Project Report #3

Quiz of Fury

Daniel Seip
Daniel Terreros
Julia Rogers

Github URL: https://github.com/DanielTKC/quiz_of_fury/

CSCI 441

Table of Contents

All Team Members Contributed Equally To This Report

1. Customer Statement of Requirements

a. Problem Statement:

i. The Reality of Student Life: Time Constraints and Academic Pressure

I am a full time college student with a full time course load. I also have a job that takes up a lot of my time outside of class, breaking up time that could traditionally be spent on only studying. I can typically only study at night or in 20 minute intervals between getting to various classes or breaks at work. There is an almost overwhelming amount of material to learn from all of my classes, sometimes with very little time to work on mastering it! I care deeply about learning this material - not just for the sake of a grade but so that I actually understand what I learn and can apply it after the class ends. I have tried different study methods: I wrote out notes, made flashcards by hand, re-read texts, even used current flashcard type webapps! As I progressed through the material, I constantly felt like studying was more difficult than it needed to be. I don't want shortcuts, I want to use a tool that better engages my memory without distractions.

ii. Information Overload and Inefficient Review Processes

Courses at my school give out an overwhelming amount of material for us to learn, often in a very short period of time. So much information that sometimes it's hard for me to know what to focus on. Reviewing for tests can be tedious and unstructured; I end up going over all the material including what I already know well instead of focusing on my weak spots. There are several tools that I've tried to use to help me learn. Quizlet is decent for flashcards but not good at tracking my long term progress and spacing repetition. Anki is confusing to set up and almost overloaded with features that distract from the core activity of studying. Putting notes into Google Docs is freeform, I have forgotten where a lot of information I need to study goes. I can review the textbook for the class and come up with quizzes for myself, but I'm not really working on the weak points when the tools continually review very basic information. When I review, I want a system that helps me find and focus on my weak areas quickly without repetition on what I've already mastered.

iii. The Setup Time Problem: Friction Between Me and Learning

Trying to find what I need to study or spending too much time setting up and learning a system breaks the flow of my studying and takes up too much of the limited amount of time that I have to study in. I want something that can direct me to my problem areas quickly with a minimal amount of clicks between me and starting to study. Without this, it feels like I'm just reviewing topics that I already know. I waste time organizing notes, creating mini-quizzes for myself, and losing motivation due to poor feedback loops. I want to feel productive, and I don't want to feel like I spend more time organizing the information than actually doing the studying that I need to.

iv. Gamification Gone Wrong: When Features Become Obstacles

Most web apps try to gamify studying to hold engagement, but these distractions don't actually help me learn. Light gamification, such as streak tracking for consecutive days, is an easy way to encourage me to keep logging in, but heavier systems with leaderboards and achievements tends to add clicks and screens between me and what I want to accomplish. I want a system that I can get into, find a set of cards to study, and do the core activity: study. I want the system to allow me to do all this quickly and fit into a day where my class schedule and work schedule mean that I may not have a lot of time to sit at my device. A system with a clean interface and useful, clearly signposted features will ensure that I don't spend too much time trying to get setup. It needs to work on any device that I have on me at the time, and not require that I sit at a computer to do the studying.

v. The Need for Intuitive Organization and Quick Setup

The system needs to be quick to set up. There should be an obvious way to create and edit cards, with controls to move them between decks. Additionally, setting up may take me too long. If I have a friend in the same class as me, I should be able to receive a deck shared by them, and bypass setting up a deck from scratch entirely. Although organization options would be nice, I should also be able to bypass those and start studying right away. I want a minimal amount of navigation between opening the web app and getting into a deck of flashcards to study. I should be able to go back and change tags and organization of decks as my needs change and I learn more about where I will need this information. There should be an option to move cards between decks and hide cards that I don't want to study. Organization in a study app should be intuitive with a minimum amount of clicks to get from logging into an account and getting to studying.

vi. Finding the Right Content: Search and Discovery Challenges

Conversely, finding flashcards on other apps can be a time consuming process in itself. There needs to be an easy way to find the deck that I currently want to study. These should be searchable by course, topic, or date created. This could possibly be implemented by some number of tags being allowed on each deck. Being able to customize and search on the name of the deck is important as well. It should be easy to review material, and the app should clearly report on scores so I can study the weak points of knowledge and start to work towards long term retention. It needs to be easy to customize. I want to be able to hide decks and cards I don't want to see anymore, and I want to be able to move and duplicate cards between decks if needed.

vii. The Solution: A Tool That Actually Enhances Learning

If there was a better tool for studying, I could spend less time thinking about how to study, and more time actually learning the course material that I'm struggling with. It would reduce my stress before tests and exams by being more effective than the guesswork method or online tools that I currently use, increasing my retention and confidence. Long term, this would be a great method for me to review any subject that I've taken in school, or any certifications I want to study for after I have completed school. I don't need complicated settings or opaque connections between decks that take learning and time to set up. I need a study system that fits into my busy schedule and is intuitive enough to pick up immediately.

2. Glossary of Terms

Term	**Definition**
Deck	A collection of flashcards organized around a specific topic or subject
Flashcard	A digital card with a question on one side and answer on the other
Study Session	A period of active learning using flashcards with performance tracking
Spaced Repetition	Learning technique where review intervals increase based on how well material is remembered
Confidence Rating	User's self-assessment (1-5 scale) of how well they remembered a flashcard answer
Active Recall	Learning method that requires retrieving information from memory rather than passive review
Study Streak	Number of consecutive days a user has completed study sessions

Term	**Definition**
Mastery Level	Indicator of how well a user knows the material in a deck based on performance
Due Cards	Flashcards scheduled for review based on spaced repetition algorithm
Next Review	Calculated date/time when a flashcard should be shown again

3. Goals, Requirements, and Analysis

a. Business Goals

Primary Goal: Create Simple, Effective Learning Tool

- Improve Student Learning Outcomes
    - Increase Information Retention (Target: 2x improvement over traditional methods)
    - Reduce Time Required for Effective Study
    - Make Studying More Engaging and Sustainable
- Maximize User Engagement
    - Achieve High Daily Usage Rate\
    - Maintain Simple, Intuitive User Experience
    - Provide Motivational Features Without Complexity
- Demonstrate Technical Competency
    - Build Working MVP in 5 Weeks
    - Showcase Software Engineering Best Practices

- Create Scalable Foundation for Future Development

b. Enumerated Functional Requirements

ID	**Priority**	**Description**
REQ-1	High	User account creation and authentication
REQ-2	High	Create and edit flashcard decks
REQ-3	High	Add, edit, and delete individual flashcards
REQ-4	High	Study flashcards with spaced repetition algorithm
REQ-5	High	Rate confidence level for each flashcard (1-5 scale) - (if time permits)
REQ-6	Medium	Track study sessions and calculate progress (if time permits)
REQ-7	Medium	Display user dashboard with deck overview (if time permits)
REQ-8	Medium	Calculate and display study streaks (if time permits)

ID	Priority	Description
REQ-9	Medium	Basic point system for completed study sessions (if time permits)
REQ-10	Low	Simple leaderboard (if time permits)

c. Enumerated Nonfunctional Requirements

ID	Priority	Description
REQ-11	High	**Usability:** New users can create first deck and study within 5 minutes
REQ-12	Medium	**Performance:** Page load times under .3 seconds on mobile
REQ-13	High	**Reliability:** System works consistently without crashes during demos
REQ-14	High	**Functionality:** Support up to 100 concurrent users (adequate for class demo)
REQ-15	Medium	**Supportability:** Clear error messages and user feedback
REQ-16	Medium	**Portability:** Responsive design works on desktop and mobile browsers
REQ-17	Low	**Scalability:** Code structure allows for future feature additions

d. User Interface Requirements

ID	Priority	Description	Visual Reference
UI-1	High	Clean, distraction-free study interface	Large flashcard display with minimal UI elements
UI-2	High	Simple deck creation with intuitive form	Step-by-step process with clear field labels
UI-3	High	Mobile-responsive design for all screens	Touch-friendly buttons, readable text on small screens
UI-4	Medium	Dashboard showing deck status and statistics (if time permits)	Card-based layout with progress indicators
UI-5	Medium	Visual feedback for confidence rating (if time permits)	5-star rating system with immediate visual response

4. Use Cases

i. Casual Description

Note: Certain use cases (e.g., progress tracking, confidence rating, leaderboard, and deck sharing) have already been marked in Section 3b - 3d as future work (if time permits). These features are referenced here for completeness but are not guaranteed in the current iteration.

UC-1: Create Flashcard Deck (REQ-2, REQ-3)
User creates a new collection of flashcards on a specific topic by entering a deck name and adding question/answer pairs.

UC-2: Study Flashcards (REQ-4, REQ-5, REQ-6)
User reviews flashcards due for study, rates confidence level for each card, and system schedules future reviews based on spaced repetition algorithm.

UC-3: View Progress Dashboard (REQ-7, REQ-8, REQ-9)
User views their study statistics, including deck progress, study streaks, and points earned.

UC-4: Edit Flashcard Content (REQ-3)
User modifies existing flashcards by changing questions, answers, or deleting cards from their decks.

UC-5: Authenticate User (REQ-1)
User creates account and logs in.

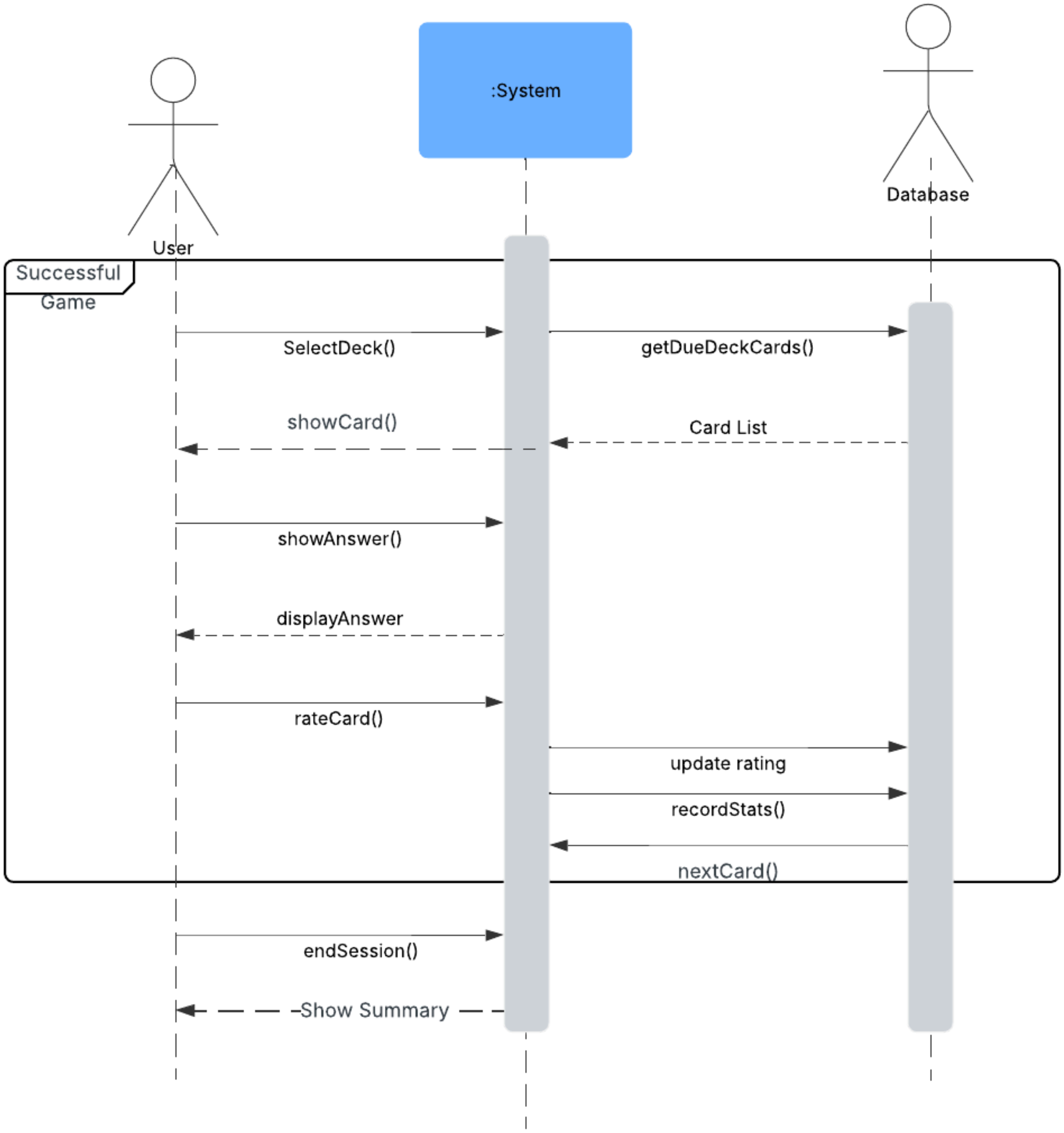
UC-6: Rate Confidence (REQ-3)
User rates their confidence with studying individual flash cards to reduce visibility while viewing deck.

UC-7: Manage Profile (REQ-6, REQ-7)
User can see profile statistics and make changes to email and displayed name.

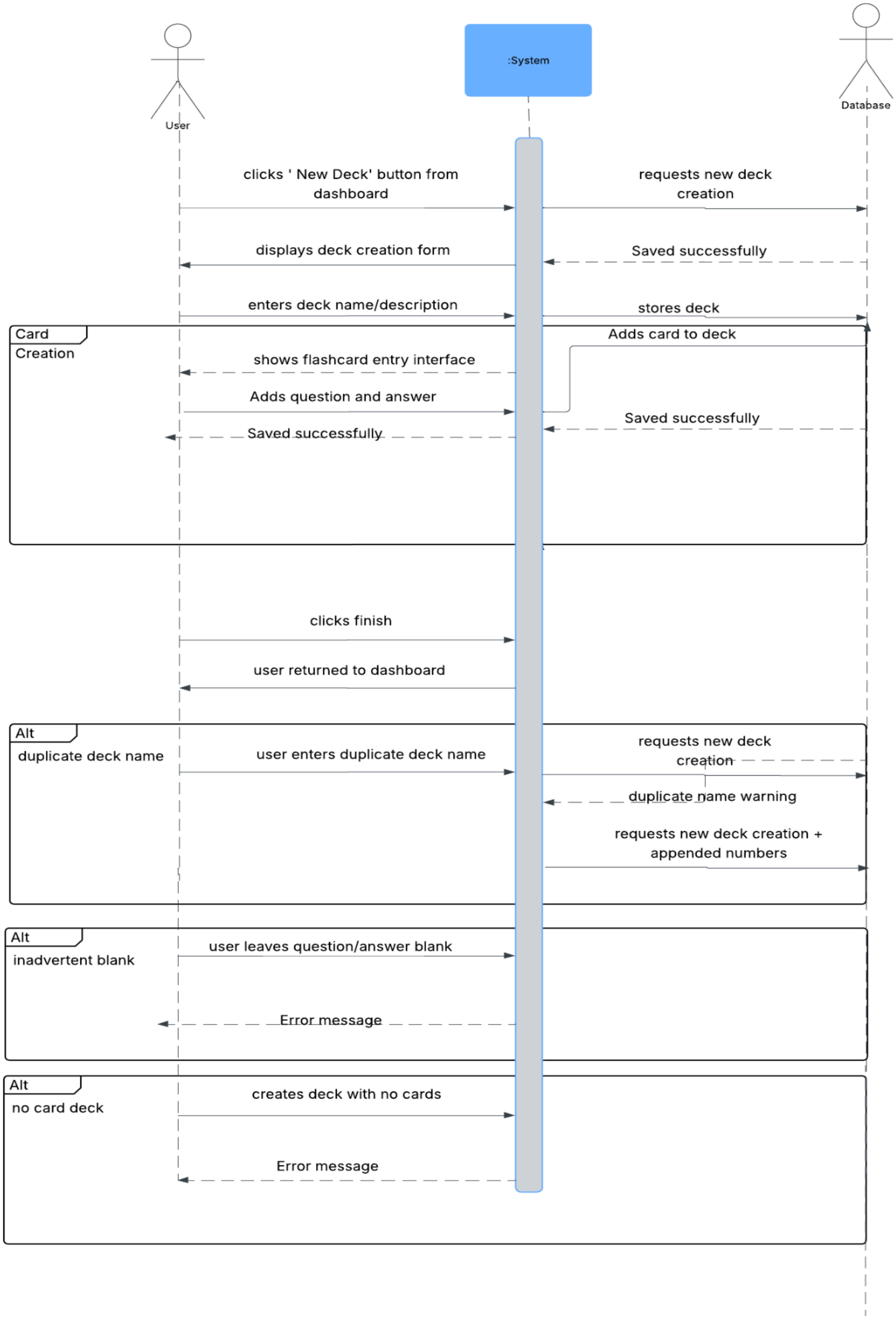
UC-8: Share Deck (REQ-10)
User shares a deck with another user via email link.

5. System Sequence Diagrams

UC-2: Study Flashcards



UC-1: Create Flashcard Deck


:System
User
Database
clicks ' New Deck' button from dashboard
requests new deck creation
displays deck creation form
Saved successfully
enters deck name/description
stores deck
Card Creation
Adds card to deck
shows flashcard entry interface
Adds question and answer
Saved successfully
Saved successfully
clicks finish
user returned to dashboard
Alt
duplicate deck name
user enters duplicate deck name
requests new deck creation
duplicate name warning
requests new deck creation + appended numbers
Alt
inadvertent blank
user leaves question/answer blank
Error message
Alt
no card deck
creates deck with no cards
Error message

iii. Traceability Matrix

Req't	**PW**	**UC-1 Create Deck**	**UC-2 Study Cards**	**UC-3 View Dashboard**	**UC-4 Edit Cards**	**UC-5 User Auth**	**UC-6 Rate Confidence**	**UC-7 Manage Profile**	**UC-8 Share Deck**
REQ-1	5	X				X			
REQ-2	5	X							X
REQ-3	5	X			X				
REQ-4	5		X						
REQ-5	4		X				X		
REQ-6	4		X				X	X	
REQ-7	3			X				X	
REQ-8	2			X					
REQ-9	2			X					
REQ-10	1								X
UI-1	4		X						
UI-2	4	X			X				
UI-3	3	X	X	X	X	X			
UI-4	3			X					
UI-5	2		X				X		
Max PW	**5**	**5**	**5**	**3**	**5**	**5**	**4**	**0**	**0**
Total PW	**55**	**17**	**20**	**11**	**10**	**5**	**6**	**0**	**0**

6. Effort Estimation Using Use Case Points

Actors Classification

Actor	Type	Complexity	Weight	Count	Total
Student/User	Simple	Simple (API-based authentication)	1	1	1
System	Simple	Simple (internal system actor)	1	1	1
Total Actor Points					**2**

Use Cases Classification

Use Case	Transactions	Complexity	Weight	Count	Total
UC-1: Create Flashcard Deck	3-4	Simple	5	1	5
UC-2: Study Flashcards	4-5	Simple	5	1	5
UC-4: Edit Flashcard Content	2-3	Simple	5	1	5
UC-5: Authenticate User	2-3	Simple	5	1	5
UC-6: Rate Confidence	1-2	Simple	5	1	5

Use Case	Transactions	Complexity	Weight	Count	Total
UC-7: Manage Profile	2-3	Simple	5	1	5
Total Use Case Points					**30**

Transaction Count

UC-1: Create Deck (3-4 transactions)

1. User inputs deck name and description
2. System validates and creates deck
3. User adds flashcards
4. System saves flashcards to deck

UC-2: Study Flashcards (4-5 transactions)

1. User selects deck to study
2. System presents flashcard question
3. User views answer and rates confidence
4. System updates card data
5. System records session progress

UC-4: Edit Flashcard Content (2-3 transactions)

1. User selects card to edit
2. User modifies question/answer
3. System saves changes

Technical Complexity Factors (TCF)

Factor	Description	Rating (0-5)	Weight	Score
T3	End-user efficiency	4	1.0	4
T5	Reusable code	2	1.0	2

Factor	Description	Rating (0-5)	Weight	Score
T6	Easy to install	4	0.5	2
T7	Easy to use	4	0.5	2
T11	Security features	2	1.0	2
Total TCF Score				**12**

Technical Complexity Factor Calculation:

TCF = 0.6 + (0.01 × Total TCF Score)

TCF = 0.6 + (0.01 × 12)

TCF = 0.6 + 0.12

TCF = 0.72

Environmental Complexity Factors (ECF)

For student projects, ECF is set to 1.0 (as specified in instructions)

Use Case Points Calculation Process

Step 1: Calculate Unadjusted Use Case Points (UUCP)

UUCP = Total Actor Points + Total Use Case Points

UUCP = 2 + 30

UUCP = 32

Step 2: Apply Complexity Factors

UCP = UUCP × TCF × ECF

UCP = 32 × 0.72 × 1.0

UCP = 32 × 0.72

UCP = 23.04 ≈ 23 Use Case Points

Duration Calculation

- **Productivity Factor (PF):** 28 hours per use case point
- **Adjusted Use Case Points (UCP):** 23

Duration Calculation Process:

Total Project Effort = UCP × PF

Total Project Effort = 23 × 28

Total Project Effort = 644 hours

Team Distribution:

Team Size = 3 developers

Hours per Developer = Total Project Effort ÷ Team Size

Hours per Developer = 644 ÷ 3

Hours per Developer = 214.67 hours per person

Weekly Breakdown:

Project Duration = 8 weeks

Hours per Week per Developer = Hours per Developer ÷ Project Duration

Hours per Week per Developer = 214.67 ÷ 8

Hours per Week per Developer = 26.83 hours per week

Final Effort Estimation Summary

Metric	Value
Unadjusted Use Case Points (UUCP)	32
Technical Complexity Factor (TCF)	0.72
Environmental Complexity Factor (ECF)	1.0

Metric	**Value**
Adjusted Use Case Points (UCP)	23
Productivity Factor (PF)	28 hours/UCP
Total Project Effort	**644 hours**
Effort per Developer	**215 hours**
Hours per Week per Developer	**27 hours/week**

7. Domain Modeling

A. Conceptual Model

I. Concept definitions

Responsibility Description	**Type**	**Concept Name**
Manage all operations of retrieving, storing, and sharing deck objects. Includes calls to spaced repetition algorithm	D	Controller
Storage system for decks (contains definitions of the decks and the associated cards). This includes the data models for the card and deck type.	K	Database
Storage system for user accounts and access from the authentication system (although part of main database, this is listed separately as it is implemented by Django)	K	Profile database
Accesses the user storage system to sign in users and retrieve decks that belong to them. (implemented by Django)	D	Authenticator

Builds HTML templates to display the interface in the user’s browser (implemented by Django)	D	Template engine
System forms that allow the user to create and edit forms in Deck Storage	D	Editor
Displays statistics tied to the user account and the card model	D	Dashboard page
Gives the user a form to do a flash card study session	D	Quiz form

II. Association definitions

Concept pair	Association description	Association name
Controller <-> Template engine	Handle requests from the browser, provide HTML. Provide card order and deck information to the template engine. Gather feedback from user - allow access to edit page and flash card practice	Conveys requests
Template engine <-> Dashboard page	Provide HTML to the browser to render the dashboard page	Show dashboard page
Controller <-> deck storage	Gather data from deck storage, optionally calculate study order if user is opted into spaced repetition	Gather flashcards
Editor <-> deck storage	Create and edit flash card fields.	Create, edit flashcards
Editor <-> Template engine	Generates HTML to show user the edit/create card interface	Show create/edit page
Quiz form <-> Template engine	Generates HTML to show user the quiz interface	Show quiz page
Controller <-> Authenticator	The user sign in/out request is sent to the authenticator for authentication. Sign out doesn’t need further verification, sign in does	Log on/off
Authenticator <-> Profile database	The authenticator searched the profile database to confirm identity of user	Authenticate

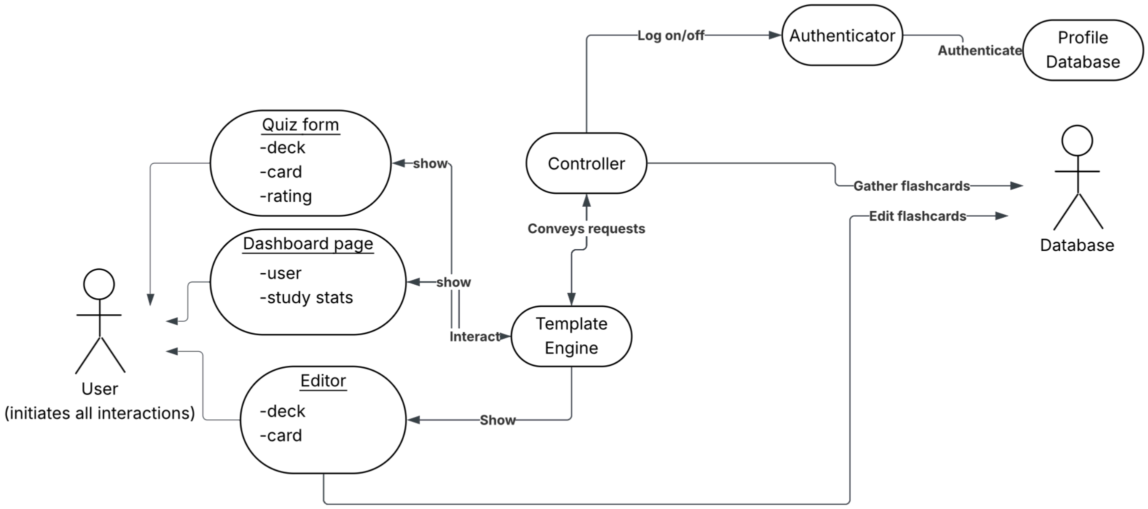
III. Attribute definitions

Concept	**Attributes**	**Attribute Description**
Quiz form	deck	The deck that is currently being quizzed on, ordered by the controller before being passed to the template engine
	card	The current card being tested
	rating	A form element allowing the user to rate their card performance. Scale of 1-5
Dashboard page	user	The current user profile
	Study statistics	Statistics aggregate or from the most recent study session
Editor	deck	Currently selected deck
	card	Card currently being modified

IV. Traceability matrix

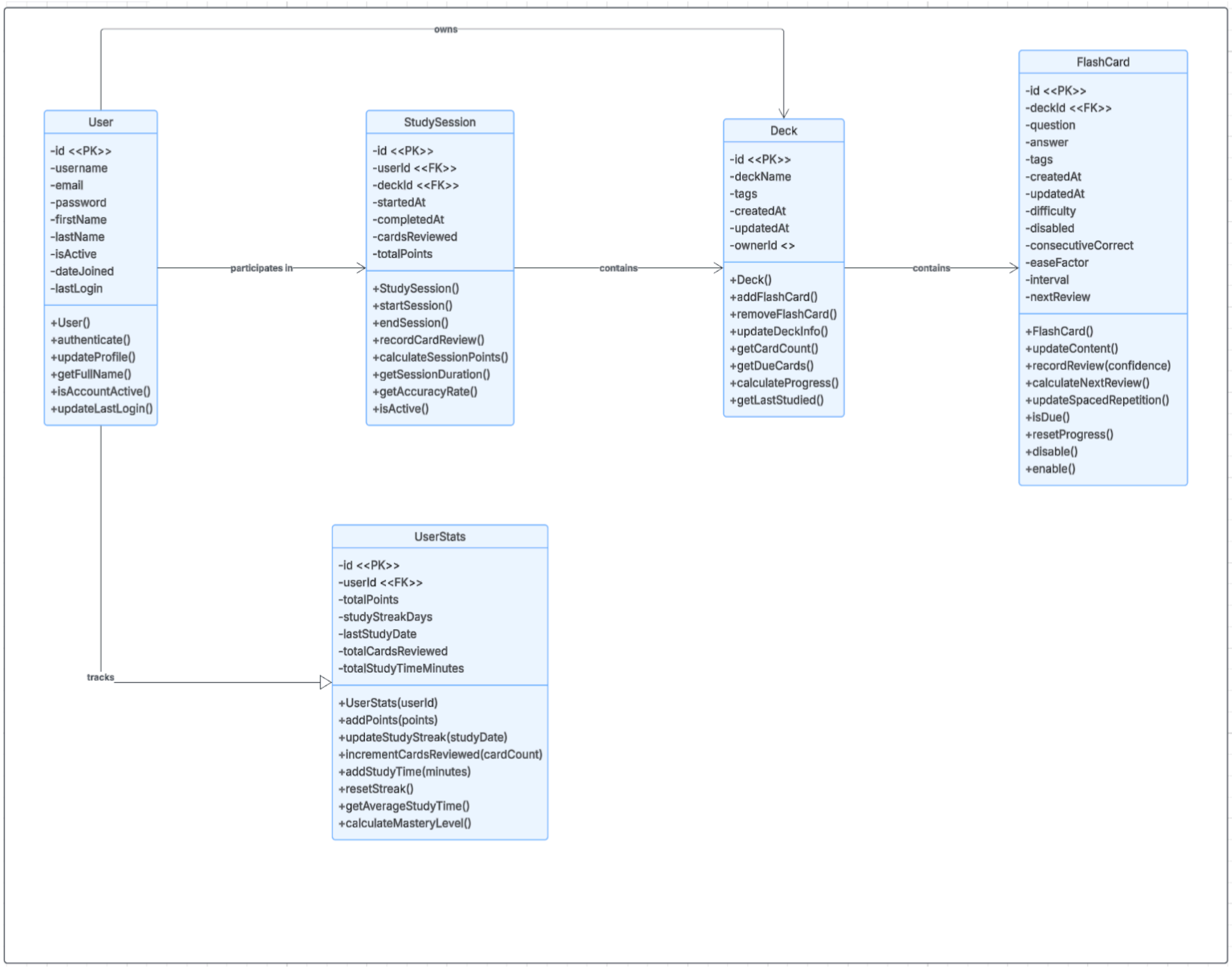
	Cont-roller	Data-base	Profile Data-base	Authen-ticator	Temp-late engine	Editor	Dash-board Page	Quiz Form
UC1		X			X	X		
UC2	X	X			X			X
UC3		X			X		X	
UC4		X			X	X		
UC5	X		X	X				
UC6	X	X			X			X
UC7			X	X				
UC8	X	X	X					

V. Domain Model Diagram



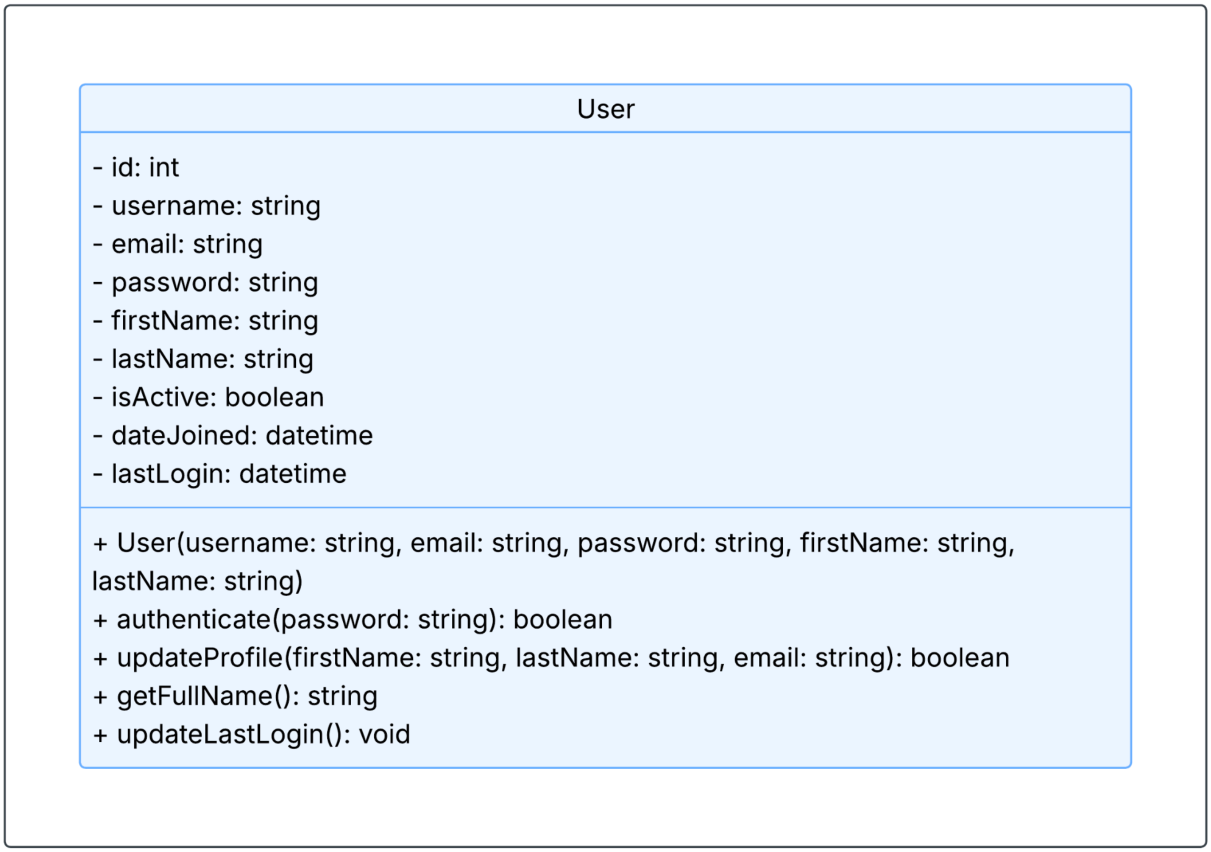
8. Class Diagrams and Interface Specification

a. Class Diagram



b. Interface specification (UML)

User Class UML Diagram



User

- id: int
- username: string
- email: string
- password: string
- firstName: string
- lastName: string
- isActive: boolean
- dateJoined: datetime
- lastLogin: datetime

+ User(username: string, email: string, password: string, firstName: string, lastName: string)
+ authenticate(password: string): boolean
+ updateProfile(firstName: string, lastName: string, email: string): boolean
+ getFullName(): string
+ updateLastLogin(): void

Definition: Represents a student or learner who uses the flashcard application to study and and track their progress in learning on a subject

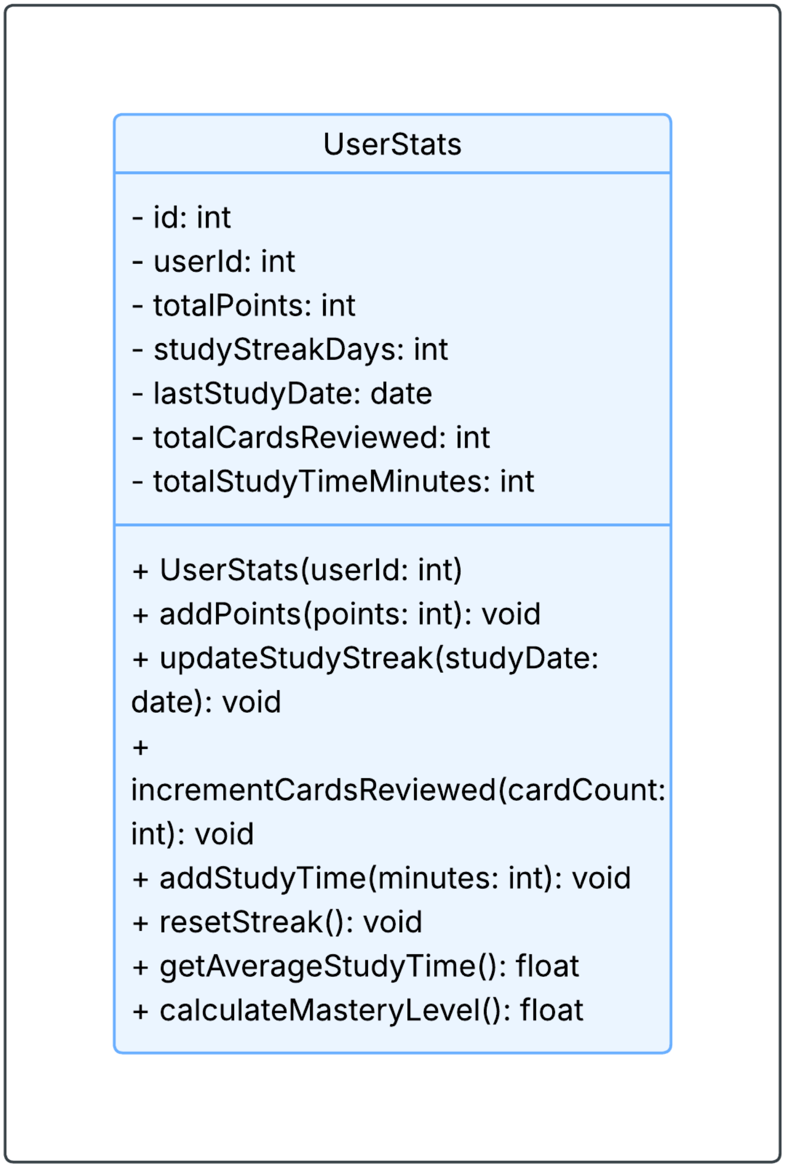
Attribute Definitions:

- id: Unique identifier for the user account
- username: Unique display name chosen by the user
- email: User's email address for authentication and communication
- password: Encrypted password for account security
- firstName: User's given name
- lastName: User's family name
- isActive: Flag indicating whether the account is currently active
- dateJoined: Timestamp when the user created their account
- lastLogin: Timestamp of the user's most recent login

Operation Definitions:

- User(): Constructor that creates a new user account with provided credentials and personal information
- authenticate(): Verifies if the provided password matches the user's stored password for login purposes
- updateProfile(): Modifies the user's personal information including name and email address
- getFullName(): Combines first and last name into a single formatted string for display purposes
- isAccountActive(): Checks and returns whether the user's account is currently active and able to log in
- updateLastLogin(): Records the current timestamp as the user's most recent login time

UserStats Class UML Diagram



UserStats

- id: int
- userId: int
- totalPoints: int
- studyStreakDays: int
- lastStudyDate: date
- totalCardsReviewed: int
- totalStudyTimeMinutes: int

+ UserStats(userId: int)
+ addPoints(points: int): void
+ updateStudyStreak(studyDate: date): void
+ incrementCardsReviewed(cardCount: int): void
+ addStudyTime(minutes: int): void
+ resetStreak(): void
+ getAverageStudyTime(): float
+ calculateMasteryLevel(): float

Definition: Tracks learning statistics and gamification metrics for each user's study progress

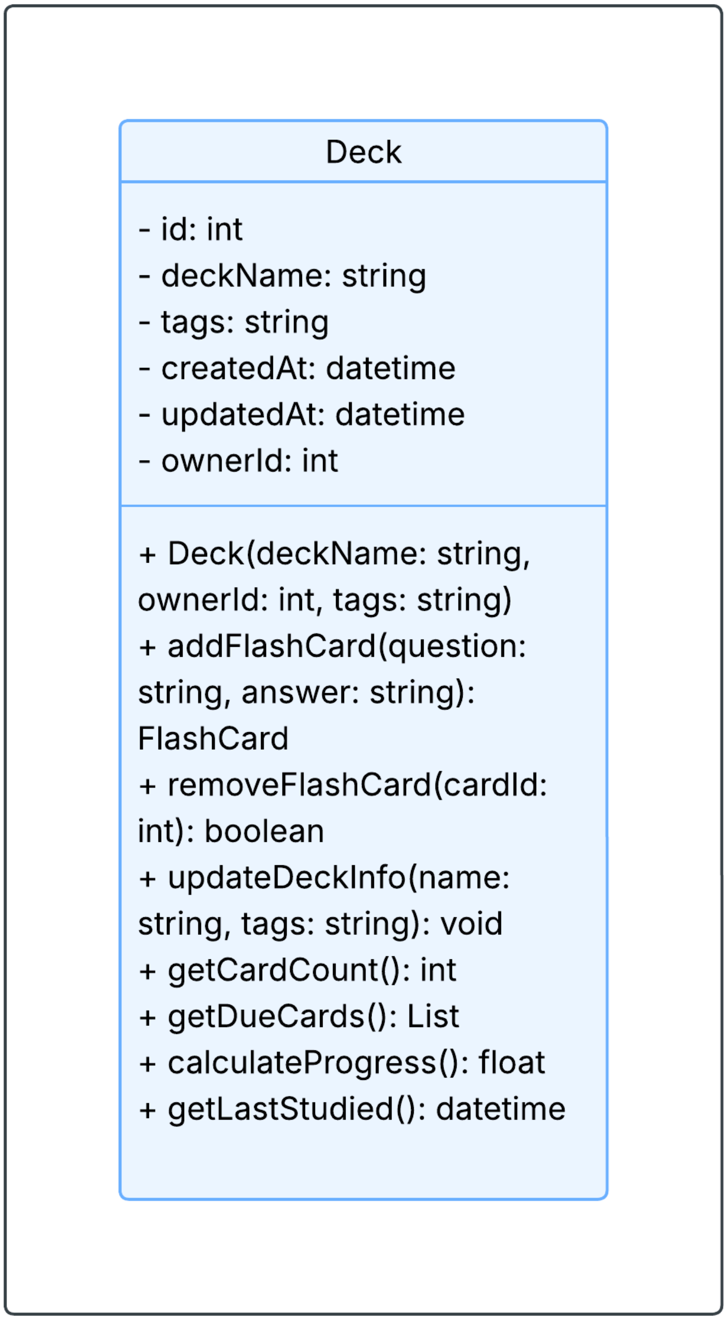
Attribute Definitions:

- id: Unique identifier for the statistics record
- userId: Foreign key linking to the User who owns these statistics
- totalPoints: Cumulative points earned through study activities
- studyStreakDays: Number of consecutive days the user has studied
- lastStudyDate: Most recent date when the user completed a study session
- totalCardsReviewed: Lifetime count of flashcards the user has reviewed
- totalStudyTimeMinutes: Total time spent studying across all sessions

Operation Definitions:

- UserStats(): Constructor that creates a new statistics record for a user with initial values set to zero
- addPoints(): Increases the user's total point balance by the specified amount earned from study activities
- updateStudyStreak(): Calculates and updates the consecutive day streak based on the provided study date
- incrementCardsReviewed(): Adds the specified number of cards to the user's lifetime review count
- addStudyTime(): Increases the total study time by the specified number of minutes from a session
- resetStreak(): Sets the study streak back to zero when the user breaks their consecutive study pattern
- getAverageStudyTime(): Calculates and returns the average time spent per study session
- calculateMasteryLevel(): Computes an overall learning progress percentage based on user's study statistics

Deck Class UML Diagram



Deck

- id: int
- deckName: string
- tags: string
- createdAt: datetime
- updatedAt: datetime
- ownerId: int

+ Deck(deckName: string, ownerId: int, tags: string)
+ addFlashCard(question: string, answer: string): FlashCard
+ removeFlashCard(cardId: int): boolean
+ updateDeckInfo(name: string, tags: string): void
+ getCardCount(): int
+ getDueCards(): List
+ calculateProgress(): float
+ getLastStudied(): datetime

Definition: A collection of related flashcards organized around a specific topic or subject area for focused studying.

Attribute Definitions:

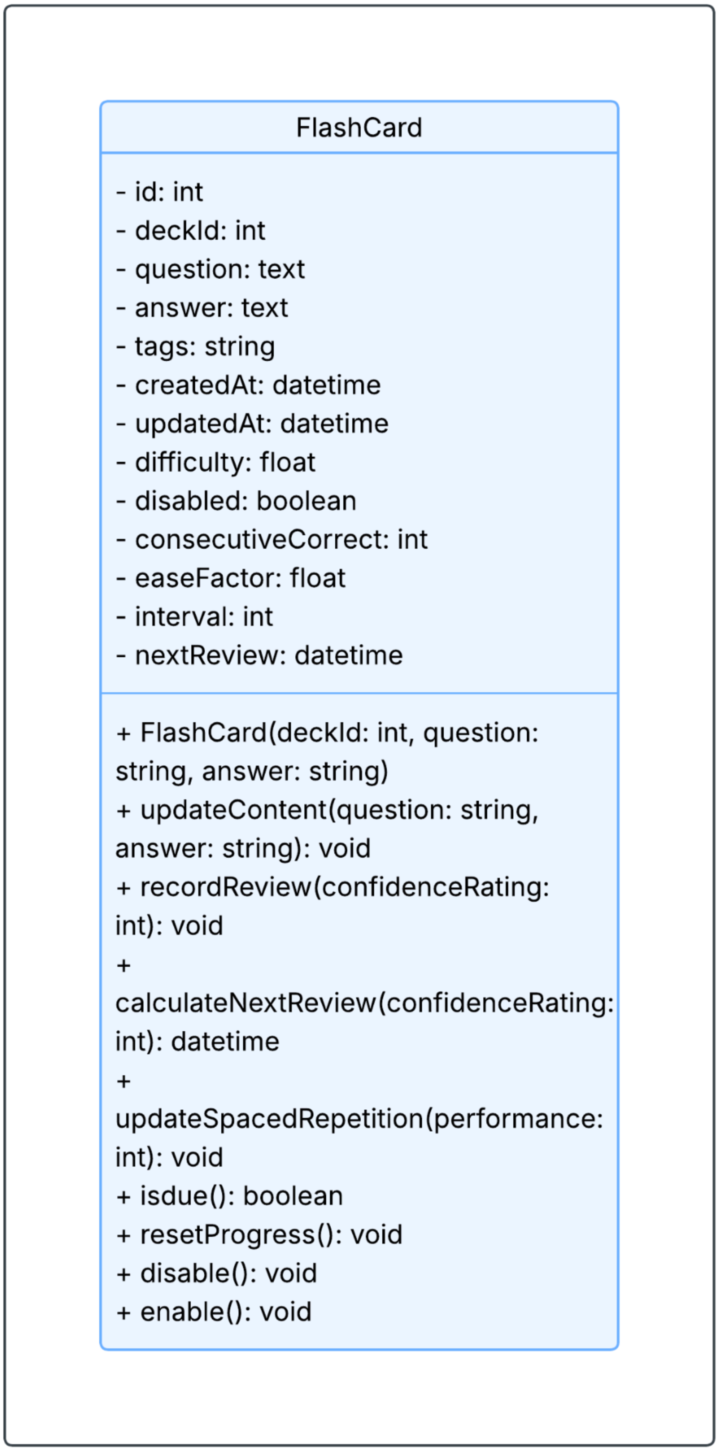
- id: Unique identifier for the deck
- deckName: Descriptive title for the flashcard collection

- tags: Comma-separated keywords for organizing and searching decks
- createdAt: Timestamp when the deck was first created
- updatedAt: Timestamp of the most recent modification to the deck
- ownerId: Foreign key linking to the User who created this deck

Operation Definitions:

- Deck(): Constructor that creates a new flashcard deck with specified name, owner, and optional tags
- addFlashCard(): Creates and adds a new flashcard to this deck with the provided question and answer
- removeFlashCard(): Deletes the specified flashcard from the deck and returns success status
- updateDeckInfo(): Modifies the deck's name and tags while updating the modification timestamp
- getCardCount(): Returns the total number of flashcards currently in this deck
- getDueCards(): Retrieves a list of all flashcards that are scheduled for review today
- calculateProgress(): Computes the percentage of cards in the deck that have been mastered
- getLastStudied(): Returns the timestamp when any card in this deck was last reviewed

FlashCard Class UML Diagram



FlashCard

- id: int
- deckId: int
- question: text
- answer: text
- tags: string
- createdAt: datetime
- updatedAt: datetime
- difficulty: float
- disabled: boolean
- consecutiveCorrect: int
- easeFactor: float
- interval: int
- nextReview: datetime

+ FlashCard(deckId: int, question: string, answer: string)
+ updateContent(question: string, answer: string): void
+ recordReview(confidenceRating: int): void
+ calculateNextReview(confidenceRating: int): datetime
+ updateSpacedRepetition(performance: int): void
+ isdue(): boolean
+ resetProgress(): void
+ disable(): void
+ enable(): void

Definition: An individual study item containing a question-answer pair with spaced repetition scheduling data.

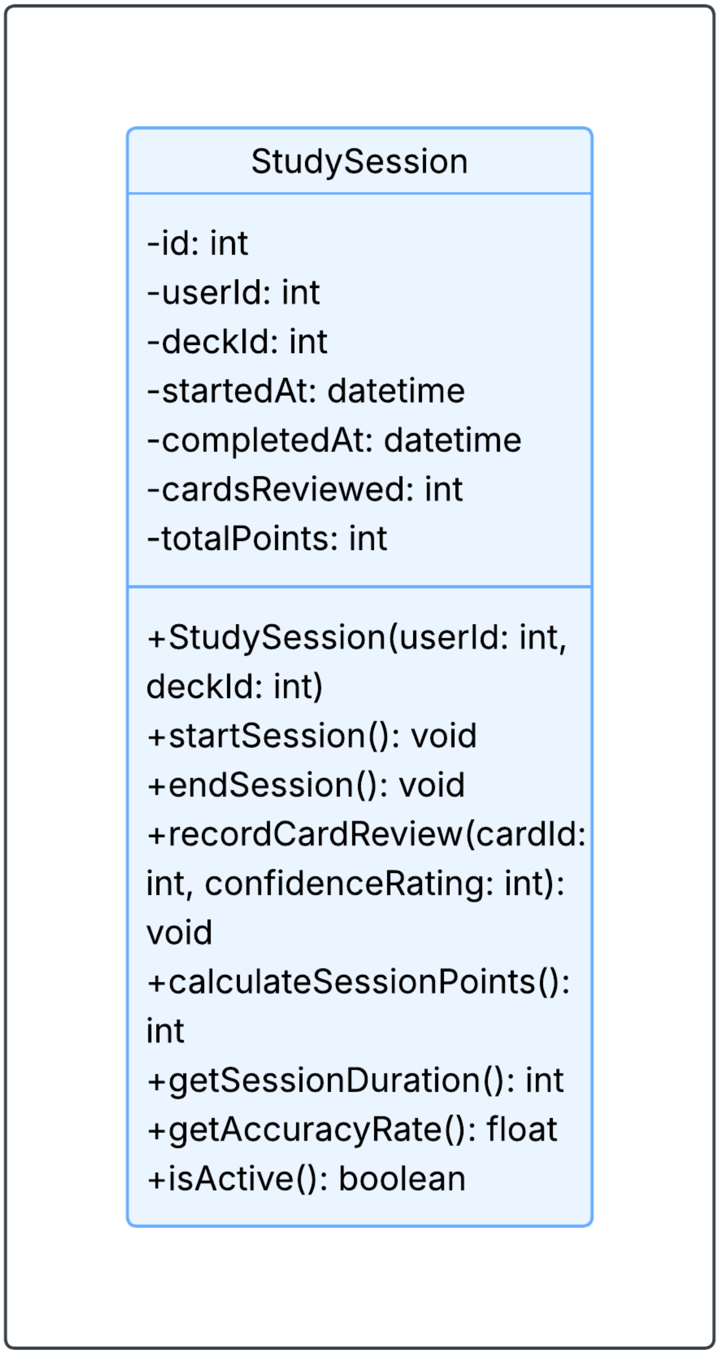
Attribute Definitions:

- id: Unique identifier for the flashcard
- deckId: Foreign key linking to the parent Deck
- question: The prompt or question text displayed to the user
- answer: The correct response or answer text
- tags: Optional keywords for additional organization within the deck
- createdAt: Timestamp when the card was created
- updatedAt: Timestamp of the most recent content modification
- difficulty: Calculated difficulty rating based on user performance (0.0-1.0)
- disabled: Flag to temporarily hide the card from study sessions
- consecutiveCorrect: Count of consecutive correct responses for spacing calculations
- easeFactor: Spaced repetition algorithm parameter affecting review intervals
- interval: Current number of days until next scheduled review
- nextReview: Calculated datetime when card should next appear for study

Operation Definitions:

- FlashCard(): Constructor that creates a new flashcard with question and answer in the specified deck
- updateContent(): Modifies the question and answer text while updating the modification timestamp
- recordReview(): Logs a study attempt with the user's confidence ratiXng and updates spaced repetition data
- calculateNextReview(): Computes when this card should next appear based on spaced repetition algorithm and performance
- updateSpacedRepetition(): Adjusts the ease factor, interval, and consecutive correct count based on user performance
- isDue(): Checks if the current date/time is past the card's scheduled next review time
- resetProgress(): Clears all spaced repetition data to start the learning process over
- disable(): Marks the card as disabled so it won't appear in study sessions
- enable(): Reactivates a disabled card so it can appear in study sessions again

UML Diagram for StudySession Class



StudySession

-id: int
-userId: int
-deckId: int
-startedAt: datetime
-completedAt: datetime
-cardsReviewed: int
-totalPoints: int

+StudySession(userId: int, deckId: int)
+startSession(): void
+endSession(): void
+recordCardReview(cardId: int, confidenceRating: int): void
+calculateSessionPoints(): int
+getSessionDuration(): int
+getAccuracyRate(): float
+isActive(): boolean

Definition: Represents a single study period where a user reviews flashcards, tracking performance and time spent.

Attribute Definitions:

- id: Unique identifier for the study session
- userId: Foreign key linking to the User conducting the session
- deckId: Foreign key linking to the Deck being studied
- startedAt: Timestamp when the study session began
- completedAt: Timestamp when the session was finished (null if ongoing)
- cardsReviewed: Number of flashcards reviewed during this session
- totalPoints: Points earned during this specific session

Operation Definitions:

- StudySession(): Constructor that creates a new study session for a user and deck with start time recorded
- startSession(): Initializes the session by recording the start time and preparing for card reviews
- endSession(): Finalizes the session by recording completion time and calculating final statistics
- recordCardReview(): Logs that a specific card was reviewed with a given confidence rating during this session
- calculateSessionPoints(): Computes the total points earned based on cards reviewed and performance ratings
- getSessionDuration(): Calculates and returns the time elapsed between session start and end in minutes
- getAccuracyRate(): Determines the percentage of cards that were rated as "easy" or "good" during the session
- isActive(): Checks whether the session is currently ongoing (has start time but no completion time)

c. Traceability Matrix (Detailed Concept Transformations)

1. Student to User

Rationale: The original domain identified “students” as primary users, but we want to target a larger audience. The User class includes college students, high school students, and even adult learners.

Key Changes:

- Generic Terminology
- Attributes for personalization and security
- Added isActive flag for account management beyond the original concept

2. FlashCard

Rationale: The question-answer card concept was improved to support spaced repetition learning. The domain concept was simple, but implementing it has lead us to scheduling and performance tracking.

Key Additions:

- Spaced repetition algorithm parameters (easeFactor, interval, nextReview)
- Performance tracking (consecutiveCorrect, difficulty)
- Lifecycle management (disabled, timestamps)

Justification: Simple flashcards are not optimized for learning. The enhanced class implements proven learning strategies

3. Deck

Rationale: "Deck" provides domain-specific language familiar to flashcard users while adding organizational and ownership abilities not present in the initial domain model.

Key Enhancements:

- Ownership tracking for multi-user system
- Metadata for organization (tags) and auditing (timestamps)
- Methods for progress calculation and due card identification

4. Studying Activity to StudySession

Rationale: The domain described "studying" as an activity, but we discovered the need f or trackable sessions. This class is all about how engagement, time and studying fit together

Why This Class Was Added:

- Enables gamification through session-based point systems
- Provides data for learning analytics and progress visualization
- Supports interrupted study sessions with state management

5. Progress/Statistics to UserStats

Rationale: Originally, user progress was attached as part of the student's profile. However, the gamification requirements (streaks, points, analytics) was creating some tightly coupled code that was bloating quickly

Design Decision Benefits:

- Separation of concerns: User identity vs. learning analytics
- Ease of adding new statistical measures

Classes Not Directly Traceable to Original Domain

StudySession

Why Added: While "studying" was a clear domain activity, it wasn't explicitly modeled in the original domain analysis. Requirements analysis revealed that session tracking is good for:

- Meaningful gamification
- Providing progress feedback
- Supporting learning analytics
- Resuming sessions

UserStats

Why Separated: Originally created as part of the User profile,our gamification of studying had a clear conflict of interest with its current location. While having a fat model in Django can be desirable, it became clear that it needed to be split. Separating these concerns:

- Keeps User class focused on identity and authentication
- Allows for complex analytics without bloating the User class
- Enables future enhancements like comparative statistics and leaderboards

Validation Against Use Cases

The class design directly supports the identified use cases:

- UC-1 (Create Deck): Deck and FlashCard classes provide creation and management abilities
- UC-2 (Study Flashcards): StudySession handles the learning process while FlashCard manages spaced repetition
- UC-3 (View Dashboard): UserStats aggregates progress data for display

- UC-4 (Edit Flashcards): FlashCard and Deck classes support content modification

9. Design Patterns

Design patterns are used in Quiz of Fury to improve maintainability, scalability, and readability by providing a tested solution and shared language around common problems.

a. Decorator Pattern
The decorator pattern is built into Django and is used primarily for adding authentication and authorization in view routes. By tagging a view route with the @login_required decorator, we enforce that a user cannot access certain features of the site without authenticating

```
@login_required
def share_deck(request, deck_id):
    """Copies a deck and all its cards, assigning the copy to the current user."""
    # Get the deck or 404
    original_deck = get_object_or_404(Deck, id=deck_id)
…
```

b. Factory Pattern
The factory pattern is utilized to abstract object creation. In Quiz of Fury, this is used to instantiate database models, which gives us a host of premade functions for saving and loading cards and decks. This pattern will also allow us to change database providers while changing a minimum of code

```
    """Display all available decks on the dashboard"""
    decks = Deck.objects.filter(owner=request.user).order_by('-updated_at')
```

c. Model-Template-View Pattern
Django is built around the Model-Template-View pattern, and it is used extensively in Quiz of Fury. This pattern separates data (models), business logic (views) and presentation (templates). The models section encapsulates the database models that we have created (decks and cards) along with the built in user model. Views drive the logic that show, add, edit, and share cards. The templates render the HTML and Javascript that the user will interact with in the browser.

```
# This sample shows view logic returning a redirect to a template with a model attached
return redirect('deck_detail', deck_id=new_deck.id)
```

10. Object Constraint Language Contracts

FlashCard

The FlashCard class represents a question/answer pair used in study sessions. It includes spaced repetition attributes managed by the scheduler.

Invariants:

- Difficulty must be between 1 and 5
  inv DifficultyRange: self.difficulty >= 1 and self.difficulty <= 5

- Ease factor must not fall below 1.3
  inv EaseFactorMinimum: self.ease_factor >= 1.3

- A FlashCard must belong to a deck
  inv DeckAssigned: self.deckId > 0

Preconditions:

- recordReview() can only be called on enabled flashcards
  pre recordReview: not self.disabled

Postconditions:

- next_review is updated after recordReview()
  post recordReview: self.next_review > pre:self.next_review

Deck

The Deck class contains flashcards and is associated with a specific user.

Invariants:

- A deck must have an owner
  inv OwnerExists: not self.ownerId.oclIsUndefined()

- All cards in a deck must reference that deck
  inv ConsistentCardLink: self.cards->forAll(c | c.deckId = self.id)

Postconditions:

- Adding a flashcard increases the count
  post addFlashCard: self.getCardCount() = pre:self.getCardCount() + 1

StudySession

StudySession tracks when a user studies a deck and how many cards they review.

Invariants:

- A completed session must have an end time after the start
  inv ValidTimeRange: not self.completedAt.oclIsUndefined() implies self.completedAt >= self.startedAt

- Cards reviewed and points earned must be non-negative
  inv NonNegativeStats: self.cardsReviewed >= 0 and self.totalPoints >= 0

Preconditions:

- endSession() requires the session to have started
  pre endSession: not self.startedAt.oclIsUndefined()

Postconditions:

- Session is marked completed and points are calculated
  post endSession: self.completedAt <> null and self.totalPoints = self.calculateSessionPoints()

- Each recordCardReview() call increments the review count
  post recordCardReview: self.cardsReviewed = pre:self.cardsReviewed + 1

UserStats

UserStats stores long-term performance metrics.

Invariants:

- All statistics must be non-negative
  inv ValidStats: self.totalPoints >= 0 and self.totalCardsReviewed >= 0 and self.totalStudyTimeMinutes >= 0

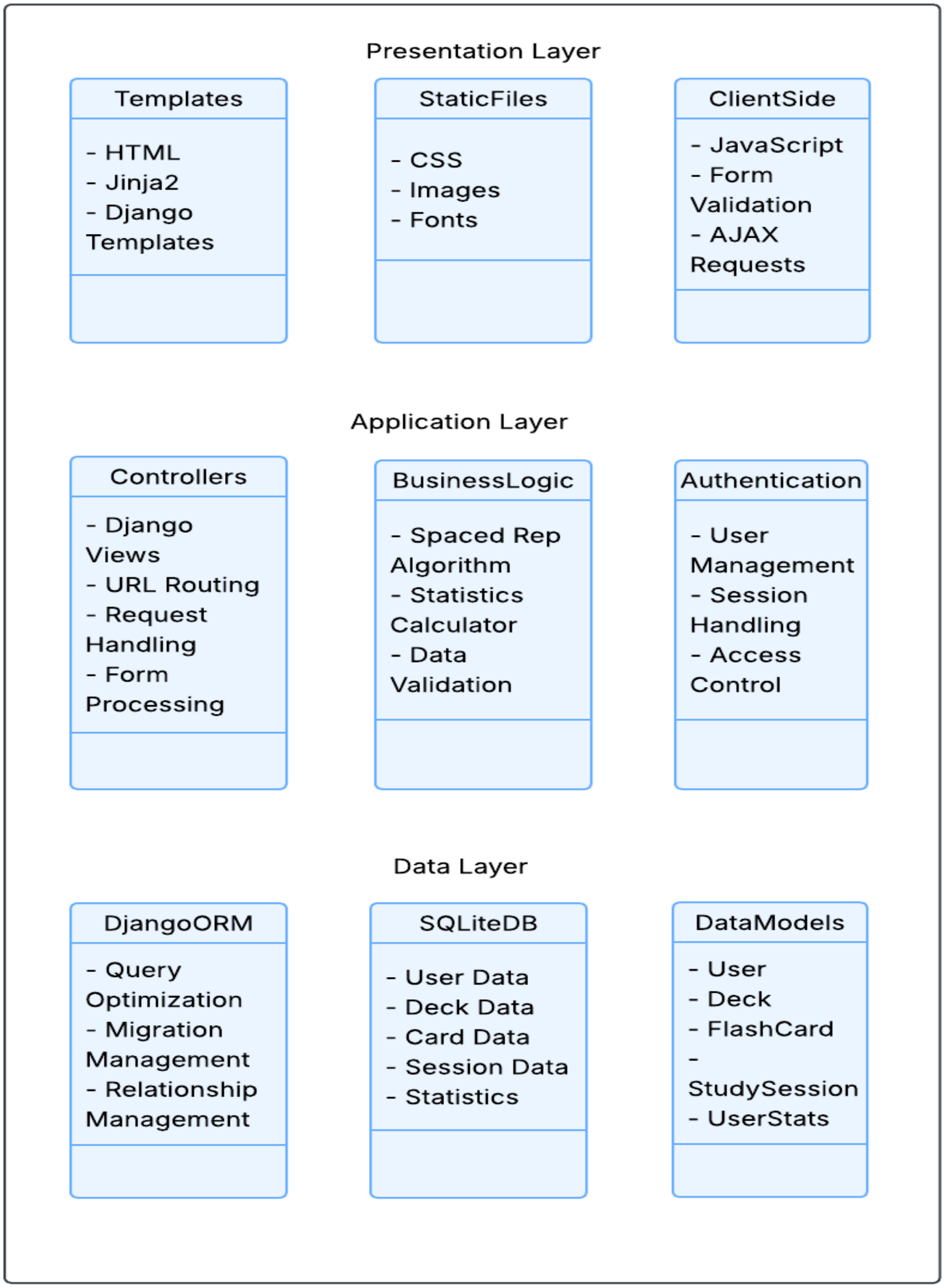
- Points earned must not exceed total cards reviewed
  inv PointsLogical: self.totalPoints <= self.totalCardsReviewed

Postconditions:

- Updating with a new session adds to totals
  post updateStats:
  self.totalCardsReviewed = pre:self.totalCardsReviewed + session.cardsReviewed and
  self.totalPoints = pre:self.totalPoints + session.totalPoints

11. System Architecture

a. Identifying Subsystems



b. Architecture Styles

Primary Style: Model-View-Template (MVT)

- Django's implementation of MVC pattern
- **Models:** Define data structure and business logic
- **Views:** Handle HTTP requests and coordinate between models and templates
- **Templates:** Present data to users

Secondary Style: Layered Architecture

- **Presentation Layer:** HTML templates with CSS and JavaScript
- **Application Layer:** Django views and business logic
- **Data Layer:** SQLite database with Django ORM

Design Patterns:

- **Active Record:** Django models encapsulate data and behavior
- **Template Method:** Django's class-based views provide extensible patterns
- **Decorator:** @login_required and other Django decorators for cross-cutting concerns

c. Mapping Subsystems to Hardware

Single-Machine Deployment (MVP Approach):

- **Development:** Local machines (Mac/Windows with Docker)
- **Database:** SQLite file on same server (suitable for MVP scale)
- **Static Files:** Served directly by Django)

Client-Server Architecture:

- **Client:** Web browsers (Chrome, Firefox, Safari, mobile browsers)
- **Server:** Django application server handling HTTP requests
- **Database:** Local SQLite database

Technology Stack Summary:

- **Backend:** Django 5.2, Python 3.12
- **Database:** SQLite (development)
- **Frontend:** HTML5, CSS3, JavaScript
- **Dependency Management:** Poetry
- **Containerization:** Docker & Docker Compose

- **Version Control:** Git
- **Testing:** Django TestCase, Coverage.py

d. Connectors and Network Protocols

HTTP/HTTPS Protocol:

- **GET requests:** Retrieve pages, deck lists, study interfaces
- **POST requests:** Form submissions for deck creation, card additions, study ratings
- **Session Management:** Django's cookie-based sessions for user authentication
- **Static Assets:** CSS, JavaScript, and images served via HTTP

Why HTTP/HTTPS:

- Standard web protocol supported by all browsers
- Django provides built-in HTTP handling
- HTTPS ensures secure authentication and data transmission

Data Exchange Formats:

- **HTML:** Primary content delivery format
- **JSON:** AJAX responses for dynamic interactions
- **Form Data:** Standard HTML form submissions
- **Session Cookies:** User authentication state

e. Global Control Flow

Execution Orderness

Event-Driven System:

- Application waits for user HTTP requests
- Users can navigate in any order (dashboard → study → create deck)
- No enforced sequence of operations
- Asynchronous request handling allows multiple concurrent users

Time Dependency

Spaced Repetition Timer:

- Background calculations determine when cards are due for review
- No real-time constraints (tolerance: ±1 hour acceptable)
- Daily cleanup tasks to update card schedules
- No periodic execution required during normal operation

System Response Type:

- Event-response system with no real-time requirements
- Typical response time: < 1 second for web requests
- Study session timing is user-paced, not system-paced

f. Data Flow Architecture

Study Session Flow:

1. User Request → Django View → Authentication Check
2. View → ORM → SQLite (fetch due cards)
3. Spaced Repetition Algorithm → Calculate next review dates
4. Template Engine → Render study interface
5. User Response → View → Algorithm Update → Database

Deck Creation Flow:

1. User Form Submission → Django Form Validation
2. View Controller → Model Creation → Database Storage
3. Success Response → Template Rendering → User Feedback

Authentication Flow:

1. Login Request → Django Auth → Session Creation
2. Session Cookie → Browser Storage
3. Subsequent Requests → Session Validation → Access Control

g. Hardware Requirements

Client Requirements:

- **Display:** Modern smartphone, responsive up to desktop
- **Browser:** Modern web browser
- **Network:** Minimum 1 Mbps for smooth operation
- **Storage:** 5MB browser cache for offline-capable features

Server Requirements (MVP):

- **CPU:** 1-2 cores
- **Memory:** 512MB RAM (basic Django application)
- **Storage:** 1GB disk space (application + database + logs)
- **Network:** 10 Mbps upload bandwidth

- **Platform:** Linux server with Python 3.12 support

Development Environment:

- **Docker:** Container orchestration for consistent environments
- **Poetry:** Python dependency management and packaging
- **Python 3.12+:** Django framework requirement
- **Git:** Version control and collaboration
- **SQLite:** Development database (file-based)
- **Code Editor:** VS Code, PyCharm, or similar

12. Algorithms and Data Structures

a. Algorithms

Spaced Repetition Algorithm Implementation

Our system implements a version of the **SM-2 (SuperMemo 2) algorithm** for spaced repetition, which is a well-established algorithm for memory retention through calculated review intervals.

Algorithm Description: The spaced repetition algorithm calculates when a flashcard should next be reviewed based on:

- User performance rating (1-5 scale)
- Previous review history
- Card difficulty (ease factor)

Algorithm Flow:

```
Input: quality (1-5), repetitions, easeFactor, interval

1. IF quality < 3 THEN
   - repetitions = 0
   - interval = 1
   ELSE
   - repetitions = repetitions + 1
2. IF repetitions = 1 THEN
```

```
  - interval = 1
  ELSE IF repetitions = 2 THEN
  - interval = 6
  ELSE
  - interval = floor(interval * easeFactor)
3. easeFactor = easeFactor + (0.1 - (5 - quality) * (0.08 + (5 - quality) * 0.02))
  IF easeFactor < 1.3 THEN easeFactor = 1.3
4. nextReview = currentDate + interval (days)
Output: updated interval, easeFactor, repetitions, nextReview
```

Implementation Rationale:

- We chose SM-2 because
  - Well-documented and proven effective
  - Simpler to implement and debug in our MVP timeframe

Other Algorithms:

- **Card Selection Algorithm**: Orders due cards by priority (overdue cards first, then by difficulty)
- **Progress Calculation**: Computes mastery percentage based on consecutive correct answers and review frequency

b. Data Structures

Our system uses several data structures, primarily leveraging Django's ORM which abstracts underlying database structures:

Primary Data Structures:

1. Django QuerySets (Database-backed)

- **Purpose**: Efficiently retrieve and filter flashcards, study sessions, and user data
- **Criteria**: Chosen for built-in optimization, lazy evaluation, and database integration
- **Performance**: O(1) for indexed lookups, O(log n) for sorted queries

2. Python Lists and Dictionaries

- **Purpose**: Temporary storage for study session data and algorithm calculations
- **Example**: due_cards = list(Deck.objects.filter(nextReview__lte=today))
- **Criteria**: Native Python structures for simplicity and readability

3. Session Storage (Dictionary-based)

- **Purpose**: Track current study session state without database writes on every interaction
- **Implementation**: Django session framework using signed cookies
- **Criteria**: Balances performance (no DB writes) with data persistence

Data Structure Decision Criteria:

Requirement	Structure Choice	Rationale
Persistent storage	Django Models/QuerySets	Built-in ORM, automatic SQL optimization
Temporary calculations	Python lists/dicts	Simple, readable, sufficient performance
User session state	Session storage	Stateless HTTP handling, automatic cleanup
Card ordering	Python lists with custom sorting	Flexibility for algorithm experimentation

c. Concurrency

Current Implementation: Single-threaded Django application
Rationale: For our MVP with expected <100 concurrent users, single-threaded processing is more than enough

13. Design of Tests

a. Unit Testing Strategy

Core Model Tests (test_models.py):

FlashCard Model Tests:

```
class FlashCardTestCase(TestCase):
```

def test_calculate_next_review_easy_rating(self):

Test that rating 5 increases interval significantly

def test_is_due_functionality(self):

Test due date calculation accuracy

Scheduler Tests:

class SchedulerTest(TestCase):

def test_modify_card_four_difficulty_no_repititions(self):
Test modifying spaced repetition algorithm (ease factor) for an easy card that has not been repeated

def test_modify_card_five_difficulty_no_repititions(self):
Test modifying spaced repetition algorithm (ease factor) for a very easy card that has not been repeated

def test_modify_card_one_difficulty_no_repititions(self):
Test modifying spaced repetition algorithm (ease factor) for a very difficult card that has not been repeated

def test_modify_card_five_difficulty_second_repitition(self):
Test modifying ease_factor for a very easy card that has had 2 repetitions

def test_modify_card_five_difficulty_third_repitition(self):
Test modifying ease factor and interval for a very easy card that has had 3 repetitions

def test_modify_card_two_difficulty_second_repitition(self):
Test modifying a difficult card that has had two repetitions (as an easy ranked card)

def test_modify_card_minimum_ease_factor(self):
Test to make sure that cards cannot drop below a minimum value for ease factor

def test_sort_cards(self):
Test to make sure that a list of cards is appropriately sorted - that the first card is the card with the closest study date, then by ease_factor descending, then by ID (for a guaranteed unique value if everything else matches)

b. Test Coverage Analysis

Target Coverage: 80% minimum for core functionality

Coverage Priorities:

1. **Critical (90%+ coverage)**: Deck/card CRUD operations, study session logic, spaced repetition algorithm
2. **Important Features (80%+ coverage)**: User authentication, statistics calculation
3. **Supporting Features (60%+ coverage)**: UI form validation, error handling

Coverage Tools: Django's built-in coverage tools with coverage.py

c. Integration Testing Strategy

Testing Phases:

Phase 1: Component Integration

- Test Model-View integration
- Verify form submission workflows
- Test template rendering with dynamic data

Phase 2: User Workflow Integration

1. User logs in

2. Selects deck

3. Studies all due cards

4. Completes session

5. Views updated statistics

Verify: all database updates, session state management

-

Non-Functional Requirements Testing:

Performance Testing:

- **Load Testing**: Simulate 10 concurrent users studying
- **Response Time**: Verify <1 second page loads

Usability Testing:

- **5-Minute Test**: New users can create deck and start studying within 5 minutes
- **Mobile Usability**: Touch targets meet 44px minimum size
- **Accessibility**: Verify keyboard navigation and screen reader compatibility

Success Criteria for Testing:

- All unit tests pass with >80% coverage
- Integration tests demonstrate complete user workflows
- Performance tests show < 1 second response times

14. History of Work

a. Previous work

i. Database models created
ii. Authentication created
iii. Template and views created for basic deck and card creation

b. Current work (changes added to from demo 1 to demo 2)

i. Spaced repetition algorithm implemented
ii. Card difficulty ranking implemented
iii. Deck sharing implemented
iv. Deck ownership added

c. Future work, additional features (not planned for demo 2)

i. Gamification features
ii. Study statistics for individual decks
iii. Leaderboards for shared decks
iv. Support for images on flash cards

References

- Django Software Foundation. (2025). *Django Documentation*. Retrieved from https://docs.djangoproject.com/
- Django QuerySet (2025) *Django Documentation* *https://docs.djangoproject.com/en/5.2/ref/models/querysets/*
- SM-2 Algorithm *Wikipedia https://en.wikipedia.org/wiki/SuperMemo*
- Quizlet. (2025). *Quizlet Help Center*. Retrieved from https://help.quizlet.com/
- Anki. (2025). *Anki Manual*. Retrieved from https://docs.ankiweb.net/