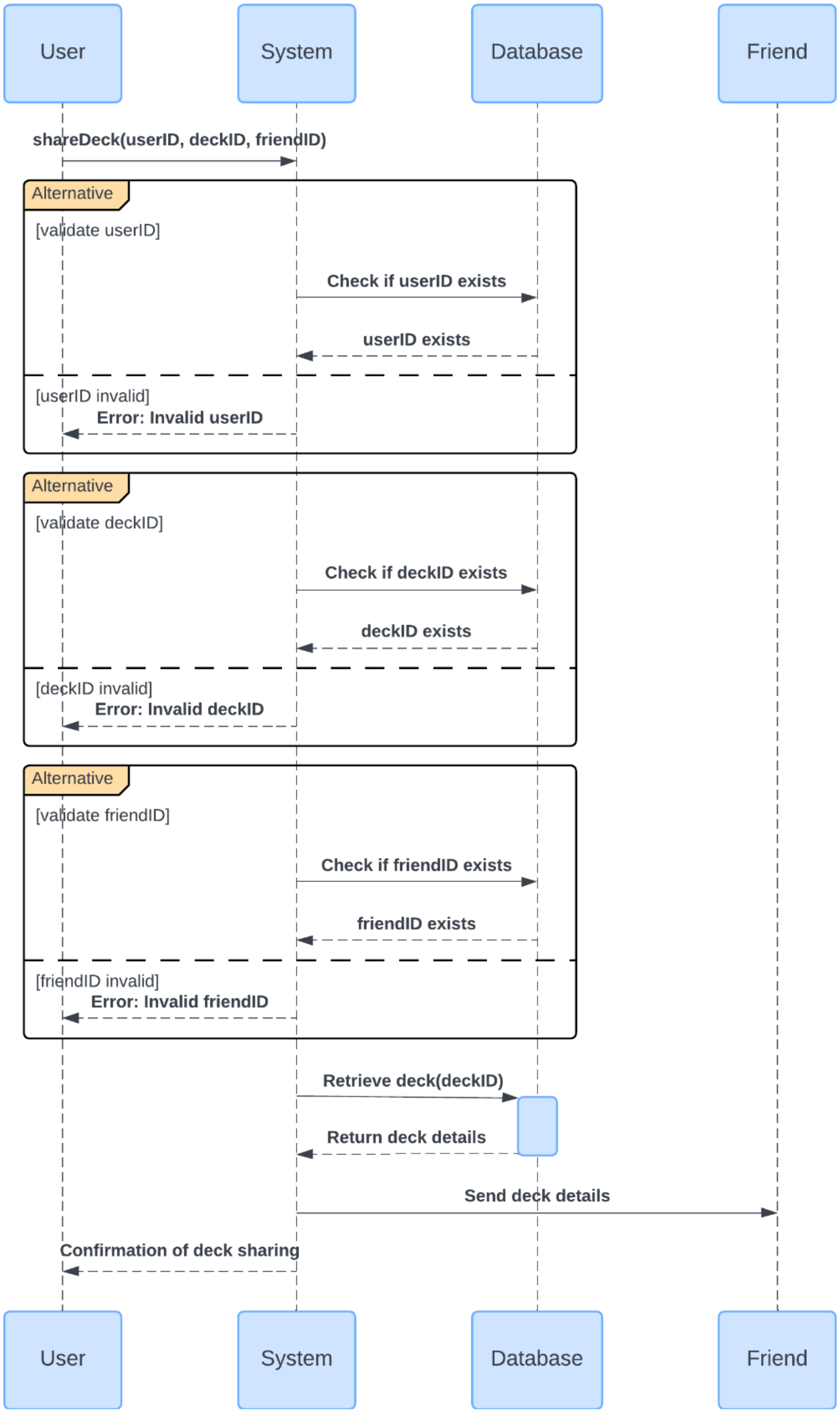
The user selects a deck to update and submits the changes. The system first validates the user ID and deck data, then retrieves the existing deck from the database. After incorporating the updates, the modified deck is saved back to the database.

viewDeck(userID: String, deckID: String)



The user requests to view a specific deck. The system validates the user ID and retrieves the requested deck from the database. The deck information is then displayed to the user, allowing them to review their deck details.

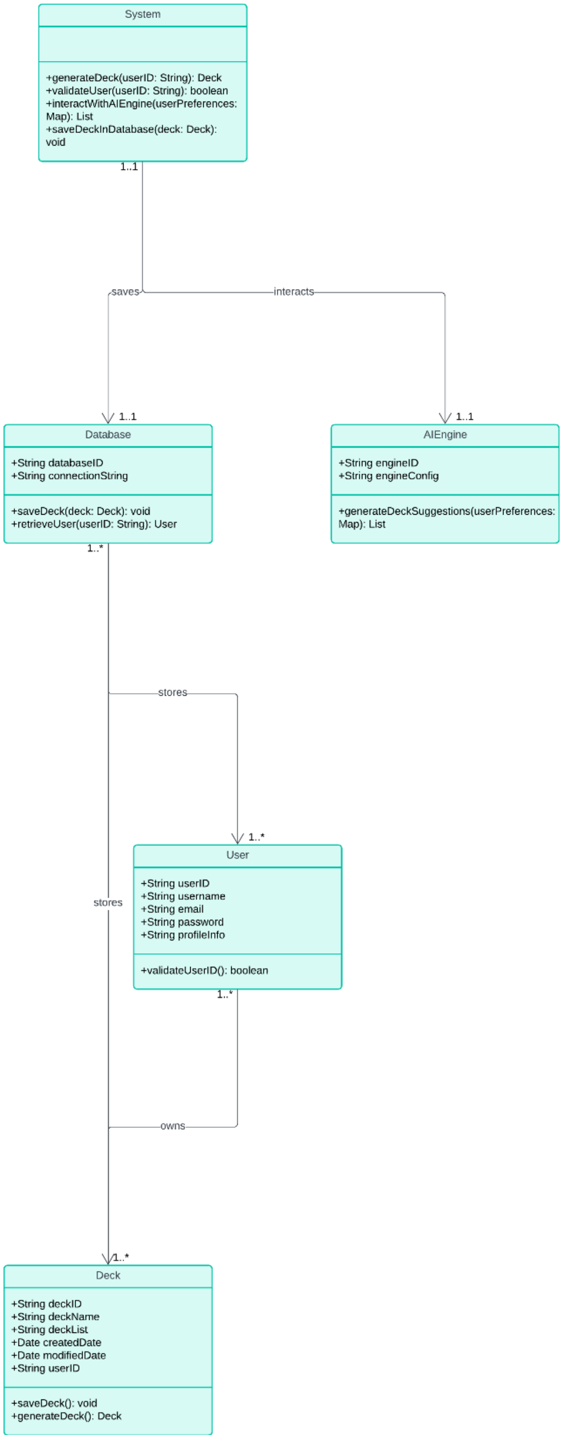
shareDeck(userID: String, deckID: String, friendID: String)



The user inputs the friend's details, and the system validates both the user ID and the friend's ID. The system then retrieves the deck from the database and sends it to the specified friend.

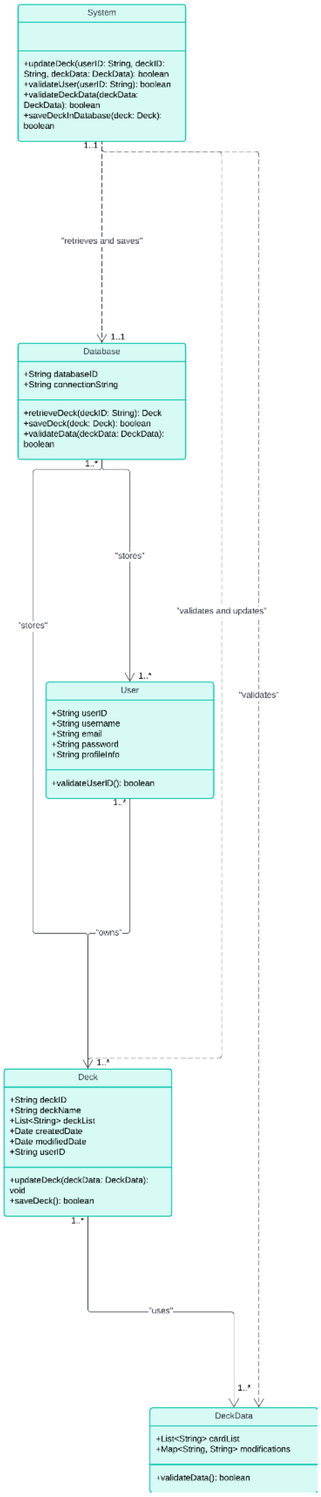
Design Class Diagrams

generateDeck



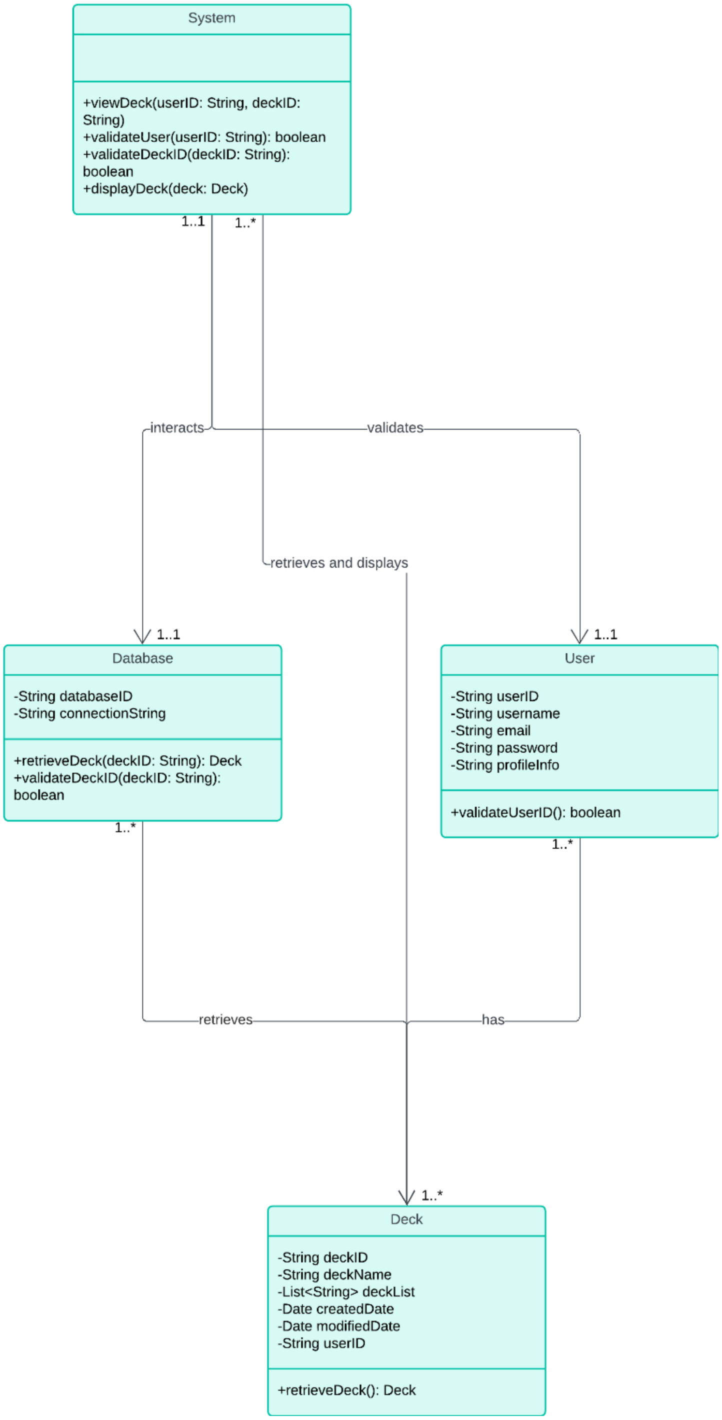
The User class provides preferences that the AIEngine uses to generate deck suggestions. The generated deck is then associated with the User and saved in the Database.

updateDeck



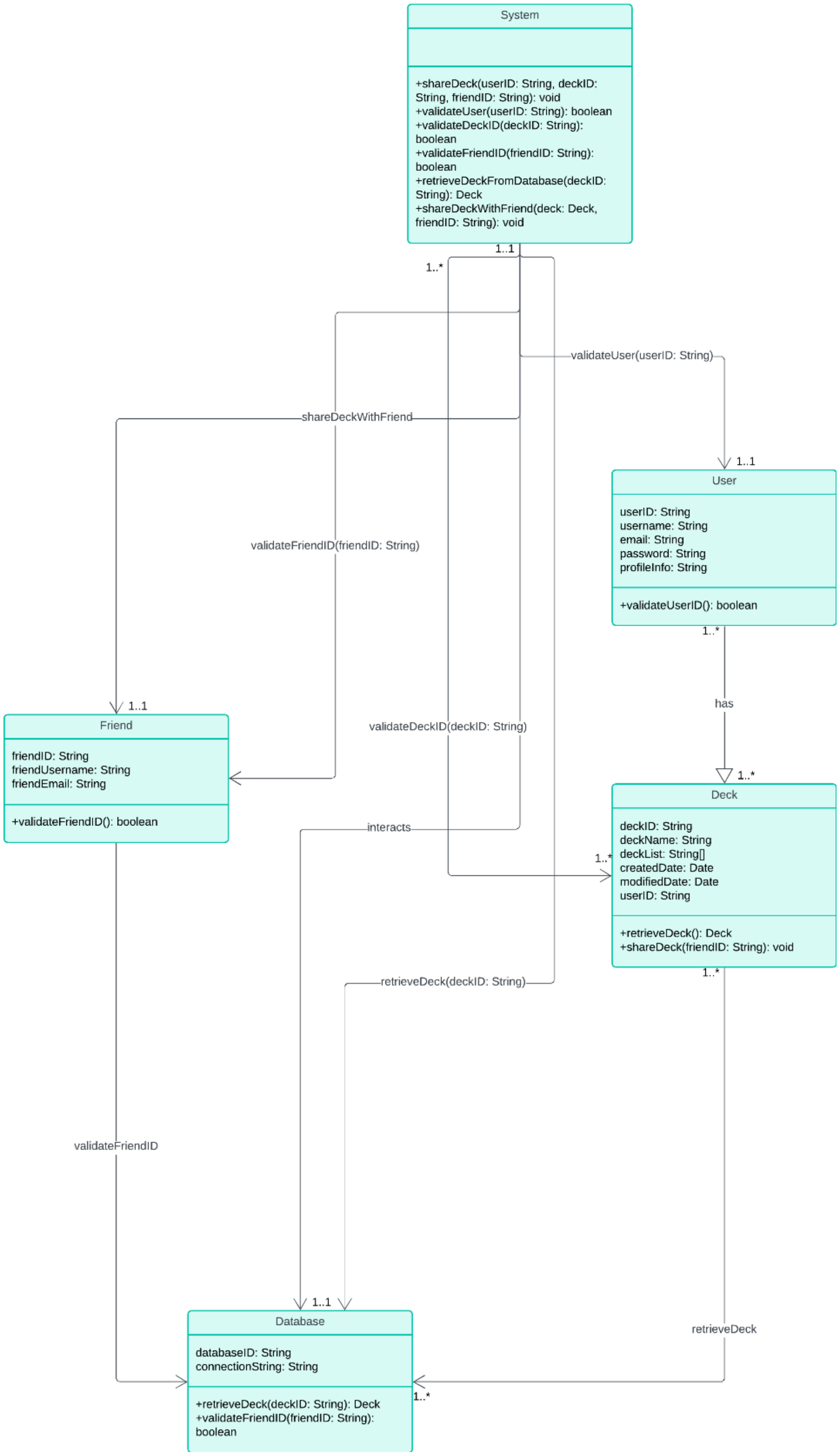
The User class initiates the update by providing new deck data. The Deck class processes these updates and interacts with the Database to save the changes.

viewDeck



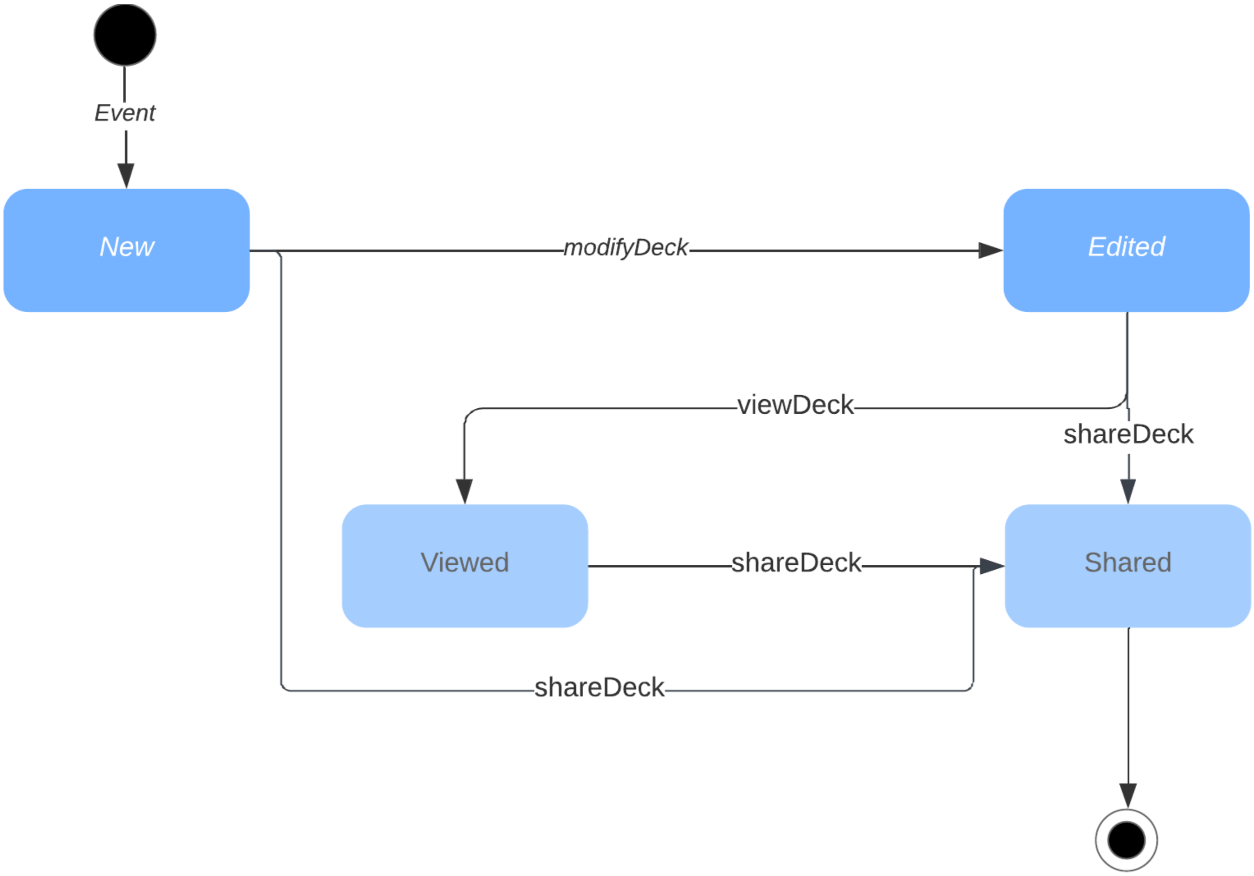
The User class requests to view a deck, which the Deck class retrieves from the Database. The deck details are then presented to the User.

shareDeck


System
+shareDeck(userID: String, deckID: String, friendID: String): void
+validateUser(userID: String): boolean
+validateDeckID(deckID: String): boolean
+validateFriendID(friendID: String): boolean
+retrieveDeckFromDatabase(deckID: String): Deck
+shareDeckWithFriend(deck: Deck, friendID: String): void
1..1
1..*
validateUser(userID: String)
shareDeckWithFriend
1..1
User
userID: String
username: String
email: String
password: String
profileInfo: String
+validateUserID(): boolean
1..*
has
validateFriendID(friendID: String)
1..1
Friend
friendID: String
friendUsername: String
friendEmail: String
+validateFriendID(): boolean
validateDeckID(deckID: String)
1..*
Deck
deckID: String
deckName: String
deckList: String[]
createdDate: Date
modifiedDate: Date
userID: String
+retrieveDeck(): Deck
+shareDeck(friendID: String): void
interacts
1..*
1..*
retrieveDeck(deckID: String)
validateFriendID
retrieveDeck
1..1
Database
databaseID: String
connectionString: String
+retrieveDeck(deckID: String): Deck
+validateFriendID(friendID: String): boolean
1..*

The User class selects a deck to share and provides the friend's details. The Deck class retrieves the deck from the Database and shares it with the Friend class.

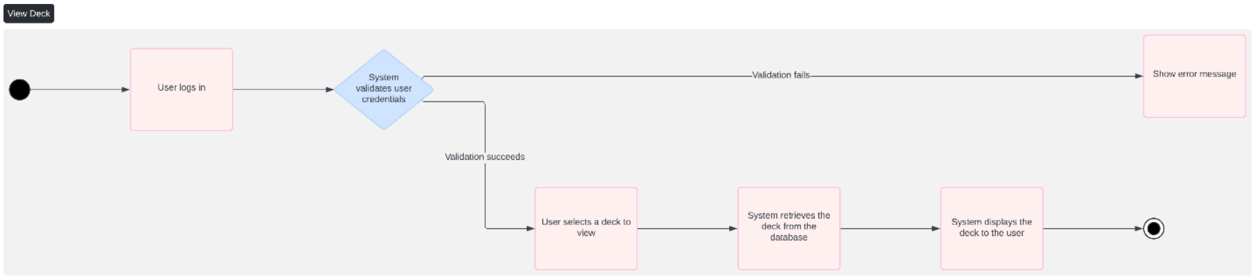
UML State Diagram



The UML State Diagram illustrates the different states a deck can be in, from creation to sharing. It starts with the Initial State, where the user initiates a deck-related action. The deck can transition into the Creating State where the AI Engine generates a new deck based on user preferences. Once created, it moves to the Editing State if the user decides to make changes, or to the Viewing State when the user wants to view their deck. If the user chooses to share the deck, it enters the Sharing State. Each state transition is triggered by specific user actions, ensuring the deck progresses through the system smoothly and logically. The Final State is reached when the deck is either successfully saved or shared, completing the interaction.

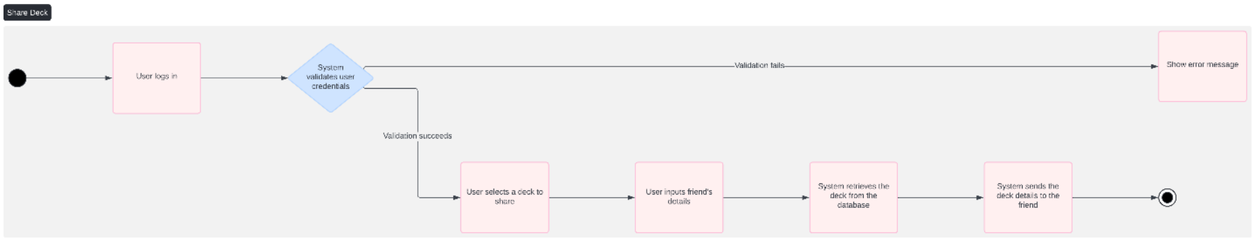
Activity Diagrams

View Deck


View Deck
User logs in
System validates user credentials
Validation fails
Show error message
Validation succeeds
User selects a deck to view
System retrieves the deck from the database
System displays the deck to the user

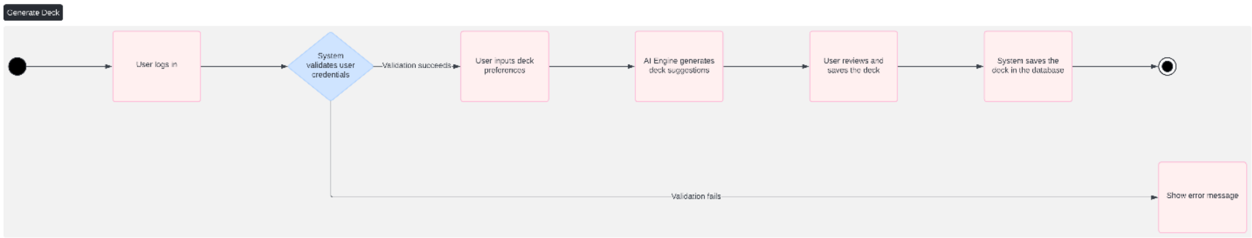
The Activity Diagram for viewing a deck starts with the user logging in and navigating to the deck view section. The system verifies the user's credentials and retrieves the deck data from the database. It then displays the deck information so the user can see all the details.

Share Deck


Share Deck
User logs in
System validates user credentials
Validation fails
Show error message
Validation succeeds
User selects a deck to share
User inputs friend's details
System retrieves the deck from the database
System sends the deck details to the friend

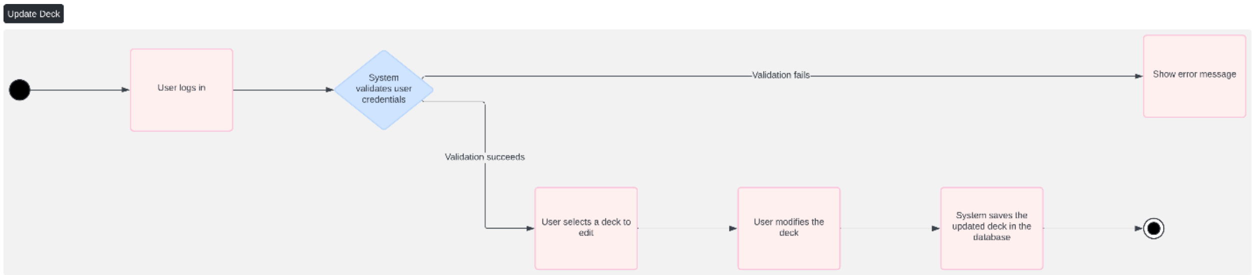
The Activity Diagram for sharing a deck begins with the user logging in and selecting a deck to share. The user enters a friend's details, and the system verifies both the user's and the friend's credentials. Once everything checks out, the system retrieves the deck from the database and sends it to the friend. The friend receives the deck, completing the sharing process.

Generate Deck


Generate Deck
User logs in
System validates user credentials
Validation succeeds
User inputs deck preferences
AI Engine generates deck suggestions
User reviews and saves the deck
System saves the deck in the database
Validation fails
Show error message

In the Activity Diagram for generating a deck, the user logs in and provides deck preferences. The AI Engine takes these preferences and generates deck suggestions. The system then validates the user's ID and preferences before saving the generated deck to the database. The user can review and save the deck, finishing the generation process.

Update Deck


Update Deck
User logs in
System validates user credentials
Validation fails
Show error message
Validation succeeds
User selects a deck to edit
User modifies the deck
System saves the updated deck in the database

The Activity Diagram for updating a deck starts with the user logging in and choosing an existing deck to update. The user makes the desired changes, and the system validates the user ID and the new deck data. The updated deck is then saved back to the database, ensuring the changes are accurately recorded. The user receives confirmation that the update was successful.

Skeleton Classes

generateDeck

```
class User {
    String userID;
    String username;
    String email;
    String password;
    ProfileInfo profileInfo;

    void validateUserID() {
        // Validation logic here
    }
}

class Deck {
    String deckID;
    String deckName;
    List<Card> deckList;
    Date createdDate;
    Date modifiedDate;
    String userID;

    void saveDeck() {
        // Save deck logic here
    }
}

class AIEngine {
    String engineID;
    String engineConfig;

    List<Card> generateDeckSuggestions(UserPreferences preferences) {
        // Deck suggestion logic here
    }
}

class Database {
    String databaseID;
    String connectionString;

    void saveDeck(Deck deck) {
        // Database save logic here
    }
    Deck retrieveDeck(String deckID) {
        // Database retrieval logic here
    }
}
```

UpdateDeck

```
class User {
    String userID;
    String username;
    String email;
    String password;
    ProfileInfo profileInfo;

    void validateUserID() {
        // Validation logic here
    }
}

class Deck {
    String deckID;
    String deckName;
    List<Card> deckList;
    Date createdDate;
    Date modifiedDate;
    String userID;
    void updateDeck(DeckData deckData) {
        // Update deck logic here
    }
}

class Database {
    String databaseID;
    String connectionString;

    Deck retrieveDeck(String deckID) {
        // Database retrieval logic here
    }
    void saveDeck(Deck deck) {
        // Database save logic here
    }
}
```

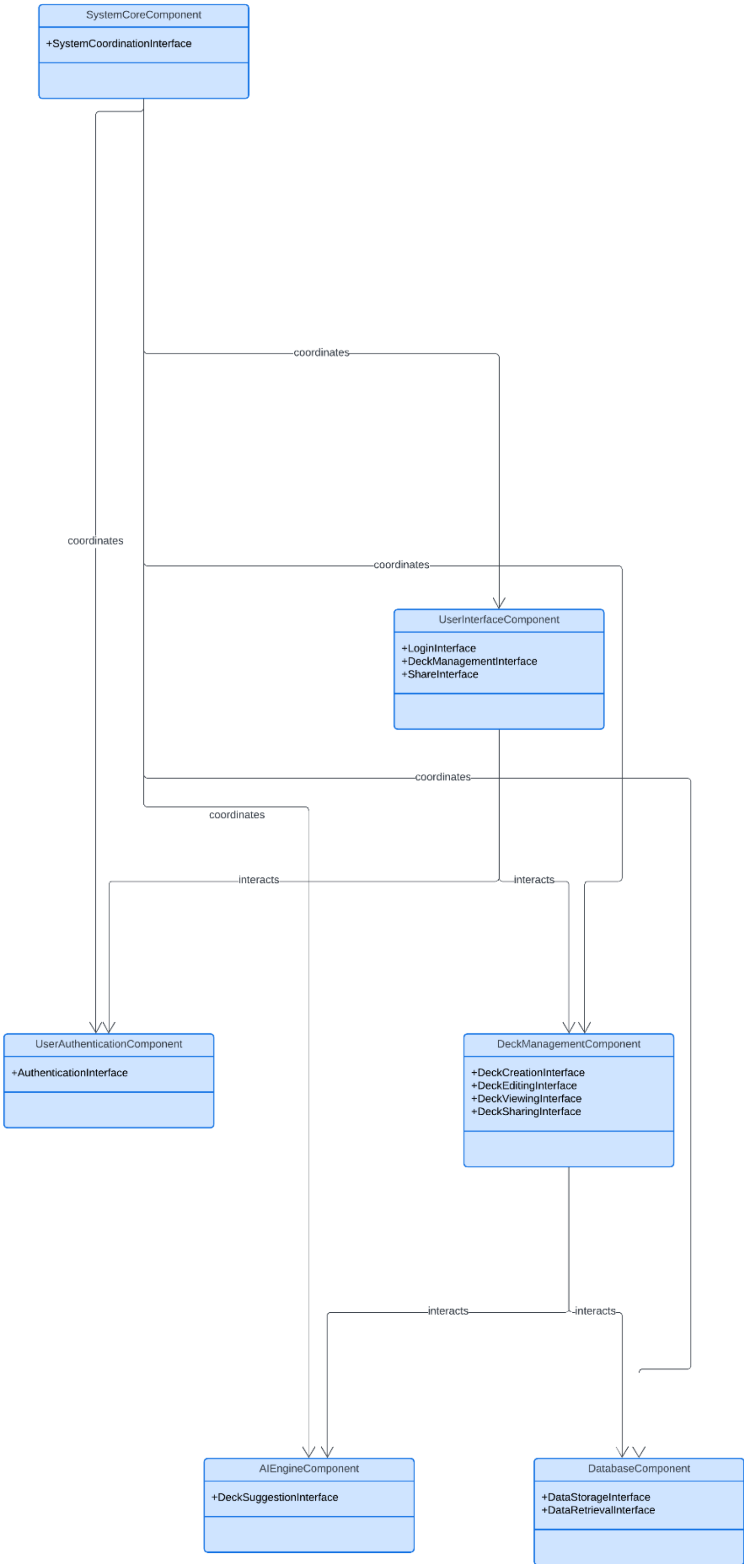
viewDeck

```
class User {
    String userID;
    String username;
    String email;
    String password;
    ProfileInfo profileInfo;

    void validateUserID() {
        // Validation logic here
    }
}

class Deck {
    String deckID;
    String deckName;
    List<Card> deckList;
    Date createdDate;
    Date modifiedDate;
    String userID;

    void retrieveDeck() {
        // Retrieve deck logic here
    }
}

class Database {
    String databaseID;
    String connectionString;
    Deck retrieveDeck(String deckID) {
        // Database retrieval logic here
    }
}
```

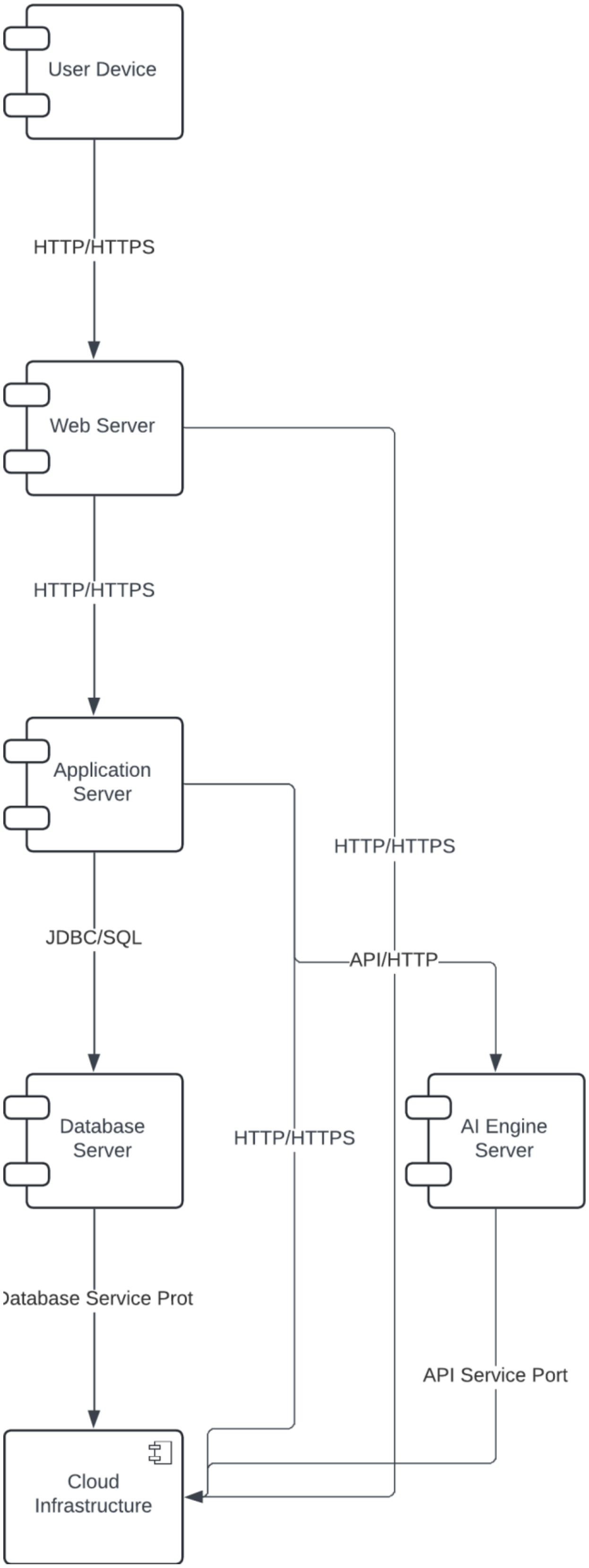
shareDeck

```
class User {
    String userID;
    String username;
    String email;
    String password;
    ProfileInfo profileInfo;

    void validateUserID() {
        // Validation logic here
    }
}

class Deck {
    String deckID;
    String deckName;
    List<Card> deckList;
    Date createdDate;
    Date modifiedDate;
    String userID;

    void shareDeck(String friendID) {
        // Share deck logic here
    }
}

class Friend {
    String friendID;
    String friendUsername;
    String friendEmail;

    void validateFriendID() {
        // Validation logic here
    }
}

class Database {
    String databaseID;
    String connectionString;
    Deck retrieveDeck(String deckID) {
        // Database retrieval logic here
    }
    void validateFriendID(String friendID) {
        // Validation logic here
    }
}
```

Component Diagram



Deployment Diagram



Tech Stack

- Frontend: React.js for the web interface and Flutter for mobile applications.
- Database: PostgreSQL hosted on AWS RDS for flexible, scalable data storage.
- Authentication: OAuth 2.0 for secure authentication and authorization.
- Deployment: Docker for containerization, Kubernetes for orchestration, and AWS for cloud services.
- Monitoring and Logging: Prometheus for monitoring, ELK stack for logging.

Conclusion

This document gives a thorough rundown of a Generative AI-Powered MTG Deck Building Framework. We've detailed the system's design and functionality, showcasing everything from use case models and domain models to interaction diagrams and design class diagrams. This framework is all about helping users create, manage, and share Magic: The Gathering decks easily. By using AI to suggest deck ideas and integrating strong user management, database interaction, and deck-sharing capabilities, I've made sure the system is both seamless and user-friendly. The detailed architecture and component descriptions ensure that all parts of the system work together smoothly, offering a reliable and scalable solution for MTG fans.