ASE Project Report - GROUP 2

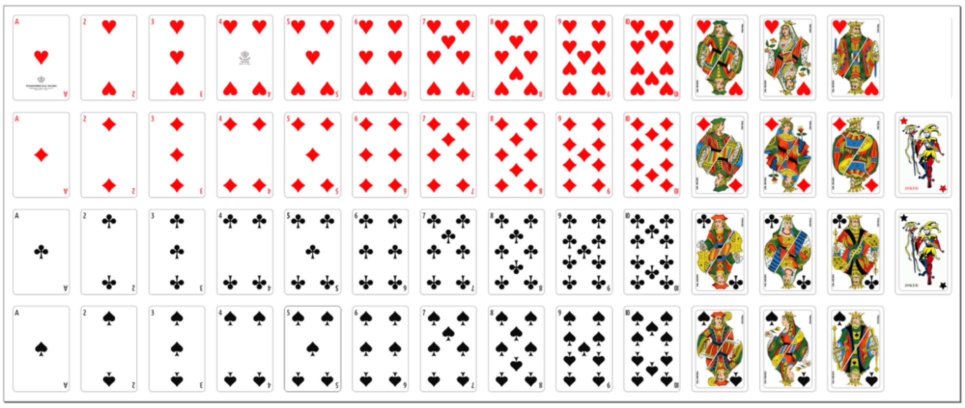
Advanced Software Engineering

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

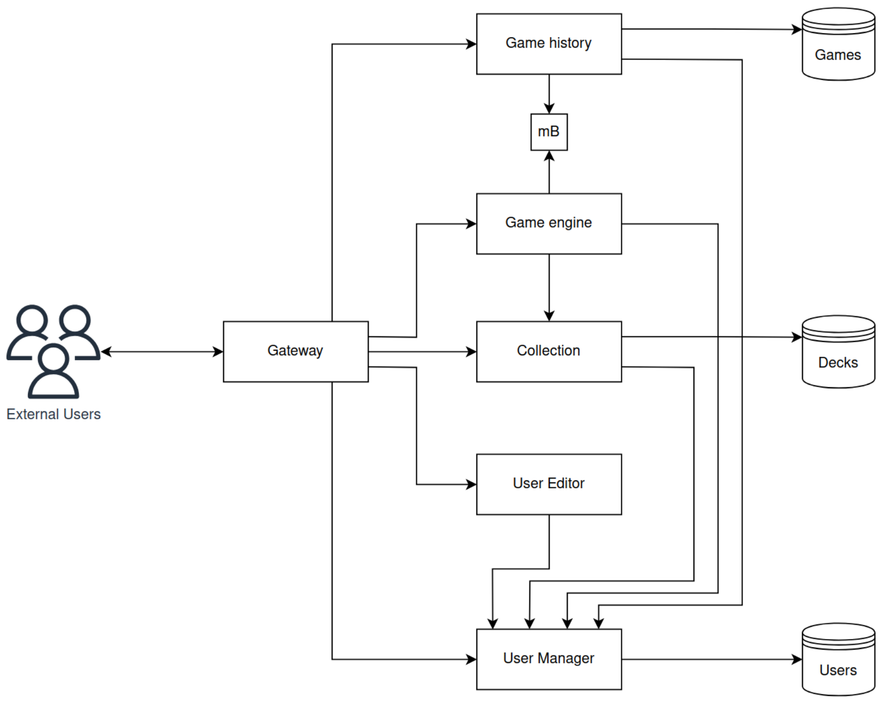
As cards we used the classical 52 + Joker deck.
As anyone could expect, the two features assigned to those cards are the number and the suit, while the Joker is a special card with no features, that beats any other card.



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:

   ♥ > ♦ > ♣ > ♠

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 API Calls of the Game Flow

All the following requests require the `Authorization` header containing the JWT token of the authenticated user:

```
Authorization: Bearer <your_jwt_token>
```

5.1 Deck Building Phase

We've separated the deck building from the game (More information in Cloud Storage of Decks)

- **GET /collection/cards**: Retrieve all available cards.
- **POST /collection/decks**: Create a new deck.

```
Body:
{
  "deckSlot": 1,
  "deckName": "My Deck",
  "cards": ["hA", "h5", "d5", "dK", "c7", "c8", "sA", "s5"]
}
```

- **GET /collection/decks**: Retrieve the user's decks.

5.2 Game Phase

- **POST /game/match/join**: Join the matchmaking queue.
- **GET /game/match/status**: Check the status of the matchmaking process.
- **POST /game/deck/{game_id}**: Submit the chosen deck for the game.

```
Body:
{
  "deck_slot": 1
}
```

- **GET /game/hand/{game_id}**: Retrieve the player's current hand.
- **POST /game/play/{game_id}**: Execute a play (e.g., play a card).

```
Body:
{
  "card": {
    "value": "K",
    "suit": "clubs"
  }
}
```

- **GET /game/state/{game_id}**: Retrieve the current state of the game.

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

All tests are located in `/docs/tests` as requested.

Unit tests are executed using dedicated Dockerfile_test files like we've seen in the lectures. We made the following decisions for mocking.

- For all microservices the databases are mocked using the mongomock library, installed in the Dockerfile_test file.
- To mock the RabbitMQ for Game History we disabled the loop pinging the RabbitMQ microservice. This meant that we needed custom endpoints to add matches and users to the DB in the testing environment. `/addmatches` and `/addusernames`.
- In Game History we also needed to mock the function to get the usernames by ids of User Manager.

Unit tests execution:

- **Collection**:

```
docker build -f collection/Dockerfile_test -t collection-test .
docker run -p 5000:5000 collection-test
newman run docs/tests/collection_ut.postman_collection.json --insecure
```

- **Game History**:

```
docker build -f game_history/Dockerfile_test -t history-test .
docker run -p 5000:5000 history-test
newman run docs/tests/game_history_ut.postman_collection.json --insecure
```

- **User Manager**:

```
docker build -f user-manager/Dockerfile_test -t user-manager-test .
docker run -p 5004:5000 user-manager-test
newman run docs/tests/user_manager_ut.postman_collection.json --insecure
```

6.2 Integration Testing

- **IT-001: Complete Game Workflow - Happy Path**: Tests end-to-end user journey from registration to game completion.
- **IT-002: Authentication & Authorization**: Verifies login, token validation, and access control enforcement.
- **IT-003: Deck Validation**: Ensures deck building rules and constraints are correctly enforced.
- **IT-004: Game History & Leaderboard**: Checks match history retrieval and leaderboard accuracy.
- **IT-005: Cross-Service Data Consistency**: Confirms data isolation and consistency across microservices.
- **IT-007: Error Handling & Edge Cases**: Validates error responses and handling of invalid or edge-case inputs.

Integration tests execution:

- **Docker Compose**:

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

# Run integration tests (from project root)
newman run docs/tests/integration.postman_collection.json --insecure
```

Those tests can also

!

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 password+salt
    "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 (`pyJWT`) 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.

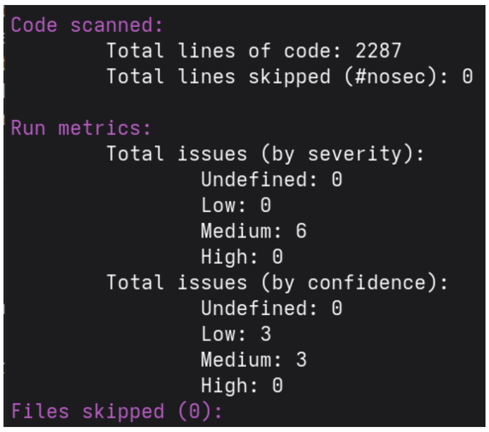
7.3 Security – Analyses

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/
```



```
Code scanned:
        Total lines of code: 2287
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 0
                Medium: 6
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 3
                Medium: 3
                High: 0
Files skipped (0):
```

7.3.2 Dependency Vulnerability Scanning

Pip Audit: Executed as a github action. It found vulnerabilities before but not after merging the Dependabot pull request.

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

- Updates python-multipart from 0.0.9 to 0.0.18
- Updates requests from 2.31.0 to 2.32.4
- Updates cryptography from 42.0.7 to 44.0.1

This also solved issues raised by pip-audit

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.
- FastAPI 0.111.0 has critical vulnerabilities caused by a starlette dependency. Solved by upgrading to 0.124.4.
- python:3.12.3-slim has critical vulnerabilities caused by the debian image used. Solved by upgrading to 3.12.12-slim.

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, neither the databases passwords in the docker compose file, but we would've passed passwords as environment variables and used certificates provided by CAs stored locally and added to .gitignore.
- **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

7.4.1 Architecture Overview

The application follows a microservices architecture with a centralized API Gateway acting as the single entry point. All inter-service communication is secured via TLS.

- **External Entry Point:** API Gateway (exposed on port 8443).
- **Internal Services:** User Manager, User Editor, Game Engine, Game History, Collection.
- **Data Stores:** Dedicated MongoDB instances for Users, History, and Decks.
- **Messaging:** RabbitMQ for asynchronous communication, configured to reject non-TLS connections.

7.4.2 Assets Identification

- **User Credentials:** Passwords (hashed) and Emails (encrypted).
- **Session Tokens:** JWTs used for authentication.
- **Game Data:** Match history and player moves.
- **Infrastructure Secrets:** TLS certificates and private keys managed via Docker Secrets.

7.4.3 STRIDE Analysis

Spoofing

- **Threat:** An attacker impersonating a legitimate user.
- **Mitigation:** Strong authentication via `user-manager`, JWT validation on every request, and TLS for all connections to prevent credential interception.
- **Threat:** A rogue service impersonating a legitimate microservice.
- **Mitigation:** Internal TLS usage with certificates managed via Docker Secrets ensures encrypted communication channels.

Tampering

- **Threat:** Modification of game results or user data in transit.
- **Mitigation:** End-to-end TLS encryption prevents Man-in-the-Middle (MitM) attacks.
- **Threat:** Modification of data at rest.
- **Mitigation:** Critical user data (emails) is encrypted at rest. Database access is restricted to specific microservices.

Repudiation

- **Threat:** A user denying they performed an action (e.g., a game move).
- **Mitigation:** The `game_history` service logs all match outcomes. Actions are authenticated via JWT, linking them irrefutably to a user ID.

Information Disclosure

- **Threat:** Leaking sensitive user information.
- **Mitigation:** Data minimization (only necessary data returned), encryption of sensitive fields in DB, and strict API Gateway routing preventing direct access to backend services.

Denial of Service (DoS)

- **Threat:** Overwhelming the system with requests.
- **Mitigation:** The API Gateway acts as a buffer. RabbitMQ decouples heavy processing (like history logging) from the critical path, preventing cascading failures.

Elevation of Privilege

- **Threat:** A regular user accessing administrative functions.
- **Mitigation:** Centralized authorization logic in `user-manager`. The API Gateway enforces route restrictions.

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. Given the recent advancments in AI, this was actually the first time we found it being reliable enough to use it as a coding assistant.
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.
  - We sometimes tried to "vibe-code ourselves out of errors" with variable results. Sometimes we accepted changes that seemed ok / useless to us but that broke the code for others and, anyway, lost their time for reviewing it.

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 work-flows, 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.

9 Additional Features

9.1 Green User Stories

The project implements two green user stories that enhance the player's gameplay experience by providing essential game state information.

9.1.1 Viewing Cards in Hand

The first green user story addresses the player's need to see their available cards: *"I want to see the cards in my hand so that I know which cards I have to play"*.

The **Game Engine** implements the `GET /hand/{game_id}` endpoint that returns the authenticated player's current hand as a JSON array of card objects. Each card object contains the `value` and `suit` properties (e.g., `{"value": "K", "suit": "hearts"}`). This allows the client application to display exactly which cards the player currently holds and can play during their turn.

The endpoint is protected by JWT authentication, ensuring that each player can only view their own cards and not their opponent's hand, maintaining game fairness and competitive integrity.

9.1.2 Turn State Tracking

The second green user story addresses the need for players to know when they can act: *"I want to know who's turn it is so that I know if I can play or not"*. The Game Engine implements turn state tracking through the `current_round` mechanism.

During each round, the game tracks which players have submitted their cards. The `GET /state/{game_id}` endpoint can be polled by clients to determine the current turn state by examining:

- **Empty current round**: Both players can play (new round starting)
- **One card played**: The opponent is waiting for the current player to submit their card
- **Both cards played**: The round is being resolved, and players will draw new cards

Additionally, when a player submits a card via `POST /play/{game_id}`, the response includes a `status` field:

- `"waiting"`: The player has submitted their card and is now waiting for the opponent
- `"resolved"`: Both players submitted cards, the round is complete, and results are provided
- `"finished"`: The match has ended with a winner

This mechanism allows players to understand the flow of the game and know when they can take action.

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