ASE Project Report - GROUP 2

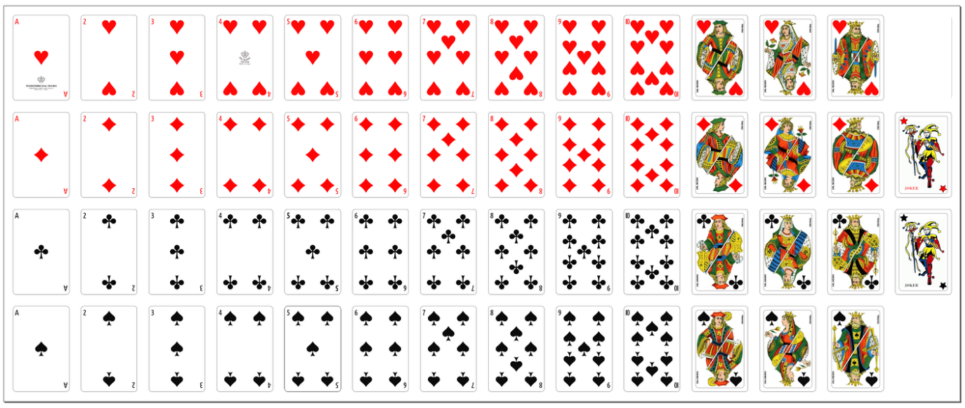
Advanced Software Engineering

First Name	Last Name	Student ID	E-Mail
Filippo	Morelli	608924	f.morelli38@studenti.unipi.it
Federico	Fornaciari	619643	f.fornaciari@studenti.unipi.it
Ashley	Spagnoli	655843	a.spagnoli9@studenti.unipi.it
Marco	Pernisco	683674	m.pernisco@studenti.unipi.it

Contents

1 Cards Overview

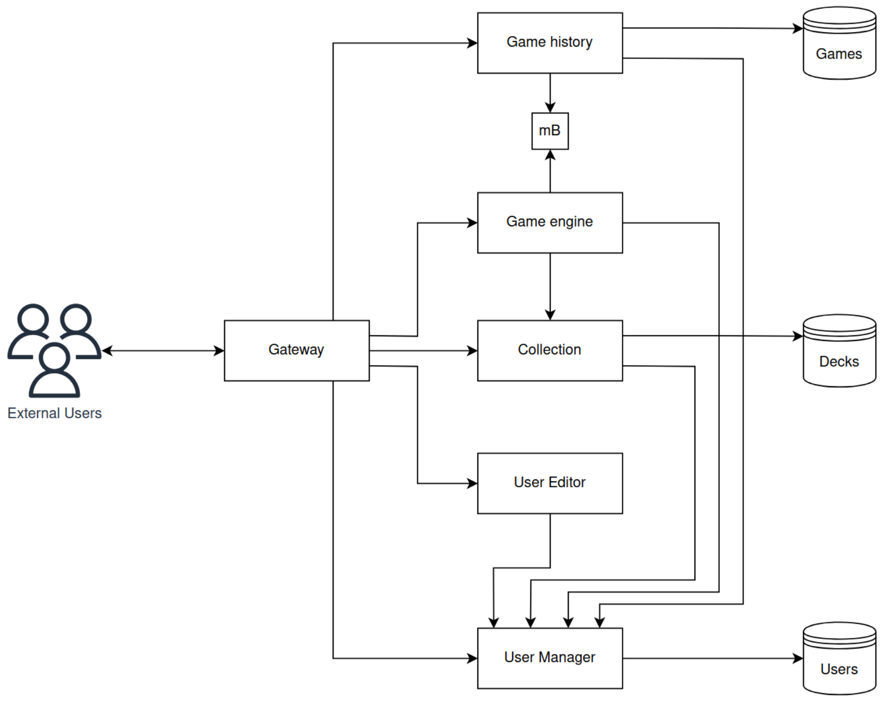
As cards we used the classical 52 + Joker deck.
As anyone could expect, the two features assigned to those cards are the number and the suit, while the Joker is a special card with no features, that beats any other card.



2 System Architecture

2.1 Architecture schema and main microservices

Below is the high-level architectural drawing of the system, illustrating the interactions between microservices.



All microservices are containerized using Docker and are written in Python. Some of them use the Flask framework and some FastAPI.
Here is a list of the main microservices of the system:

Microservice	Framework	Description
Game history	Flask	Stores and retrieves past game data
Game engine	Flask	Handles the game runtime
Collection	Flask	Handles cards and users decks
User Editor	FastAPI	Manages user profiles and settings
User Manager	FastAPI	Manages user authentication and authorization
Gateway	FastAPI	Routes requests to appropriate microservices

2.2 Design choices and microservices interactions

- **Centralized Authorization:** every microservice connects to the User Manager to verify the token header, this is for two main reasons:
  - The users can change their usernames that are stored in the User-DB, this is a way for microservices to verify they have the updated username.
  - In the future we may want to add token revocation, having a single microservice handle that would make it easier.

  This is done through the */users/validate-token* endpoint of the User Manager.
- **Shared persistance smell solutions:**
  - We connected a single microservice to each database (Games, Decks and Users).
  - Since as cards we've choosen the classical French-suited deck, that we don't expect to change, we've decided to hard-code their behavior and their images in the microservices, without relying on databases.
- **Game History -> User-Manager:** Game History needs to connect to User Manager also to get a list of (updated) usernames from a list of user-ids to return the leaderboard to users.
  This is done through the `/users/usernames-by-ids` endpoint of the User Manager.
- **User Editor:** We've separated the user profile editing and data viewing endpoints from the user manager to keep the user manager smaller and allow horizontal scalability. The User Editor then connects to dedicated endpoints of the User Manager that allow it to update user information in the DB User-DB.
- **Separated in-game logic:** The Game Engine microservice, as the name suggest, handles the whole game logic, while delegating to Game History and Collection the handling of old games and decks.
  This is done through the endpoints:
  - `/internal/update-password`
  - `/internal/update-email`
  - `/internal/update-username`
  - `/users/my-username-my-email`
- **Cards and decks:** The Collection microservice allows the users to see all the cards, their images and to handle their decks.
  The Game Engine will connect to this microservice to get the deck selected from the user: the user provides the deck's number and the Game Engine asks for the corresponding cards to the Collection. This is done through the endpoint
  `/collection/user-decks?user={user_id}&slot={slot_number}`.

- **Game History and Leaderboard:** The Game Engine delegates the Game History to store finished games. How this is handled is discussed in detail in the Additional Features section.
  This microservices then allow users to see their old matches, and allows everyone to see the leaderboard (think about a website that shows the best players to unauthenticated users).

3 User Stories

- Account
  1. Create an account SO THAT I can participate to the game
     - /users/register (Gateway → User Manager → Users DB → User Manager → Gateway)
  2. Login into the game SO THAT I can play a match
     - /users/login (Gateway → User Manager → Users DB → User Manager → Gateway)
  3. Check/modify my profile SO THAT I can update my information
     - /change-username
     - /change-password
     - /change-email
     - /view-data

     (Gateway → User Editor → User Manager → MongoDB → User Manager → User Editor → Gateway)
     Note: Profile checking is handled via token validation or login response.
  4. Be safe about my account data SO THAT nobody can steal/modify them
     - Implemented via JWT Authentication, Password Hashing in User Manager, and HTTPS communication.
- Cards
  5. See the overall card collection SO THAT I can think of a strategy
     - /collection/cards (Gateway → Collection → File System [cards.json] → Collection → Gateway)
  6. View the details of a card SO THAT I can think of a strategy
     - /collection/cards/card_id (Gateway → Collection → File System [cards.json] → Collection → Gateway)
- Game
  7. Start a new game SO THAT I can play
     - /game/match/join (Gateway → Game Engine → Gateway)
  8. Select the subset of cards SO THAT I can start a match
     - For deck creation: /collection/decks (Gateway → Collection → Collection DB → Collection → Gateway)
     - /game/deck/game_id (Gateway → Game Engine → Gateway)
  9. Select a card SO THAT I can play my turn
     - /game/play/game_id (Gateway → Game Engine → Gateway)

10. Know the score SO THAT I know if I am winning or losing
    * /game/state/game_id (Gateway → Game Engine → Gateway)
11. Know the turns SO THAT I can know how many rounds there are till the end
    * /game/state/game_id (Gateway → Game Engine → Gateway)
12. See the score SO THAT I know who is winning
    * /game/state/game_id (Gateway → Game Engine → Gateway)
13. Know who won the turn SO THAT I know the updated score
    * /game/state/game_id (Gateway → Game Engine → Gateway)
14. Know who won a match SO THAT I know the result of a match
    * /game/state/game_id (Gateway → Game Engine → Gateway)
15. Ensure that the rules are not violated SO THAT I can play a fair match
    * Handled internally by Game Engine during /game/play/game_id execution.
    * Double checks for zero trust between Game Engine and Collection.

- Others

16. View the list of my old matches SO THAT I can see how I played
    * /history/matches (Gateway → Game History → History DB → Game History → Gateway)
17. View the details of one of my matches SO THAT I can see how I played
    * /history/matches (Gateway → Game History → History DB → Game History → Gateway)

    extitNote: The list endpoint returns match details.
18. View the leaderboards SO THAT I know who are the best players
    * /history/leaderboard (Gateway → Game History → History DB → Game History → Gateway)
19. Prevent people to tamper my old matches SO THAT I have them available
    * Implemented via internal service isolation (only Game Engine writes to History via RabbitMQ/Internal API) and Database access controls.

4 Rules of the Game

4.1 Deck Building Constraints

Each player constructs a **personal deck** of exactly **9 cards**. The deck must adhere to the following constraints:

- **Composition:** The deck must contain exactly **1 Joker** and **8 Suited Cards**.
- **Suit Distribution:** You must include exactly **2 cards** from each suit (♥, ♦, ♣, ♠).
- **Cost Limit:** For any suit, the total point value of the two cards **must not exceed 15**.

Card Point Costs

When calculating your deck limits, use the following costs.

4.2 Combat Hierarchy

When two cards are played, the winner is decided by the following hierarchy.

Card Type	Ranks	Cost per Card	Example Pair Limit
Numbers	2 – 10	Face Value	10 + 5 = 15✓
Ace	A	7	$A + 8 = 15$✓
Jack	J	11	$J + 4 = 15$✓
Queen	Q	12	$Q + 3 = 15$✓
King	K	13	$K + 2 = 15$✓

1. The Combat Triangle The core mechanics function like Rock-Paper-Scissors:

- **Numbers (2–10)** beat **Aces**.
- **Aces** beat **Face Cards (J, Q, K)**.
- **Face Cards (J, Q, K)** beat **Numbers**.

2. The Joker The **Joker** beats all other cards automatically. (If both players play a Joker, it is a tie).

3. Tie-Breakers If the combat rules above do not determine a clear winner (e.g., Number vs Number, or Face vs Face), compare the specific ranks:

1. **Higher Rank Wins:** (e.g., 9 beats 6, King beats Jack).
2. **Equal Rank → Suit Priority:** If ranks are identical (e.g., ♥9 vs ♦9), check suits:

   $$\heartsuit > \diamondsuit > \clubsuit > \spadesuit$$

3. **Mirror Match:** If players play the *exact same card* (Rank and Suit), both players win the round and gain a point.

4.3 Gameplay Flow

1. **Setup:** Both players shuffle their pre-built decks.
2. **Initial Draw:** Each player draws **3 cards** to form their starting hand.
3. **The Round:**
   - **Draw Phase:** At the end of every turn, each player draws **1 card** to get back to **3**.
   - **Battle Phase:** Both players play one card **face down**, then reveal it at the same time.
   - **Scoring:** Determine the winner based on the Combat Hierarchy. The winner earns **1 point**. If it's a tie, both players earn **1 point**.
4. **Victory:** The game ends immediately when one of the players reaches **5 points**.

4.4 Example of a Match

Alice's Deck: ♥A, ♥7, ♦A, ♦7, ♣K, ♣2, ♠K, ♠2, Joker.
Bob's Deck: ♥2, ♥3, ♦2, ♦3, ♣3, ♣4, ♠2, ♠3, Joker.

Turn	Alice	Bob	Result Reasoning	Score (A-B)
1	♥A	♣K	**Ace beats Face** (Special Rule)	1 – 0
2	♦A	♦3	**Number beats Ace** (Special Rule)	1 – 1
3	♣K	♣3	**Face beats Number** (Special Rule)	2 – 1
4	♠K	♠2	**Face beats Number** (Special Rule)	3 – 1
5	♥7	♣4	Both Numbers: 7 > 4	3 – 2
6	♦7	♠3	Both Numbers: 7 > 3	3 – 3
7	♣2	Joker	**Joker beats Everything**	3 – 4
8	Joker	♥2	**Joker beats Everything**	4 – 4
9	♥7	♠3	Both Numbers: 7 > 3	**5 – 4**

5 Game Flow Guide

This guide outlines the steps to play a game, from deck creation to gameplay, assuming the user is already logged in.

5.1 1. Deck Building Phase

Before joining a game, a player must create a valid deck.

1. **Retrieve Available Cards**
   Endpoint: `GET /collection/cards`
   Returns a list of all available cards with their IDs.
2. **Create a Deck**
   Endpoint: `POST /collection/decks`
   Body (JSON):

   ```
   {
       "deckSlot": 1,          // 1-5
       "deckName": "My Deck",
       "cards": ["c1", "c2", ...] // List of 8 card IDs
   }
   ```

   Constraints:

   - Exactly 8 cards.
   - Exactly 2 cards per suit.
   - Max 15 points per suit.

5.2 2. Matchmaking Phase

Once a deck is ready, the player can join the queue.

1. **Join Queue**
   Endpoint: `POST /game/match/join`
   Adds the player to the matchmaking pool.
2. **Check Status**
   Endpoint: `GET /game/match/status`
   Poll this endpoint until a match is found.
   Returns: `{"status": "matched", "game_id": "..."}` when a game is found.

5.3 3. Gameplay Phase

After a match is found, the game begins.

1. **Select Deck**
   Endpoint: `POST /game/deck/{game_id}`
   Body (JSON): `{"deck_slot": 1}`
   Selects the deck to use for this match.

2. **Game Loop**
   Repeat these steps until the game ends:

   - **Check Game State**: `GET /game/state/{game_id}`
     Returns the current turn, board cards, and scores.
   - **Get Hand**: `GET /game/hand/{game_id}`
     Returns the cards currently in the player's hand.
   - **Play Card**: `POST /game/play/{game_id}`
     Body (JSON): `{"card": "card_id"}`
     Plays a card when it is the player's turn.

6 Testing

The project implements a testing strategy covering unit tests, integration tests, and performance tests to ensure system reliability, correctness, and scalability across all microservices.

6.1 Unit Testing

Unit tests verify individual microservice functionality in isolation using mocked dependencies. Each microservice has its own dedicated test suite executed within Docker containers to ensure consistent testing environments.

6.1.1 Test Environment Setup

Unit tests are executed using specialized Docker containers with test-specific configurations:

- **Isolated execution**: Each service runs in a dedicated test container
- **Mocked dependencies**: External service calls are mocked to test in isolation
- **Consistent environment**: Docker ensures identical test conditions across machines
- **Postman collections**: API tests are defined in structured JSON collections

6.1.2 Collection Service Unit Tests

The Collection service manages card collections and deck building with validation logic.

Test Execution:

```
# Build the test container
cd src
docker build -f collection/Dockerfile_test -t collection-test .

# Run the test container
docker run -d -p 5006:5000 --name collection-test collection-test

# Import docs/tests/collection_ut.postman_collection.json in Postman
# Run the collection

# Cleanup
docker stop collection-test
docker rm collection-test
```

Test Coverage:

- `GET /collection/cards`: Retrieves all 53 cards and validates response structure
- `GET /collection/cards/{card_id}`: Fetches individual card details (e.g., `hA`)
- `GET /collection/cards/{card_id}/image`: Serves card images
- `GET /collection/decks`: Returns user's deck collection
- `POST /collection/decks`: Creates decks with validation rules:
  - Exactly 8 cards required
  - 2 cards per suit (hearts, diamonds, clubs, spades)
  - Maximum 15 points per suit
  - Rejects invalid configurations (e.g., 3 hearts cards, point overflow)
- `DELETE /collection/decks/{deck_id}`: Removes decks and validates cleanup

Key Validation Tests:

- Invalid card IDs return 404 errors
- Deck creation with 16 points in diamonds suit fails with appropriate error
- Deck creation with wrong card count per suit is rejected
- Successfully created decks are retrievable and deletable

6.1.3 Game History Unit Tests

The Game History service tracks matches and maintains leaderboards with pagination.

Test Execution:

```
# Build and run test container
cd src
docker build -f game_history/Dockerfile_test -t history-test .
docker run -d -p 5007:5000 --name history-test history-test

# Import docs/tests/game_history_ut.postman_collection.json
# Set base URL to http://localhost:5007
# Run the collection

# Cleanup
docker stop history-test
docker rm history-test
```

Test Coverage:

- **Setup Phase**:
  - Seeds database with 12+ user mappings (alice, bob, user3-user12)
  - Bulk inserts 13 matches with varied outcomes (wins, losses, draws)

  - Creates test data for pagination validation

- **Match History Endpoint** (`GET /matches`):
  - Page 0: Returns up to 10 matches with row numbers 1-10
  - Page 1: Returns next batch with row numbers 11+, no overlap with page 0
  - Unauthorized access without JWT token returns 401

- **Leaderboard Endpoint** (`GET /leaderboard`):
  - Page 0: Returns top 10 players ranked by points
  - Page 1: Returns next 10 players with no username overlap
  - POST request to leaderboard returns 405 (Method Not Allowed)

Pagination Validation: The tests verify that pagination works correctly by checking:

- Each page contains at most 10 entries
- Row numbers increment correctly across pages
- No duplicate entries between consecutive pages
- Usernames in leaderboard pages are distinct

6.1.4 User Manager Unit Tests

The User Manager service handles authentication and user registration.

Test Execution:

```
# Build and run test container
cd src
docker build -f user-manager/Dockerfile_test -t user-manager-test .
docker run -d -p 5004:5000 --name user-manager-test user-manager-test

# Import docs/tests/user_manager_ut.postman_collection.json
# Set base URL to http://localhost:5004
# Run the collection

# Cleanup
docker stop user-manager-test
docker rm user-manager-test
```

Test Coverage:

- **Registration** (`POST /users/register`):
  - Success case: Creates new user with username, password, email
  - Returns 201 Created with confirmation message
  - Duplicate username: Returns 400 Bad Request
  - Duplicate email: Returns 400 Bad Request

- **Login** (`POST /users/login`):
  - Success case: Returns 200 OK with JWT token (50+ characters)
  - Token type is "bearer"
  - Invalid password: Returns 401 Unauthorized
  - Error detail: "Invalid username or password"
  - Saves token for subsequent authenticated requests

6.2 Integration Testing

Integration tests verify the complete workflow across all microservices (User Manager, Collection, Game Engine, Game History) through the API Gateway. These tests ensure proper service communication, data consistency, and end-to-end functionality.

6.2.1 Test Environment

Integration tests require all services running:

```
cd src
docker compose up --build
```

The tests use the API Gateway endpoint (`https://localhost:8443`) and are organized into test suites covering different aspects of the system.

6.2.2 Test Suites

IT-001: Complete Game Workflow - Happy Path
Validates the entire user journey from registration to game completion:

1. User registration and JWT token acquisition for Alice and Bob
2. Deck creation with valid card configurations
3. Deck retrieval verification
4. Matchmaking initialization
5. Game state progression

IT-002: Authentication & Authorization
Tests security and access control:

- Invalid login credentials return 401 Unauthorized
- Accessing decks without token returns 401/403
- Invalid JWT tokens are rejected
- Duplicate username registration returns 400 with appropriate error

IT-003: Deck Validation
Ensures deck building rules are enforced across services:

- Decks with wrong number of cards are rejected (must be 8)
- Suit distribution validation (2 cards per suit)
- Point limits per suit (maximum 15 points)
- Invalid card IDs are caught

IT-004: Game History & Leaderboard
Verifies historical data tracking:

- Match history retrieval for authenticated users
- Paginated results with consistent ordering
- Leaderboard rankings reflect game outcomes

- Specific match details are accessible

IT-005: Cross-Service Data Consistency
Tests data isolation and integrity:

- Users can only access their own decks
- Bob's deck list does not include Alice's decks
- User IDs are consistently used across services
- Game results properly reference both players

IT-007: Error Handling & Edge Cases
Validates robust error handling:

- Deleting non-existent decks returns 404
- Malformed request bodies return 400/422
- Services gracefully handle invalid inputs
- Appropriate error messages guide users

6.2.3 Test Execution

Using Postman:

1. Import `docs/tests/integration.postman_collection.json`
2. Configure environment variables:
   - `gateway_url`: `https://localhost:8443`
   - User credentials are auto-generated with timestamps
3. Run entire collection or specific test suites
4. Tests must run in order as later tests depend on earlier state

Using Python Script: A complete game simulation can be executed programmatically:

```
cd src
python test_match.py
```

This script performs:

1. Registers two random users with unique credentials
2. Creates valid decks for both players
3. Initiates matchmaking and pairs the users
4. Simulates a complete game with card plays
5. Displays comprehensive game statistics including:
   - Round-by-round results
   - Final scores
   - Winner determination
   - Game duration

6.3 Performance Testing with Locust

Performance tests simulate realistic user load to measure system behavior under concurrent access and identify bottlenecks. The tests use Locust, a Python-based load testing framework.

6.3.1 Test Scenarios

The performance test suite simulates a complete user workflow representing realistic usage patterns:

1. **User Registration**: Creates new accounts with random credentials
2. **Authentication**: Performs login and JWT token generation
3. **Deck Creation**: Builds valid 8-card decks following game rules
4. **Matchmaking**: Joins queue and waits for opponent matching
5. **Gameplay**: Simulates complete game sessions with:
   - Deck selection for matched game
   - Iterative card plays
   - Hand retrieval between turns
   - Game state validation
6. **History Access**: Queries match history and leaderboard data

6.3.2 User Types

Three user types with different think times simulate varied usage patterns:

- **QuickUser**: 1-3 second wait time between actions (rapid gameplay)
- **NormalUser**: 3-7 second wait time (typical gameplay)
- **SlowUser**: 5-15 second wait time (casual gameplay)

6.3.3 Implementation Details

The Locust test implementation (`docs/locustfile.py`) features:

- **Sequential Task Execution**: `GameUserFlow` class orchestrates the complete workflow
- **Session Management**: Each user maintains state across requests:
  - Username with random suffix (e.g., `loadtest_user_12345`)
  - JWT token for authenticated requests
  - Active game ID during matches
  - Selected deck slot
- **SSL Configuration**: Disables certificate verification for self-signed certificates
- **Error Handling**: Graceful handling of concurrent access scenarios:
  - 400 responses during registration marked as success, duplicate usernames expected and should not impact on the test results
  - 401 responses during opponent's turn marked as success, they indicate it's not the user's turn and should not impact on the test results
  - Failed operations are logged but don't interrupt test flow

6.3.4 Test Execution

Setup:

```
# Install Locust
pip install locust

# Ensure all services are running
cd src
docker compose up -d
```

Running Tests:

```
# Start Locust web interface
cd docs
locust

# Access web UI at http://localhost:8089
```

Configuration:

1. Set number of users (e.g., 50 concurrent users)
2. Set spawn rate (e.g., 4 users/second)
3. Set host: `https://localhost:8443`
4. Click "Start" to begin test

6.3.5 Metrics and Analysis

Locust provides real-time metrics during test execution:

- **Request Statistics**:
  - Requests per second (RPS) by endpoint
  - Response time percentiles (50th, 95th, 99th)
  - Failure rates and error types
  - Average response sizes
- **Endpoints**:
  - `/users/register`: user creation
  - `/users/login`: authentication
  - `/collection/decks`: deck management operations
  - `/game/match/join`: matchmaking queue
  - `/game/hand`: cards in hand
  - `/game/play`: card play action
  - `/history/matches`: user's matches history
  - `/history/leaderboard`: leaderboard

7 Security

7.1 Security – Data

7.1.1 Input Sanitization

Selected Input: User email addresses during registration and modification.

Description: The email input is a critical user identifier used for account recovery and communication. It must be sanitized to prevent:

- NoSQL injection attacks
- Cross-site scripting (XSS) attempts
- Duplicate account creation

Microservices Involved:

- **User-Manager:** Handles registration and email validation
- **User-Editor:** Manages email modification requests
- **API Gateway:** Performs initial validation before forwarding requests

Sanitization Strategy: **Type Validation:**

```
class UserCreate(UserBase):
    password: str = Field(..., min_length=3)
    email: str = Field(..., description="User's email")
```

Hash-based Duplicate Prevention:

```
def hash_search_key(data: str) -> str:
    """SHA-256 hash for consistent, secure searching"""
    return hashlib.sha256(data.lower().encode('utf-8')).hexdigest()

# Usage in registration
hashed_email = hash_search_key(user_in.email)
if USERS_COLLECTION.find_one({"hashed_email": hashed_email}):
    raise HTTPException(status_code=400,
                        detail="Email already registered")
```

Query Parameter Validation:

```
# Explicit type enforcement prevents injection
page = request.args.get('page', default=0, type=int)
```

7.1.2 Data Encryption at Rest

Encrypted Data: To protect confidentiality in case of database compromise email addresses and usernames in user-db have been encrypted at rest.

Encryption Implementation: **Encryption Method:** Fernet
Key Management:

- Encryption key stored as Docker secret at `/run/secrets/user_db_encryption_secret_key`
- Generated using cryptographically secure random bytes (32 bytes/256 bits)
- Encoded in Base64 URL-safe format for compatibility

Encryption Location: `user-manager` microservice
Usage in Data Storage:

```
# main.py - User registration
user_data = {
    "username": user_in.username,
    "email": encrypt_data(user_in.email),  # Encrypted before storage
    "hashed_password": get_password_hash(user_in.password),
    "hashed_email": hash_search_key(user_in.email)
}
USERS_COLLECTION.insert_one(user_data)
```

Decryption Location: `user-manager` microservice (during token validation and user data retrieval)

```
# Automatic decryption when retrieving user data
user = UserInDB(
    username=user_doc['username'],
    email=decrypt_data(user_doc['email']),  # Decrypted on read
    hashed_password=user_doc['hashed_password']
)
```

Reasons:

- **Defense in Depth:** Even if the database is compromised, sensitive data remains encrypted
- **Dual Hashing Strategy:** Email addresses are both encrypted (for storage) and hashed (for duplicate checking), preventing inference attacks
- **Separation of Concerns:** Encryption keys are managed externally via Docker secrets

7.2 Security – Authentication and Authorization

7.2.1 Architecture: Centralized Authentication

Our system implements a **centralized authentication model** where the `user-manager` microservice acts as the single source of truth for authentication and token validation.

Token Validation Flow:

1. **Client Request:** User sends credentials to API Gateway
2. **Authentication:** API Gateway forwards to `user-manager`
3. **Token Generation:** `user-manager` creates JWT and returns to client
4. **Subsequent Requests:** Client includes JWT in Authorization header
5. **Token Validation:** Each microservice calls `user-manager` to validate token
6. **Authorization:** Microservice proceeds with request if token is valid

Token Validation Process:

Figure 1: Centralized token validation architecture

Aspect	**Implementation**
Signing Key	256-bit secret key (HS256 algorithm)
Storage Location	Docker secret: `/run/secrets/jwt_secret_key`
Access Control	Only `user-manager` container has read access
Generation	Cryptographically secure random bytes (Base64 URL-safe)
Rotation Strategy	Manual rotation via Docker secret update and container restart

Table 1: JWT signing key management

Key Management and Storage:

7.2.2 Access Token Payload Format

```
{
  "sub": "username",                 # Subject (username identifier)
  "username": "john_doe",            # Explicit username field
  "id": "507f1f77bcf86cd799439011",  # MongoDB ObjectId
  "exp": 1735220400                  # Expiration timestamp (Unix epoch)
}
```

Field Descriptions:

- `sub`: Standard JWT subject claim containing the username
- `username`: Redundant field for backward compatibility
- `id`: User's unique database identifier for efficient lookups
- `exp`: Expiration timestamp (30 minutes from issuance)

7.2.3 Token Generation Process

```
def create_access_token(data: dict):
    to_encode = data.copy()
    # Ensure 'sub' claim for OAuth2 compatibility
    if "sub" not in to_encode and "username" in to_encode:
        to_encode["sub"] = to_encode["username"]

    expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})

    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

7.2.4 Expired Token Handling

Strategy: Automatic rejection with explicit error messaging
Mechanism:

1. JWT library (`python-jose`) automatically validates `exp` claim during decoding
2. `JWTError` exception is raised if token is expired
3. Client receives HTTP 401 Unauthorized response
4. Client must re-authenticate to obtain new token

```
def get_current_user(token: str = Depends(oauth2_scheme)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Invalid credentials",
        headers={"Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
    except JWTError:  # Catches ExpiredSignatureError automatically
        raise credentials_exception

    user = get_user(username)
    if user is None:
        raise credentials_exception
    return user
```

Client-Side Handling:

- Client detects 401 response
- Redirects user to login page
- Clears stored token from local state
- User re-authenticates to receive fresh token

No Token Refresh Mechanism: For simplicity and security, we do not implement refresh tokens. Users must re-authenticate after 30 minutes of inactivity. This reduces the attack surface by limiting token lifetime.

7.3 Security – Analyses

7.3.1 Static Analysis with Bandit

Bandit was used to perform static application security testing on the Python codebase.

Command executed:

```
bandit -r src/
```

7.3.2 Dependency Vulnerability Scanning

Pip Audit:

```
pip-audit
```

Result: One low-severity vulnerability found in pip version 24.0, fixed upgrading it to 25.3.

Dependabot (GitHub): Enabled on the repository to automatically scan for vulnerable dependencies.

Result: No vulnerabilities detected in production dependencies.

7.3.3 Container Image Vulnerability Analysis

Docker Scout: **Usage:** all container images where scanend for vulnerabilities using Docker Scout directly from Docker Desktop.

Mitigation Actions:

- **RabbitMQ:** 2 critical, 9 high-severity vulnerabilities because of the golang and stdlib versions in the linux image used by 3-management. Solved upgrading to `rabbitmq:4.2.1-alpine`.
- **MongoDB:** This was more of an issue,
  - The mongo:latest image we used in development had 6 high-severity vulnerabilities.
  - All the versions of the official mongo image have either a lot of vulnerabilities or run only on windows servers (windows OS images).
  - "latest" images are not an option as a final shipment version, this would mean the images possibly breaking for an update.
  - All the companies we've found providing mongo linux images moved to a paid model, providing only "latest" versions for free for development environments.
  - The only option we had were old legacy images made by those companies in the past. Fortunately we found *circleci/mongo:4.0-xenial-ram*, a 4 years old image that relied on a really light-weight ubuntu version with 0 vulnerabilities.
  - This is definitely not a future-proof solution. The best path forward would probably be to build a custom image where we install mongoDB on top of a lightweight alpine or debian image.
- **JWT:** The User Manager microservice used an abandoned library to handle the JWT called python-jose with a high severity vulnerability. Fortunately there was a new and updated library that had the same functions, called PyJWT.

Trivy: Command executed:

```
trivy image --severity HIGH,CRITICAL <image_name>
```

Findings:

- **Exposed Secrets:** Trivy detected hardcoded keys in repository
- **Academic Context:** As agreed with the Professor, keys are intentionally included since it's a project with educational purposes. In a real context, we wouldn't have published neither keys on the GitHub repository
- **Production Mitigation:** In a real deployment, keys would be:
  - Generated dynamically per environment
  - Stored in secure secret management systems
  - Never committed to version control

7.4 Security – Threat Model

8 Use of Generative AI

During the project development we've made extensive use of various AI models, experimenting with them and learning how to use those tools as efficiently as possible. We've mainly taken advantage of what was given to us for free as students: GitHub Copilot Pro and Gemini Pro. We've used them in the following two areas.

- Researching informations and tools: this has gotten a lot better in the past years, with models combining their knowledge with informations found on the web, providing extensive explanations and sources, while almost never allucinating. Researching using LLMs has definitely almost replaced our reliance on classical search engines.
- Writing Code. To which we've made the following considerations.
  - The non-premium AI models of Copilot Pro (GPT-4.1, GPT-4o, GPT-5 mini, Grok Code Fast 1) were pretty bad and mostly generated bad-quality results we lost time in reviewing and then discarding. This was true even in writing something as simple as latex code, e.g. GPT-4.1 had a hard time understanding what an hyperref was.
  - We tried various AI coding VSCode extensions: Github Copilot, Roo Code, Gemini Code Assist etc. We found the Copilot extension as the most robust implementation of agentic behavior for vscode, and, together with the free premium requests, we ended up using it almost exclusively.
  - We found Claude Sonnet 4.5 as the best premium model for one-shot code generation, it has been useful to generate some initial drafts for the microservices and to re-generate whole functions.
  - We found the Copilot Gemini 3.0 Agent behaving in a more *human* way, trying to get the best result while doing as least work as possible. E.g. for big changes in a single file that meant almost a whole refactoring, it often tried to delete the whole file and start from scratch.
  - The inline suggestions were a dividing topic, while some of us found them useful and kept them on, some others found them distracting and fatigue-inducing, where they suggested mostly wrong code while constantly drawing the attention of the user.

  - Even if ChatGPT models are still the most popular ones, we found them working on par, if not worse, than the others, hence we made less extensive use of them. This resonates with the recent "Code Red" raised by OpenAI after the release of Gemini 3.0.

8.1 Final Remarks

We found experienced and smart human beings better in any way but speed compared to the most powerful AI models that are publicly available. At the moment LLMs can be a useful tool for those people to work faster, but definitely not as a replacement of them.
Reasoning models behaved way better, showing the robustness of more rigid and iterative workflows, applied to the more efficient and smaller MoE (Mixture of Experts) models, compared to one-shot larger models. This resonates with what is shown in the course for cloud services, where decomposing complex tasks (monoliths) into smaller, more easily verifiable tasks (microservices), often yields better results.

first time doing this, happened that we accepted meaningless changes (that lost time of the others for reviewing code and sometimes lead to problems or reverting changes), tried vibe-coding out of problems when tired,

9 Additional Features

9.1 Green User Stories

9.2 Cloud Storage of Decks

To allow players to keep their decks stored in the cloud, we added

9.3 Client

9.4 Endpoint-based Service Interaction Smell - Proof of Concept

The project requirements asked to make up for the **Wobbly service interaction** smell. We solved it mainly with timeouts as suggested.
Regarding the **Endpoint-based service interaction** smell, the requirements said to ignore it since it would've been difficult to solve.
We decided to partially solve it anyway in a single interaction between the **Game History** and **Game Engine** microservices as a proof-of-concept.
In this interaction we replaced the timeout with RabbitMQ, a Message Broker, that solved both the **Wobbly service interaction** and the **Endpoint-based service interaction** smells. This meant a new docker container for RrabbitMQ and running code to fill up the RabbitMQ queue in Game Engine and to ping it for new data in Game History.

9.5 Test-match

10 Build and Run Instructions

This project is a multi-service card game platform designed for two players. It features authentication, deck building, a match-simulation engine, and history tracking. All services are containerized and orchestrated via Docker Compose.

10.1 Prerequisites

Before starting, ensure your environment meets the following requirements:

- **Docker & Docker Compose** (Required for orchestration).
- **Python 3.10+** (Optional, only required for local non-containerized development).

10.2 Quick Start Guide

1. **Clone the repository**

```
git clone https://github.com/ashleyspagnoli/ASE_project.git
cd ASE_project/src
```

2. **Build and launch services** Run the following command to build the images and start the containers:

```
docker compose up --build
```

3. **Verify Service Status** Once the containers are running, the architecture exposes the following endpoints:

Service Name	Responsibility	Local URL
User Manager	Authentication & JWT	`https://localhost:5004`
Collection	Deck Management	`http://localhost:5003`
Game Engine	Core Logic & Matchmaking	`http://localhost:5001`
Game History	Match Logging	`http://localhost:5002`

10.3 Development & Testing

10.3.1 Environment Configuration

Each microservice is configured via environment variables. For specific configuration keys, refer to the `Dockerfile` and `requirements.txt` located in each service's directory.

10.3.2 Testing the Workflow

A Postman collection is provided for end-to-end testing. Import `game_workflow.postman_collection.json` into Postman to simulate a full lifecycle:

- User Registration and Login (Token generation).
- Deck creation and validation.
- Matchmaking and gameplay simulation.

10.3.3 Key API Endpoints

Authentication

- `/users/register`: user registration [POST]
- `/users/login`: user login [POST]
- `/users/validate-token`: internal JWT token validation [GET]

Deck Management

- `/collection/cards`: get the collection of cards [GET]
- `/collection/decks`: create a new deck [POST]

Game Engine

- `/game/connect`: connect to start playing the game [POST]
- `/game/matchmake`: manual request to start the match [POST] (TO REMOVE)
- `/game/play/{game_id}`: play a card [POST]
- `/game/state/{game_id}`: get the state of the game [GET]

Game History

- `/leaderboard`: get the whole leaderboard [GET]
- `/matches`: get the history of your played matches [GET]
- `/addmatch`: memorize a new match [POST]

10.4 Maintenance

To stop the application and remove containers/networks, run:

```
docker compose down
```