Functional Requirements for the ASE Lab 2025/26 Project

Card Game Backend

1) Introduction

This document defines the project's functional requirements and may be updated if unforeseen situations arise. Most items refer to the lab activities; course's slides are additional sources for the project realisation.

The service was introduced in Lab 2 ("User Stories") and the project guidelines were outlined in Lab 3 ("Project Kickoff"). Some requirements relate to topics that will be explained later in the course. Refer to the "Project Take away" slide in each lab for direct links to the relevant lab topics and project details.

The final software must implement all red and yellow user stories for players listed in *US.xlsx*. Implementing additional user stories (even those not listed in the file) counts as an additional feature. Green user stories (or any additional features) are optional but will be rewarded with a higher evaluation if implemented and documented effectively. Achieving the maximum grade (30 cum laude) is possible by implementing only the red and yellow user stories. Following best practices for every tool you use is highly recommended.

2) Architecture & Repository Requirements

The backend must follow a microservices architecture and be deployable with Docker Compose. Running the following must build and start the entire system:

```
docker compose up
```

Each microservice must expose a REST API, communicate over HTTP(S), and persist its state in a data store.
The code must be hosted on GitHub. The repository must include a README.md with a standard Get Started section containing commands to install, build, run the backend, and run the main tests.
When declaring external dependencies, specify the exact version. For example, in Python, each requirements.txt must pin versions (e.g., Flask==3.0.3).
Implementing a client for interacting with the backend is not mandatory, but it can be useful to test end-to-end flows (e.g., the registration and login flow). You can use Postman as a client by exploiting its advanced features.

3) Documentation (in the docs/ folder of the repo)

Include at least the following:

- Final report in PDF format (based on the template that will be published).
- OpenAPI specification file for the REST API exposed by the backend.
- Postman collections (JSON files) with unit tests for at least three internal (no API gateway or DB manager) microservices. Use a subfolder if there are many files.
- Locust file(s) for performance tests.
- A copy of the YAML file(s) specifying GitHub Actions workflow(s).

The microservice architecture must be free of architectural smells, with the exception of the "Endpoint-based Interaction" smell, as discussed in the class on "Smells and Refactoring for Microservices".

For each core functionality, provide a step-by-step guide (i.e., the sequence of API calls) showing how to use it. Each additional feature must be explained in the report in terms of motivation, realisation, and usage.

4) Core Functionalities

Even though a client is not required, design the backend with client usage in mind. The main business logic should reside in the backend, but you may assume some logic in the client. For example, the image and the details of a single card may be retrieved by two separate requests, while a client can display them together. Any assumptions about client-side logic must be documented in the final report.

The main core functionalities are:

- Registration and login: each player can register once and then log in to start playing. (See Security for details.)
- Cards view: each player can view the complete set of cards before playing.
- Match history: each player can see all their previous matches, including the sequence of moves and results.
- Play a match: each player can play a match against another player. At least two players must be available concurrently. The rule set and all aspects of match behaviour are free to design, provided they respect the constraints presented in Lab 3. You must define and document the complete match lifecycle, state transitions, termination conditions, and the handling of contingencies.

Important: Communication between microservices must not rely on the client. For example, match history should be updated by the microservice(s) managing live matches, not by a client request that posts the match result.

In general, consider and mitigate potential fraudulent behaviours by players and avoid actions that could lead to inconsistent state or game balance.

Logging every event is not mandatory. However, for debugging, you are encouraged to implement appropriate logging.

5) Testing

Unit tests (i.e., testing a microservice in isolation) are required for at least three internal (not the API gateway or any DB manager) microservice. For each endpoint, provide at least:

- one test for valid input (expecting 200 OK), and
- one test for invalid input (expecting an error response).

These unit tests must be conducted using Postman, and the collection should be exported as a JSON file. The same requests can be reused as integration tests against the gateway with the full architecture running.

Performance testing must be conducted using Locust, targeting only the gateway. The performance test should cover the core functionalities.

6) Security

- All containers must run in user mode to prevent privilege escalation on the host.
- All communication must use HTTPS, with each microservice utilising self-signed certificates.
- Inputs received by each microservice must be sanitised, and sensitive data must be encrypted in the database(s).
- Authentication and authorisation must comply with OAuth2, using JWT tokens. Authentication uses username and password credentials, which must be transmitted, stored, and validated securely. Follow the specifications indicated in the Access Security lab.
- The codebase must undergo automatic static and dependency analysis using free tools available for the language(s) in use. Docker images must be free from critical and high-severity vulnerabilities. Follow the specifications indicated in the Security Analysis lab.

7) Evaluation

Each group member is responsible for the entire project: they must understand the backend’s design, deployment, execution, testing, and the rationale behind the realisation. This does not require knowledge of every low-level implementation detail.

Evaluation criteria:

- Correctness & operability: we must be able to install and execute the architecture and perform the core functionalities by following the documentation.
- Documentation quality: compliance with specifications, clarity, usefulness, and appropriate length (not too much, not too little).
- Design choices: there is a high degree of freedom; explain your decisions to show they are reasonable and well-justified.
- Testing: tests should sufficiently cover the main expected behaviours of the backend.
- Security: all specified security requirements must be implemented. We will attempt attacks to verify the implementation.
- Additional features (optional): reasonable, well-implemented, and well-documented features can earn up to 3 extra points.

8) Miscellaneous

In general, any solution that complies with these functional requirements is acceptable. If you are unsure about any aspect of your solution, send an email to alessandro.bocci@unipi.it to raise your concern or to arrange an office-hour meeting.