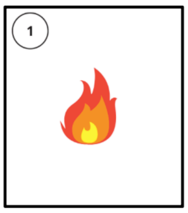
ZM2 - Documentation

Card Overview

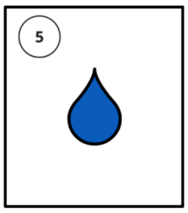
Each card has two main characteristics:

1. **Element** — defines the card's type (Fire, Water, Earth, Air, Spirit)
2. **Power Level** — a numerical value from 1 to 5

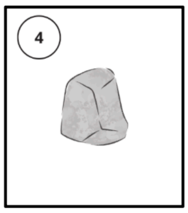
Elements (Card Types):



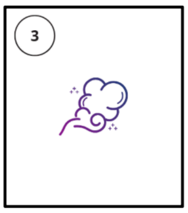
Fire



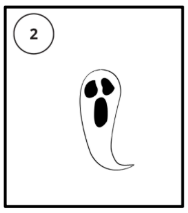
Water



Earth



Air



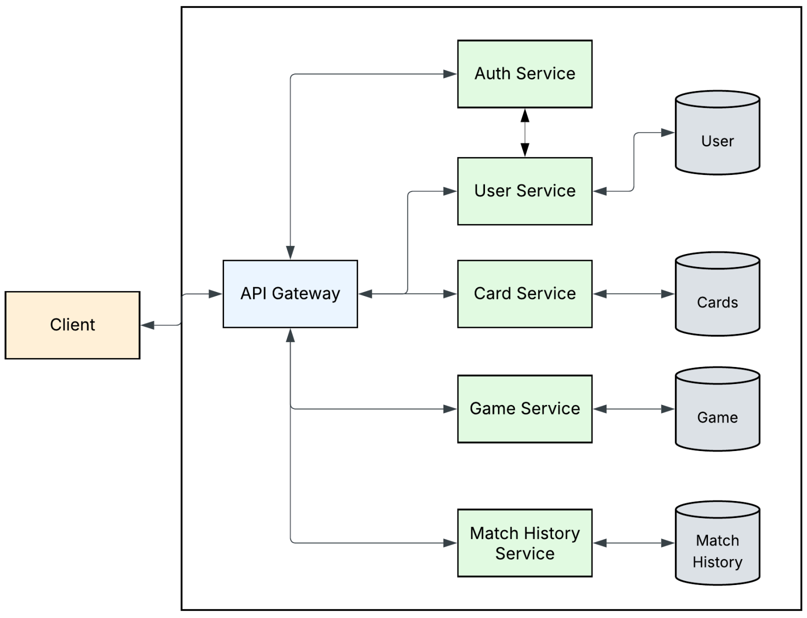
Spirit

Name	Element	Power Level
Inferno Dragon	FIRE	5
Blazing Phoenix	FIRE	4
Flame Sprite	FIRE	3
Ember Wolf	FIRE	2
Spark Wisp	FIRE	1
Tidal Leviathan	WATER	5
Ocean Guardian	WATER	4
River Serpent	WATER	3
Pond Nymph	WATER	2
Droplet Elemental	WATER	1
Mountain Titan	EARTH	5
Stone Golem	EARTH	4
Rock Troll	EARTH	3

Name	Element	Power Level
Cave Dweller	EARTH	2
Pebble Sprite	EARTH	1
Storm Wyrm	AIR	5
Wind Djinn	AIR	4
Sky Hawk	AIR	3
Breeze Fairy	AIR	2
Zephyr Wisp	AIR	1
Ethereal Sage	SPIRIT	5
Ghost Knight	SPIRIT	4
Phantom Mage	SPIRIT	3
Soul Wisp	SPIRIT	2
Spirit Spark	SPIRIT	1

Architecture

The architecture is structured around a set of core domain entities that represent users, cards, games, and match histories. Each component is responsible for managing a specific part of the gameplay lifecycle. The diagram illustrates how real-time games are handled, how cards are assigned and played, and how results are stored for long-term statistics. It also highlights the flow between active game sessions and archived match history, ensuring both live interaction and persistent data storage.



User records
- id
- username
- password (hashed)
- email
- totalWins
- totalLosses
- totalScore
- createdAt

Cards records
- id
- name
- element (FIRE, WATER, EARTH, AIR, SPIRIT)
- power (1-5)
- imageUrl
- description

Game records
- id
- player1Id
- player2Id
- status (WAITING, IN_PROGRESS, FINISHED)
- currentTurn
- player1Score
- player2Score
- currentPlayerTurn
- startedAt
- finishedAt

Match History Records
- id
- gameId
- player1Id
- player2Id
- winnerId
- finalScoreP1
- finalScoreP2
- totalTurns
- finishedAt

User Service: Account Management, Profile CRUD operations, Player Statistics

Auth Service: Token Generation, User Registration & Login (Calls the User Service)

Card Service: Card Collection, Card Details, Element & Power Data

Game Service: Live Matches, Game Logic, Matchmaking, WebSocket (Real-time)

Match History Service: Match Records, Player History, Leaderboards.

- Auth Service communicates with User Service to register or login a user. (User Service has these functions exposed)
- GameService communicates with Match History Service: It sends a request to save a match.
- GameService communicates with User Service: It sends requests to update a user stats.
- Match History Service communicates with User Service: It sends requests to fetch users.

User Stories

Priority is set from most important first, to least important last with the user stories written in red already implemented. Red, yellow & green features are shown by text color.

Priority	User Story
1	As a player, I want to log in to the game so that I can play a match.
2	As a player, I want to create an account so that I can participate in the game.
3	As a player, I want to be safe about my account data so that nobody can steal or modify them.
4	As a player, I want to start a new game so that I can play.
5	As a player, I want to select a subset of cards so that I can start a match.
6	As a player, I want to select a card so that I can play my turn.
7	As a player, I want to know the score so that I know if I am winning or losing.
8	As a player, I want to know who won the turn so that I can see the updated score.
9	As a player, I want to know who won the match so that I can see the final result.
10	As a player, I want to ensure that the rules are not violated so that I can play a fair match.
11	As a player, I want to check or modify my profile so that I can update my information.
12	As a player, I want to view the overall card collection so that I can plan a strategy.
13	As a player, I want to view the details of a card so that I can plan my strategy.
14	As a player, I want to know the turns so that I can see how many rounds remain.
15	As a player, I want to view the list of my old matches so that I can review my performance.
16	As a player, I want to view the details of one of my matches so that I can analyze how I played.
17	As a player, I want to view the leaderboards so that I can see who the best players are.
18	As a player, I want to prevent people from tampering with my old matches so that I always have them available.

Priority	User Story
19	**As a player, I want to see who turn it is so that I know if I can play or not**
20	**As a player, I want to be able to see the cards in my hand so that I know that card I can play next**

Game Rules

Elemental Hierarchy:

> Fire 🔥 < Water 💧 < Earth 🌍 < Air 🌪️ < Spirit ✨ < Fire 🔥

- Fire beats Spirit
- Spirit beats Air
- Air beats Earth
- Earth beats Water
- Water beats Fire

If two cards have the same element, their **Power Level** determines the winner.

Power Level and Order

> 1 < 2 < 3 < 4 < 5

If both element and power level are equal, the round is considered a **draw**.

Game Flow

The game implements a **real-time multiplayer card battle** where two players compete using a deck of cards with elemental and numerical attributes.

The logic defines how player data, cards, sessions, and scores are managed, as well as how game outcomes are calculated.

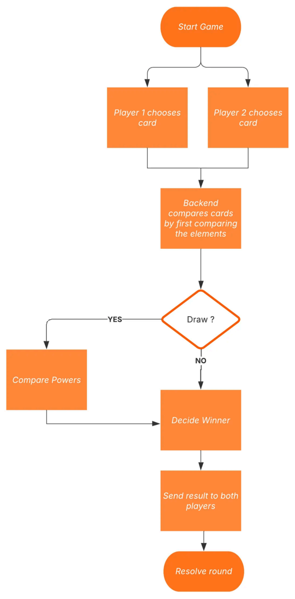
All gameplay logic is handled on the backend to ensure fairness and prevent manipulation from the client side.

Player Information: Each player is represented by a user profile containing both gameplay and account information.

1. *Game Setup*
    - Two players join a room through matchmaking.

- Each player receives a predefined set of cards from the backend.
- A WebSocket connection is established for real-time interaction.

2. *Round Play*
    - Both players select a card.
    - The backend compares both cards according to the *Elemental Hierarchy*.
    - If elements are equal, the comparison falls back to *Power Level*.
    - The winner gains points, and both played cards are discarded.
    - The backend sends round results and updated scores to both players.
3. *End of Match*
    - The game continues until all cards are played.
    - The player with the highest score wins the match.
    - The result is stored in the database, updating the player's match history and overall score.



Overall Score Logic

Each player's **Overall Score** is calculated based on the number of rounds (turns) won. When a user win a round we add 1 to its score.

Testing

1. Unit Testing in Isolation (Postman Collections)

Important !

If you are using MacBook with an Apple Silicon chip (M1, M2, or M3), problem of a **compatibility issue between the Docker image and your Mac's processor could happen.**

The error `no match for platform in manifest` means that the specific image tag you are trying to use (`eclipse-temurin:17-jre-alpine`) has not been built for the ARM64 architecture that your Mac uses.

Here is the fix: Open your `Dockerfile.test-isolation` file.

From: `FROM eclipse-temurin:17-jre-alpine`

TO: `FROM eclipse-temurin:17-jre`

Each microservice is tested independently, without involving the API Gateway or other services. For each microservice, a dedicated **Postman collection** is created and executed against the service running in isolation. We mock some logics that replace the actual real behavior when the app is runned in `“test-isolation”` environment.

Test Types:

- Successful request scenarios (200 OK)
- Error scenarios (e.g., invalid ID, missing payload, invalid state)

Microservices Covered:

- Auth Service
- User Service
- Card Service
- Game Service
- Match History Service

All the Postman collections (JSON file) are located in the `/docs` folder of the project.

To test. We have two options:

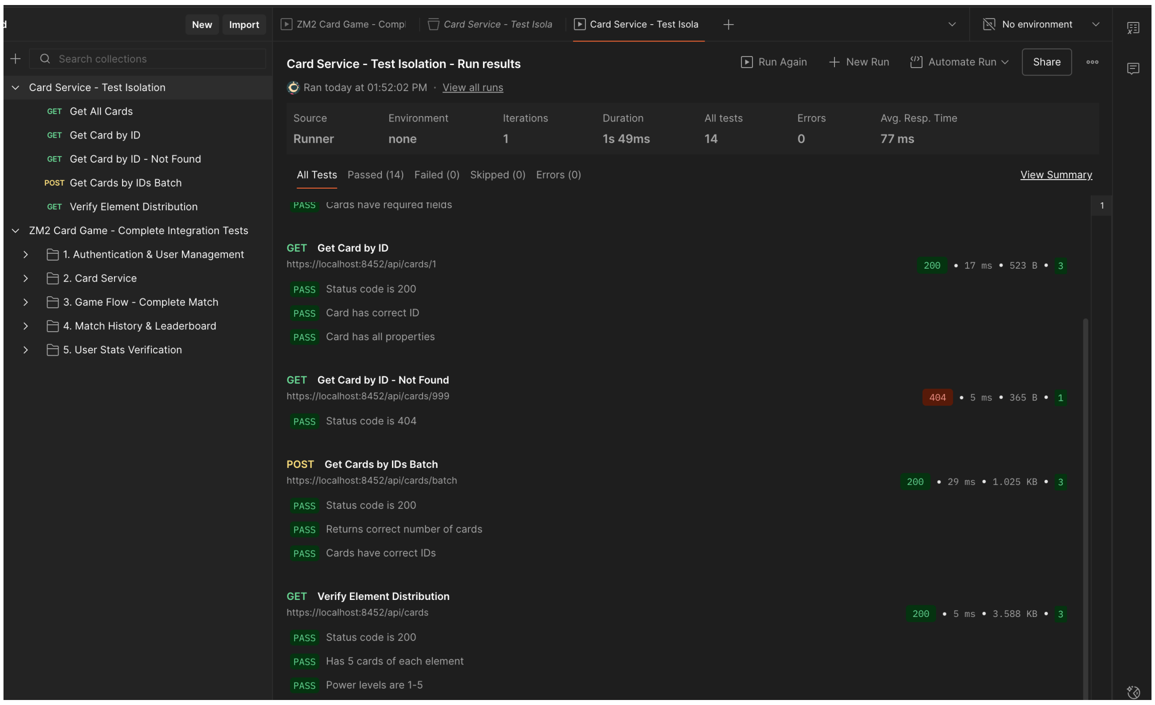
Build and run the dockerfile of each microservice:

1. Card Service

a. `docker build -f card-service/Dockerfile.test-isolation -t card-service-test .`

b. `docker run --name card-service-test -p 8452:8452 card-service-test`

Card Service Isolation test screenshot:


New
Import
ZM2 Card Game - Compl
Card Service - Test Isola
Card Service - Test Isola
No environment
Search collections
Card Service - Test Isolation
GET Get All Cards
GET Get Card by ID
GET Get Card by ID - Not Found
POST Get Cards by IDs Batch
GET Verify Element Distribution
ZM2 Card Game - Complete Integration Tests
1. Authentication & User Management
2. Card Service
3. Game Flow - Complete Match
4. Match History & Leaderboard
5. User Stats Verification
Run Again
New Run
Automate Run
Share
All Tests
Passed (14)
Failed (0)
Skipped (0)
Errors (0)
View Summary

Card Service - Test Isolation - Run results

Ran today at 01:52:02 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Errors	Avg. Resp. Time
Runner	none	1	1s 49ms	14	0	77 ms

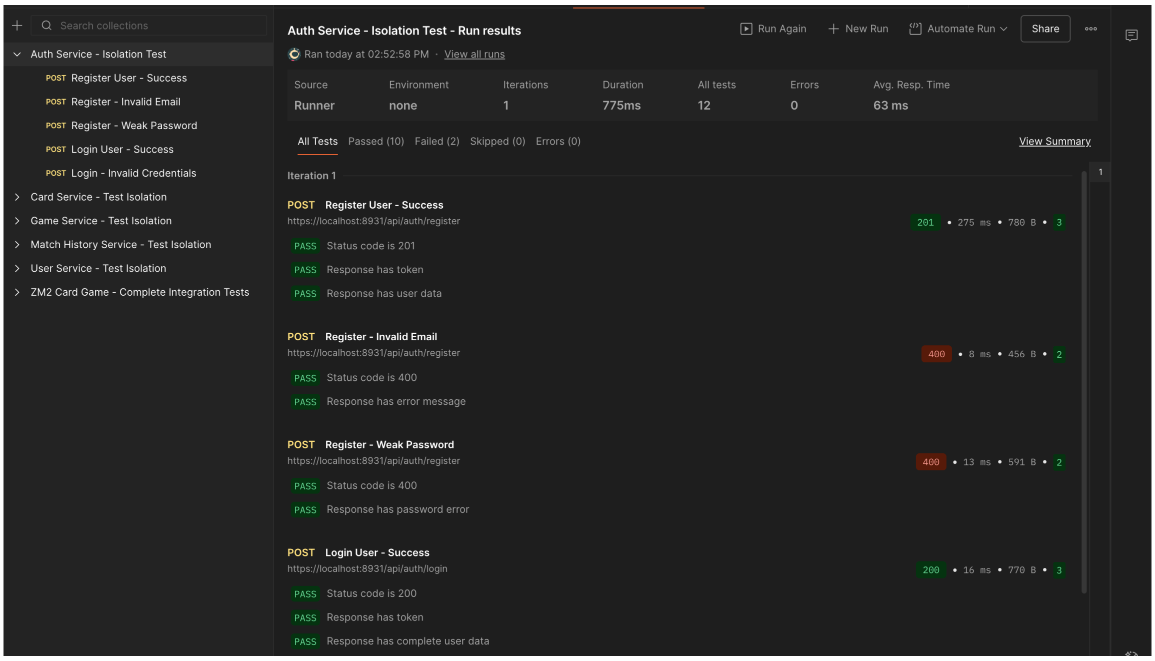
```
PASS  Cards have required fields

GET  Get Card by ID
https://localhost:8452/api/cards/1                 200 • 17 ms • 523 B • 3
PASS  Status code is 200
PASS  Card has correct ID
PASS  Card has all properties

GET  Get Card by ID - Not Found
https://localhost:8452/api/cards/999               404 • 5 ms • 365 B • 1
PASS  Status code is 404

POST  Get Cards by IDs Batch
https://localhost:8452/api/cards/batch             200 • 29 ms • 1.025 KB • 3
PASS  Status code is 200
PASS  Returns correct number of cards
PASS  Cards have correct IDs

GET  Verify Element Distribution
https://localhost:8452/api/cards                   200 • 5 ms • 3.588 KB • 3
PASS  Status code is 200
PASS  Has 5 cards of each element
PASS  Power levels are 1-5
```

2. Auth Service

a. `docker build -f auth-service/Dockerfile.test-isolation -t auth-service-test .`

b. `docker run --name auth-service-test -p 8931:8931 auth-service-test`

Auth Service Isolation test screenshot:



Auth Service - Isolation Test
- POST Register User - Success
- POST Register - Invalid Email
- POST Register - Weak Password
- POST Login User - Success
- POST Login - Invalid Credentials

Card Service - Test Isolation
Game Service - Test Isolation
Match History Service - Test Isolation
User Service - Test Isolation
ZM2 Card Game - Complete Integration Tests

Auth Service - Isolation Test - Run results
Ran today at 02:52:58 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Errors	Avg. Resp. Time
Runner	none	1	775ms	12	0	63 ms

All Tests Passed (10) Failed (2) Skipped (0) Errors (0) View Summary

Iteration 1

POST Register User - Success
https://localhost:8931/api/auth/register 201 • 275 ms • 780 B • 3
PASS Status code is 201
PASS Response has token
PASS Response has user data

POST Register - Invalid Email
https://localhost:8931/api/auth/register 400 • 8 ms • 456 B • 2
PASS Status code is 400
PASS Response has error message

POST Register - Weak Password
https://localhost:8931/api/auth/register 400 • 13 ms • 591 B • 2
PASS Status code is 400
PASS Response has password error

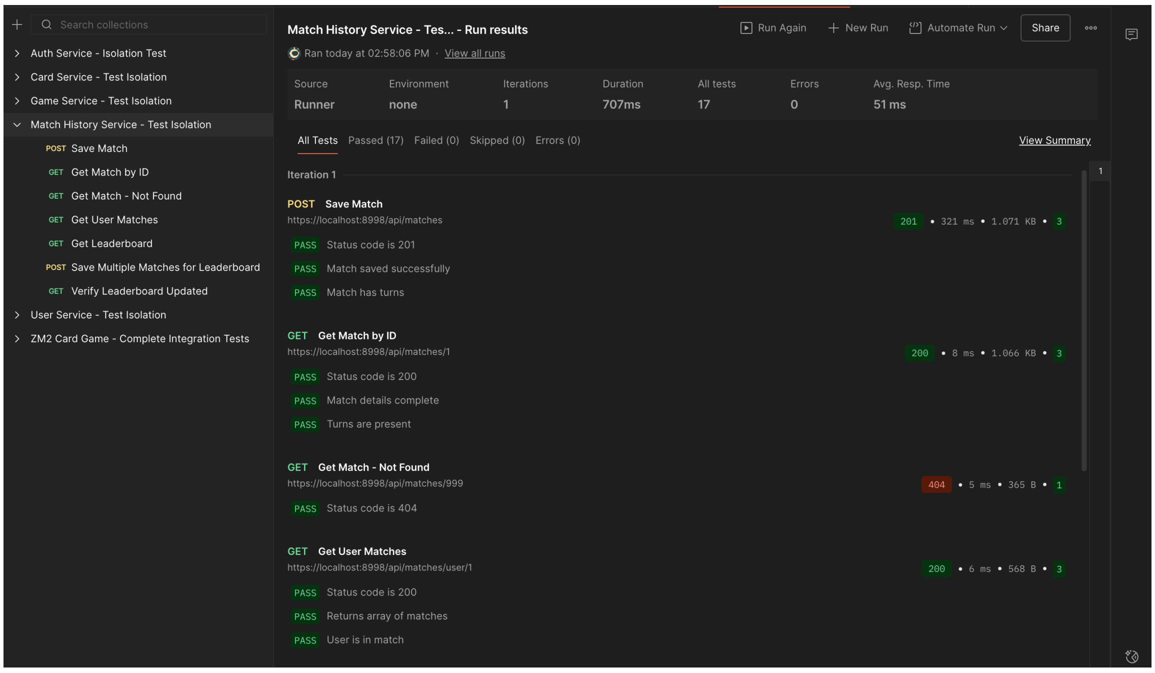
POST Login User - Success
https://localhost:8931/api/auth/login 200 • 16 ms • 770 B • 3
PASS Status code is 200
PASS Response has token
PASS Response has complete user data

3. Match History Service

a. `docker build -f match-history-service/Dockerfile.test-isolation -t match-history-service-test .`

b. `docker run --name match-history-service-test -p 8998:8998 match-history-service-test`

Match History Service Isolation test screenshot:



Auth Service - Isolation Test
Card Service - Test Isolation
Game Service - Test Isolation
Match History Service - Test Isolation
- POST Save Match
- GET Get Match by ID
- GET Get Match - Not Found
- GET Get User Matches
- GET Get Leaderboard
- POST Save Multiple Matches for Leaderboard
- GET Verify Leaderboard Updated

User Service - Test Isolation
ZM2 Card Game - Complete Integration Tests

Match History Service - Tes... - Run results
Ran today at 02:58:06 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Errors	Avg. Resp. Time
Runner	none	1	707ms	17	0	51 ms

All Tests Passed (17) Failed (0) Skipped (0) Errors (0) View Summary

Iteration 1

POST Save Match
https://localhost:8998/api/matches 201 • 321 ms • 1.071 KB • 3
PASS Status code is 201
PASS Match saved successfully
PASS Match has turns

GET Get Match by ID
https://localhost:8998/api/matches/1 200 • 8 ms • 1.066 KB • 3
PASS Status code is 200
PASS Match details complete
PASS Turns are present

GET Get Match - Not Found
https://localhost:8998/api/matches/999 404 • 5 ms • 365 B • 1
PASS Status code is 404

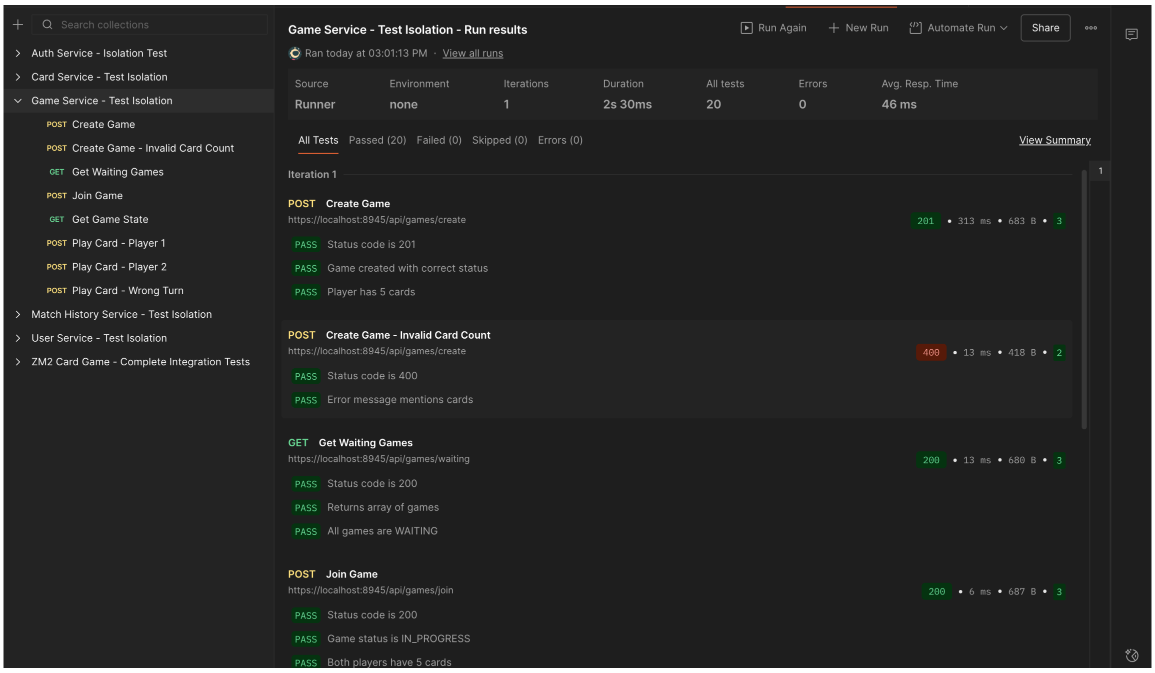
GET Get User Matches
https://localhost:8998/api/matches/user/1 200 • 6 ms • 568 B • 3
PASS Status code is 200
PASS Returns array of matches
PASS User is in match

4. Game Service

a. `docker build -f game-service/Dockerfile.test-isolation -t game-service-test .`

b. `docker run --name game-service-test -p 8945:8945 game-service-test`

Game Service Isolation test screenshot:



Search collections

- Auth Service - Isolation Test
- Card Service - Test Isolation
- Game Service - Test Isolation
  - POST Create Game
  - POST Create Game - Invalid Card Count
  - GET Get Waiting Games
  - POST Join Game
  - GET Get Game State
  - POST Play Card - Player 1
  - POST Play Card - Player 2
  - POST Play Card - Wrong Turn
- Match History Service - Test Isolation
- User Service - Test Isolation
- ZM2 Card Game - Complete Integration Tests

Game Service - Test Isolation - Run results

Ran today at 03:01:13 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Errors	Avg. Resp. Time
Runner	none	1	2s 30ms	20	0	46 ms

All Tests Passed (20) Failed (0) Skipped (0) Errors (0) View Summary

Iteration 1

POST Create Game
https://localhost:8945/api/games/create
201 • 313 ms • 683 B • 3
PASS Status code is 201
PASS Game created with correct status
PASS Player has 5 cards

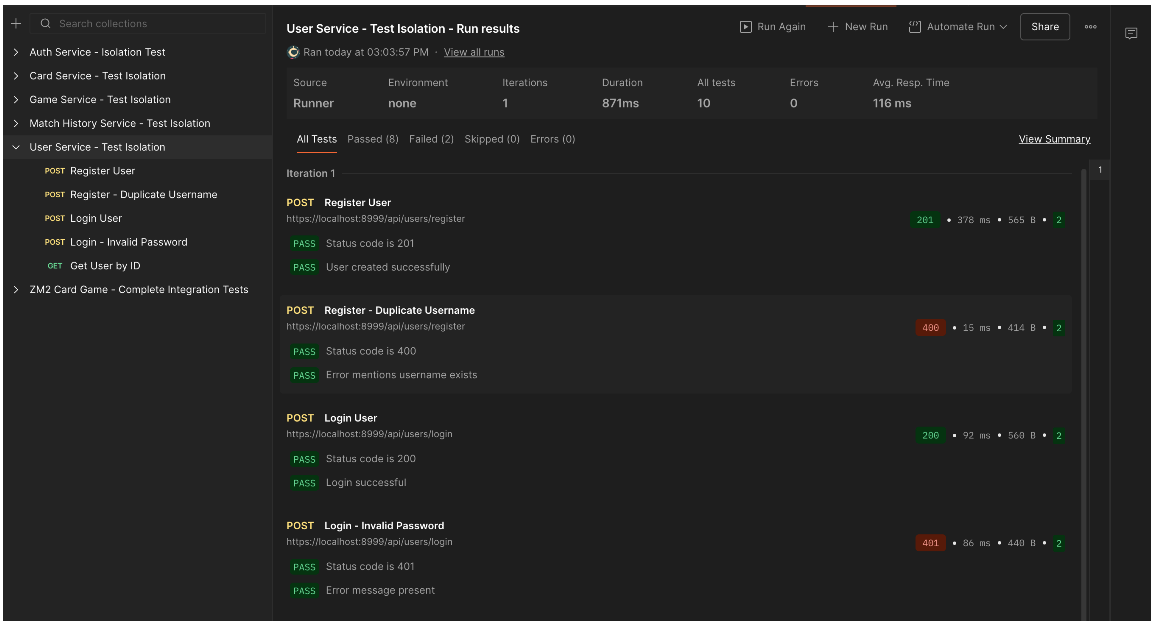
POST Create Game - Invalid Card Count
https://localhost:8945/api/games/create
400 • 13 ms • 418 B • 2
PASS Status code is 400
PASS Error message mentions cards

GET Get Waiting Games
https://localhost:8945/api/games/waiting
200 • 13 ms • 680 B • 3
PASS Status code is 200
PASS Returns array of games
PASS All games are WAITING

POST Join Game
https://localhost:8945/api/games/join
200 • 6 ms • 687 B • 3
PASS Status code is 200
PASS Game status is IN_PROGRESS
PASS Both players have 5 cards

5. User Service
   a. `docker build -f user-service/Dockerfile.test-isolation -t user-service-test .`
   b. `docker run --name user-service-test -p 8999:8999 user-service-test`

User Service Isolation test screenshot:



Search collections

- Auth Service - Isolation Test
- Card Service - Test Isolation
- Game Service - Test Isolation
- Match History Service - Test Isolation
- User Service - Test Isolation
  - POST Register User
  - POST Register - Duplicate Username
  - POST Login User
  - POST Login - Invalid Password
  - GET Get User by ID
- ZM2 Card Game - Complete Integration Tests

User Service - Test Isolation - Run results

Ran today at 03:03:57 PM · View all runs

Source	Environment	Iterations	Duration	All tests	Errors	Avg. Resp. Time
Runner	none	1	871ms	10	0	116 ms

All Tests Passed (8) Failed (2) Skipped (0) Errors (0) View Summary

Iteration 1

POST Register User
https://localhost:8999/api/users/register
201 • 378 ms • 565 B • 2
PASS Status code is 201
PASS User created successfully

POST Register - Duplicate Username
https://localhost:8999/api/users/register
400 • 15 ms • 414 B • 2
PASS Status code is 400
PASS Error mentions username exists

POST Login User
https://localhost:8999/api/users/login
200 • 92 ms • 560 B • 2
PASS Status code is 200
PASS Login successful

POST Login - Invalid Password
https://localhost:8999/api/users/login
401 • 86 ms • 440 B • 2
PASS Status code is 401
PASS Error message present

OR Run docker compose ***docker-compose -f docker-compose-isolation-test.yml up --build*** from the root folder.

Import in postman the postman testing collection files located in the postman testing collections folder in the `docs` folder and run the collection.

- auth-service.json (in AuthService folder) to test auth-service
- game-service.json (in GameService folder) to test game-service
- user-service.json (in UserService folder) to test user-service
- card-service.json (in CardService folder) to test card-service
- match-history-service.json (in MatchService folder) to test match-history-service

2. Integration Testing via API Gateway (Postman Collections)

Integration testing validates the interaction between multiple microservices through the **API Gateway**. A **Postman integration test collection** is executed against the API Gateway.

The same logical test cases used in isolation testing are reused, but routed through the gateway.

Test Scenarios Include:

1. Authentication & User Management (Register player, Login, Player profile)
2. Game Service (Get Cards, Specific Card, Batch Cards)
3. Game Flow (Complete Game)
4. Match History & Leaderboard
5. User Stats Verification

Test Coverage:

- Successful requests (200 OK)
- Error scenarios (4xx / 5xx)

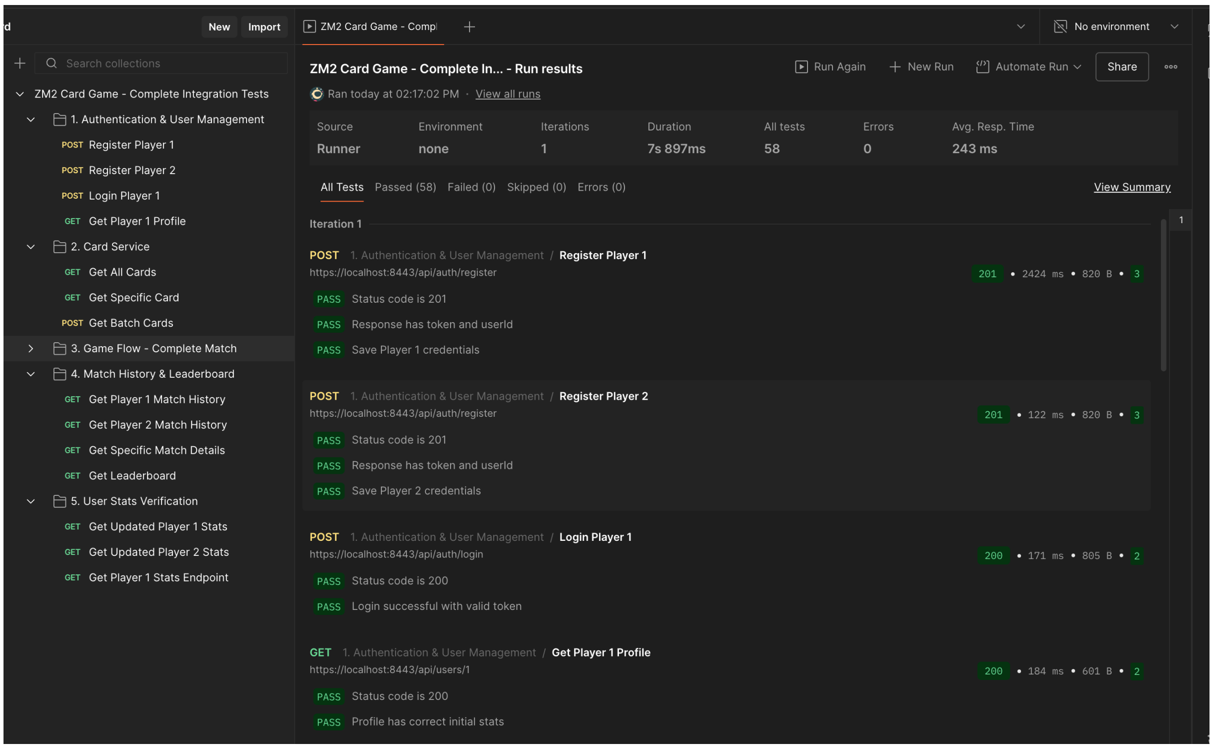
All the Postman collections (JSON file) are located in the `/docs` folder of the project.

Execution Instructions:

1. Start all services using Docker Compose: `docker compose up --build`

2. Import the Integration Postman collection.
3. Execute the collection against the API Gateway endpoint.
4. Observe responses and verify correct service interaction.

During team testing, all tests were passed successfully. Screenshot:



ZM2 Card Game - Complete Integration Tests
1. Authentication & User Management
POST Register Player 1
POST Register Player 2
POST Login Player 1
GET Get Player 1 Profile
2. Card Service
GET Get All Cards
GET Get Specific Card
POST Get Batch Cards
3. Game Flow - Complete Match
4. Match History & Leaderboard
GET Get Player 1 Match History
GET Get Player 2 Match History
GET Get Specific Match Details
GET Get Leaderboard
5. User Stats Verification
GET Get Updated Player 1 Stats
GET Get Updated Player 2 Stats
GET Get Player 1 Stats Endpoint

ZM2 Card Game - Complete In... - Run results
Ran today at 02:17:02 PM

Source	Environment	Iterations	Duration	All tests	Errors	Avg. Resp. Time
Runner	none	1	7s 897ms	58	0	243 ms

All Tests Passed (58) Failed (0) Skipped (0) Errors (0)

Iteration 1

```
POST 1. Authentication & User Management / Register Player 1
https://localhost:8443/api/auth/register                    201 • 2424 ms • 820 B • 3
PASS Status code is 201
PASS Response has token and userId
PASS Save Player 1 credentials

POST 1. Authentication & User Management / Register Player 2
https://localhost:8443/api/auth/register                    201 • 122 ms • 820 B • 3
PASS Status code is 201
PASS Response has token and userId
PASS Save Player 2 credentials

POST 1. Authentication & User Management / Login Player 1
https://localhost:8443/api/auth/login                       200 • 171 ms • 805 B • 2
PASS Status code is 200
PASS Login successful with valid token

GET 1. Authentication & User Management / Get Player 1 Profile
https://localhost:8443/api/users/1                          200 • 184 ms • 601 B • 2
PASS Status code is 200
PASS Profile has correct initial stats
```

3. Performance Testing (Locust)

Performance testing focuses on evaluating system behavior under load.

Locust is selected as the performance testing tool. Custom Locust scripts are intended to simulate:

- multiple concurrent users,
- repeated game actions,
- repeated card acquisition requests.

Target Areas:

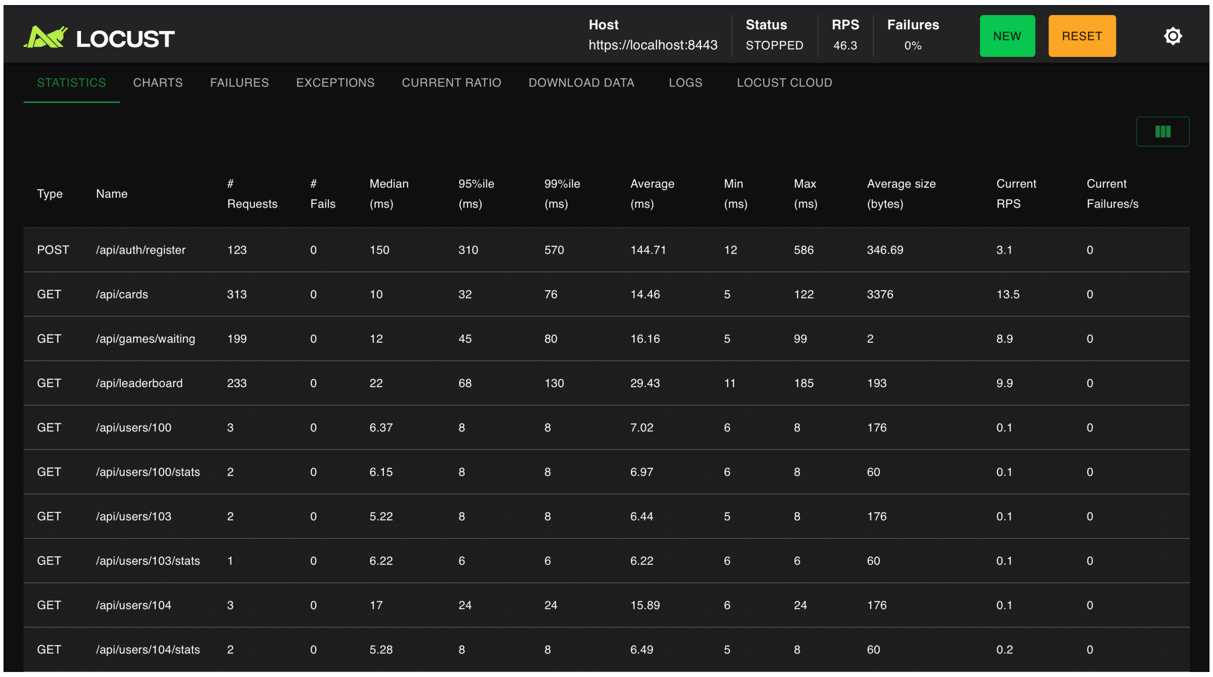
- Game Service
- Card Service
- API Gateway routing

The testing `Locustfile` is located in the `/docs` folder of the project and in the root folder of project.

Execution Instructions:

1. Start backend services using Docker Compose.
2. Run `Locust` in the root folder of project and visit: http://localhost:8089
3. Configure User Loads and **locust --host =** `https://localhost:8443`
4. Monitor response times and error rates.

During team testing, all tests were passed successfully. Screenshot:
Number of Users: 100 & Ramp up: 4



Host	Status	RPS	Failures
https://localhost:8443	STOPPED	46.3	0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/api/auth/register	123	0	150	310	570	144.71	12	586	346.69	3.1	0
GET	/api/cards	313	0	10	32	76	14.46	5	122	3376	13.5	0
GET	/api/games/waiting	199	0	12	45	80	16.16	5	99	2	8.9	0
GET	/api/leaderboard	233	0	22	68	130	29.43	11	185	193	9.9	0
GET	/api/users/100	3	0	6.37	8	8	7.02	6	8	176	0.1	0
GET	/api/users/100/stats	2	0	6.15	8	8	6.97	6	8	60	0.1	0
GET	/api/users/103	2	0	5.22	8	8	6.44	5	8	176	0.1	0
GET	/api/users/103/stats	1	0	6.22	6	6	6.22	6	6	60	0.1	0
GET	/api/users/104	3	0	17	24	24	15.89	6	24	176	0.1	0
GET	/api/users/104/stats	2	0	5.28	8	8	6.49	5	8	60	0.2	0

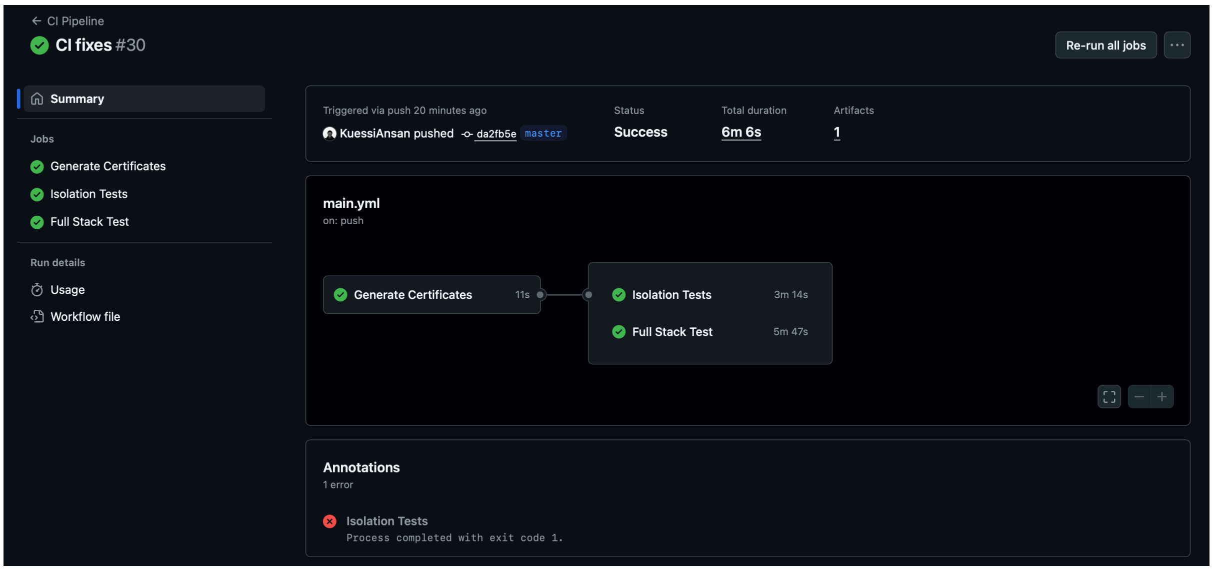
Continuous Integration – GitHub Actions

To automate testing during code updates by executing tests on every push or pull request.

A GitHub Actions workflow is used to:

1. Build all microservices.
2. Run **unit tests** for each microservice in isolation.
3. Run **integration tests** through the API Gateway.

Workflow File Location: `/docs` folder of the project also in `.github/workflow/main.yml`



← CI Pipeline

CI fixes #30

Re-run all jobs

Summary

Jobs

- Generate Certificates
- Isolation Tests
- Full Stack Test

Run details

- Usage
- Workflow file

Triggered via push 20 minutes ago: KuessiAnsan pushed da2fb5e master

Status: Success

Total duration: 6m 6s

Artifacts: 1

main.yml
on: push

Generate Certificates 11s → Isolation Tests 3m 14s; Full Stack Test 5m 47s

Annotations
1 error

Isolation Tests
```
Process completed with exit code 1.
```

We have also implemented some tests in the backend (In the test folder of each service):

Method	Role
testRegisterUser()	User registration
`testGetUser()`	`test getting a user`
`testGetAllCards()`	`Get All Cards`
`testGetCardById()`	`Get a specific card based on it's ID`
`testCreateGame()`	`create a game`
`testGetGame()`	`get a game`
`testFireBeatsWater()`	`test comparison of cards to see winning and losing card - Fire VS Water`
`testFireVsEarthComparePower()`	`test comparison of cards to see winning and losing card - Fire VS Earth`
`testSpiritBeatsFire()`	`test comparison of cards to see winning and losing card - Spirit VS Fire`
`testCompleteDraw()`	`test draw by implementing its logic`

OpenAPI:

An OpenAPI 3.1 specification for the API Gateway has been defined and is available in the `/docs` folder of the repository.

Also at: https://localhost:8443/v3/api-docs/merged.yaml

REST API Endpoints:

Method	Endpoint	Description
`POST`	`/api/auth/register`	Register new user
`POST`	`/api/auth/login`	Login
`POST`	`/api/users/{id}/stats`	Display users statistics
`POST`	`/api/cards/batch`	Get multiple cards
`POST`	`/api/games/{id}/play-cards`	Play a card
`POST`	`/api/games/{id}/join`	Join an existing game
`POST`	`/api/games/{id}/create`	Create a new game
`POST`	`/api/matches`	Save all matches
`GET`	`/api/users`	Get all users
`GET`	`/api/users/{id}`	Get user by Id
`GET`	`/api/users/{id}/stats`	Get users statistics
`GET`	`/api/cards`	Get all cards
`GET`	`/api/cards/{id}`	Get card by Id
`GET`	`/api/games/{id}/state`	Get current game state with scores and turn info
`GET`	`/api/games/{id}/waiting`	Get waiting games
`GET`	`/api/games/{id}`	Get game informations
`GET`	`/api/match/{id}`	Get match details
`GET`	`/api/match/user/{userId}`	Get users match history
`GET`	`/api/leaderboard`	Get Leaderboard
`PUT`	`/api/users/{id}`	Update User Profile
`WS`	`/api/ws/game/{gameId}`	Web Socket endpoint

Security - Data

Input Sanitization

Data sanitized: Email.

User email addresses are sanitized and validated before being stored in the database. Validation is enforced using a regular expression pattern to ensure strong formatting rules and reduce the risk of invalid or malicious input.

Validation pattern used:

`"^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z\\d@$!%*?&]{8,}$"`

Validation rules enforced:

- Minimum length of 8 characters
- At least one lowercase letter
- At least one uppercase letter
- At least one digit
- At least one special character (`@$!%*?&`)

If an email does not match this pattern, it is considered invalid and is rejected before database persistence.

Email Uniqueness Constraint:

- Each email address must be **unique**
- The system checks for existing records before insertion
- Duplicate emails are rejected at the application and/or database level

Microservices Using This Input

- **Auth Service**
  - Receives the email during user registration and login
  - Validates credentials and generates authentication tokens
- **User Service**
  - Stores the email as part of the user profile
  - Uses the email for account management and profile operations
- **Match History Service**
  - Requests user information (including email-linked user identity) from the User Service when retrieving match records
- **Game Service**
  - Indirectly relies on email-based user identity when updating player statistics through the User Service

Data Encryption at Rest.

Encrypted Data: User Password.

The password represents the secret credential used to authenticate a user during login.

Database Storing the Encrypted Data

- Stored in the **User Service database**
- Never stored or transmitted in plain text

Microservices Handling the Data

- **Auth Service**
  - Receives the plain-text password during registration and login
  - Encrypts the password before storage
  - Verifies login credentials by comparing hashes
- **User Service**
  - Persists the encrypted password hash
  - Never performs decryption or exposes the password

Encryption / Decryption Strategy

- Passwords are encrypted using `BCryptPasswordEncoder`
- BCrypt is a one-way hashing algorithm:
  - Passwords cannot be decrypted
  - Each password is automatically salted
  - Hash comparison is used for authentication instead of decryption

Where Encryption Occurs

- Encryption happens in the **Auth Service** during user registration
- During login, the Auth Service hashes the provided password and compares it with the stored hash retrieved from the User Service

Security – Authorization and Authentication

We adopt a **distributed authentication and authorization** model. In this scenario, each microservice is responsible for validating access tokens independently, without relying on a centralized authorization component at request time.

1. Authentication Flow (Login)

Actors involved:

- Gateway
- Auth Service
- User Service
- Client (Frontend)

Steps:

1. The user sends a **login request** (credentials) to the **Gateway**.
2. The Gateway routes the request to the **Auth Service**.
3. The Auth Service validates the credentials by calling the **User Service** login endpoint.
4. If credentials are valid, the Auth Service generates:
    - an **Access Token (JWT)**
    - (optionally) an ID Token
5. The token is returned to the client via the Gateway.

At this stage, the user is authenticated.

2. Authorization Flow

After authentication, **all subsequent requests** follow the distributed validation model.

Steps

1. The client sends a request containing the **Access Token (JWT)** in the `Authorization` header.
2. The request passes through the **Gateway**.
3. The Gateway forwards the request to the target microservice.
4. The receiving microservice:

    - Validates the token signature
    - Verifies token expiration
    - Checks required claims (e.g., user ID, role)
5. If validation succeeds, access to the protected resource is granted.
6. If validation fails, the request is rejected with an authorization error.

3. Token Validation Responsibility

In this **distributed model**:

- **Each microservice validates the token independently**
- There is **no runtime dependency** on the Auth Service for token verification
- This avoids bottlenecks and improves scalability

Microservices performing token validation

- User Service
- Card Service
- Game Service
- Match History Service

4. Key Management (Signing and Verification)

Token signing

- Access Tokens are **signed by the Auth Service**
- A cryptographic signing key is used during token generation

Token verification

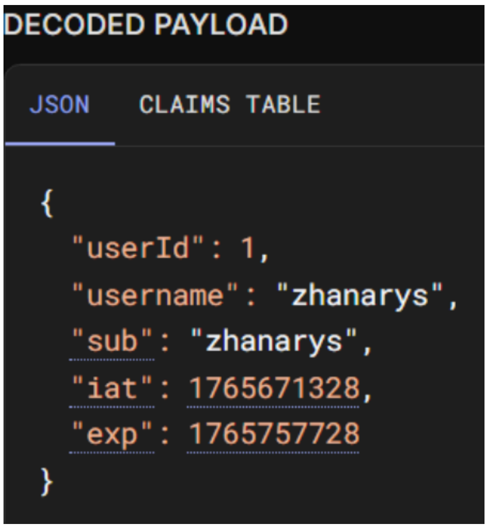
- The **same verification key** (or corresponding public key) is shared with all microservices
- Each microservice stores the key securely (e.g., environment variables or secrets)
- Microservices use this key to:
    - Verify the JWT signature
    - Ensure the token has not been tampered with

Key storage

- Keys are **not hardcoded**
- Stored as configuration or secrets at deployment time
- Identical verification key is available to all services that need to validate tokens

5. Access Token Payload Structure

The Access Token is a **JWT** containing the following claims:



```
{
  "userId": 1,
  "username": "zhanarys",
  "sub": "zhanarys",
  "iat": 1765671328,
  "exp": 1765757728
}
```

- `sub` – User identifier
- `iat` – Issued-at timestamp
- `exp` – Expiration timestamp

6. Handling Expired Access Tokens

Backend behavior: Each microservice checks the `exp` claim

If the token is expired:

- The request is rejected
- Access to the protected resource is denied

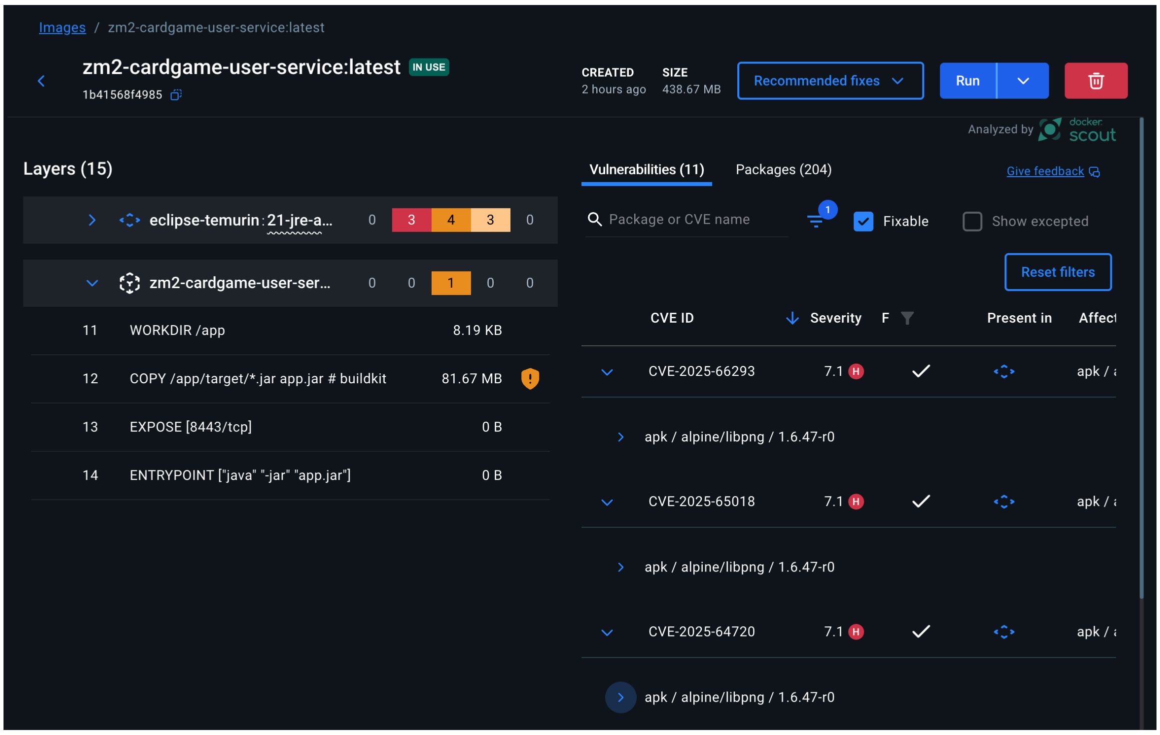
Frontend behavior. When an expired token is detected:

- The user is automatically redirected to the **login page**
- A new authentication flow is required

Security – Analyses

During the project, no free static code analysis tool for Java could be reliably integrated within the given constraints, therefore static code analysis was not included.

Docker Scout was used to analyze the project's Docker images. At first Test: We used an Alpine-based base image, and there three high-severity vulnerabilities were identified.


Images / zm2-cardgame-user-service:latest

zm2-cardgame-user-service:latest IN USE
1b41568f4985

CREATED 2 hours ago · SIZE 438.67 MB · Recommended fixes · Run

Analyzed by docker scout

Layers (15)

eclipse-temurin:21-jre-a...	0	3	4	3	0	
zm2-cardgame-user-ser...	0	0	1	0	0	

11	WORKDIR /app	8.19 KB
12	COPY /app/target/*.jar app.jar # buildkit	81.67 MB
13	EXPOSE [8443/tcp]	0 B
14	ENTRYPOINT ["java" "-jar" "app.jar"]	0 B

Vulnerabilities (11) · Packages (204) · Give feedback

Package or CVE name · Fixable [✓] · Show excepted · Reset filters

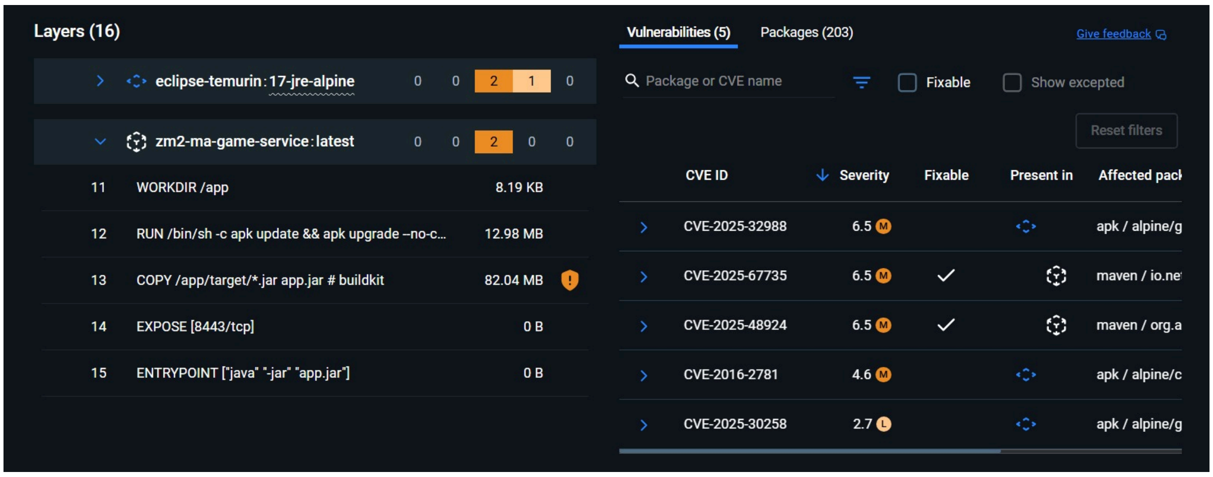
CVE ID	Severity	F	Present in	Affect
CVE-2025-66293	7.1 H	✓		apk /
apk / alpine/libpng / 1.6.47-r0				
CVE-2025-65018	7.1 H	✓		apk /
apk / alpine/libpng / 1.6.47-r0				
CVE-2025-64720	7.1 H	✓		apk /
apk / alpine/libpng / 1.6.47-r0				

We changed the runtime stage image in the dockerfiles

We used this image instead: eclipse-temurin:17.0.13_11-jre-alpine

And also we update all Alpine packages to latest security patches

Last Result was:


Layers (16)

eclipse-temurin : 17-jre-alpine	0	0	2	1	0	
zm2-ma-game-service : latest	0	0	2	0	0	

11	WORKDIR /app	8.19 KB
12	RUN /bin/sh -c apk update && apk upgrade --no-c...	12.98 MB
13	COPY /app/target/*.jar app.jar # buildkit	82.04 MB
14	EXPOSE [8443/tcp]	0 B
15	ENTRYPOINT ["java" "-jar" "app.jar"]	0 B

Vulnerabilities (5) Packages (203) Give feedback

Package or CVE name Fixable Show excepted Reset filters

CVE ID	Severity	Fixable	Present in	Affected pack
CVE-2025-32988	6.5 M			apk / alpine/g
CVE-2025-67735	6.5 M	✓		maven / io.ne
CVE-2025-48924	6.5 M	✓		maven / org.a
CVE-2016-2781	4.6 M			apk / alpine/c
CVE-2025-30258	2.7 L			apk / alpine/g

Threat Model

Scope: Frontend Angular → WebSocket channel.→ API Gateway → Microservices (User / Card / Game / Match-History) → PostgreSQL DBs

1) Work from a Model (System Model & Data Flow)

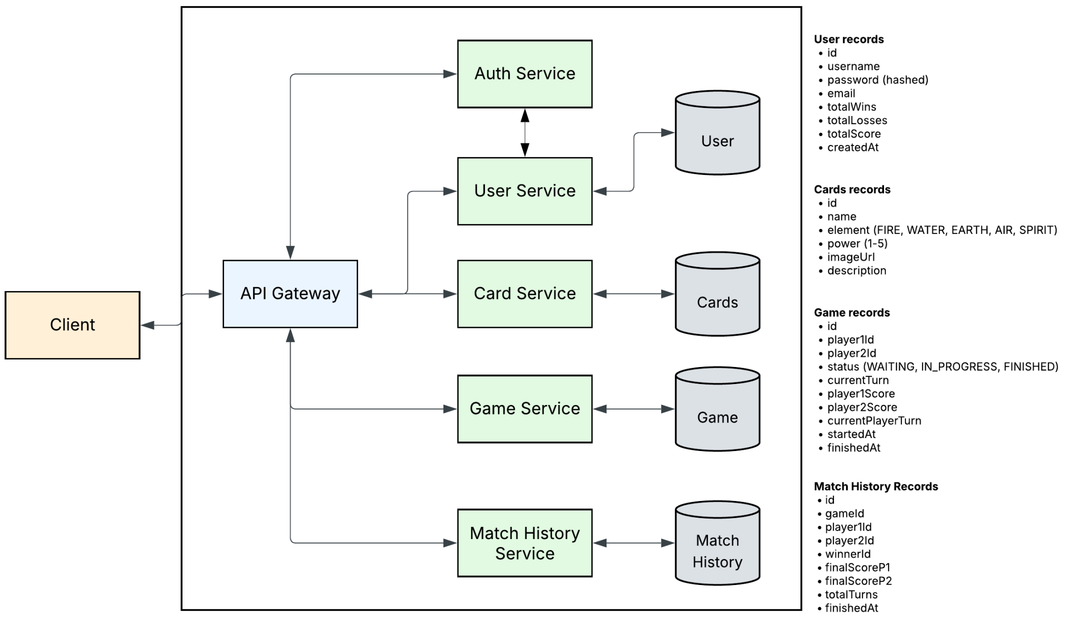
The model focuses on a real-time multiplayer card game: the player's browser connects to an Angular frontend, which communicates with a Spring Boot backend through both REST (for login, profile, card retrieval, and game setup) and WebSocket (for real-time gameplay).

The backend is fully authoritative: it validates moves, applies game rules, updates scores, and stores turn results.

All persistent data — users, cards, active games, turn logs, and archived match history — is stored in a PostgreSQL database.

The system exposes only three attack surfaces: public REST endpoints, real-time WebSocket messages, and backend database access.

No external third-party services or admin console are currently used.



User records
- id
- username
- password (hashed)
- email
- totalWins
- totalLosses
- totalScore
- createdAt

Cards records
- id
- name
- element (FIRE, WATER, EARTH, AIR, SPIRIT)
- power (1-5)
- imageUrl
- description

Game records
- id
- player1Id
- player2Id
- status (WAITING, IN_PROGRESS, FINISHED)
- currentTurn
- player1Score
- player2Score
- currentPlayerTurn
- startedAt
- finishedAt

Match History Records
- id
- gameId
- player1Id
- player2Id
- winnerId
- finalScoreP1
- finalScoreP2
- totalTurns
- finishedAt

2) Identify Assets

- **A1: Personally identifiable information (PII) and match history.**

Player profiles, account settings, and archived match/turn history.

- **A2: Credentials, sessions, refresh tokens.**

User passwords, access tokens, WebSocket session authentication, refresh tokens.

- **A3: WebSocket auth tokens & session secrets.**

Tokens/keys used to establish and validate WSS connections and per-connection session state.

- **A4: Active game state and card hands.**

In-progress game records: dealt cards, current turn, played cards, temporary game state.

- **A5: Match results, leaderboards & audit logs.**

Final scores, archived matches, leaderboard aggregates, and any administrative/audit trail entries.

- **A6: Configuration secrets (DB credentials, JWT signing keys, env vars).**

Database credentials, container registry credentials.

- **A7: Code integrity and CI/CD artifacts.**

Source code, build artifacts, Docker images, dependency manifests and CI pipeline configuration.

- **A8: Service availability (matchmaking, API, WebSocket uptime).**

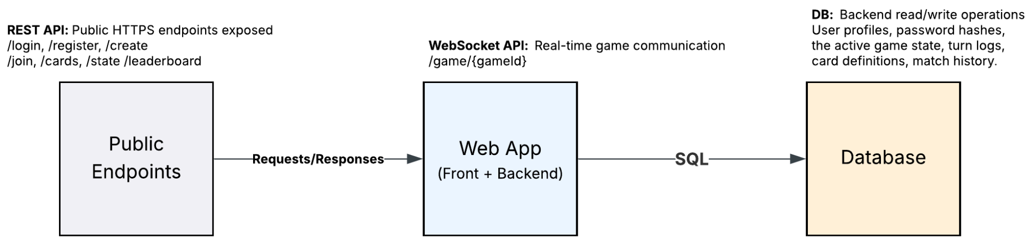
Availability of REST API, matchmaking, WSS game channels and database responsiveness.

3) Identify Attack Surfaces - ZM2

Public endpoints: Credentials, user IDs, game creation requests, card datasets, profile lookups, leaderboard data

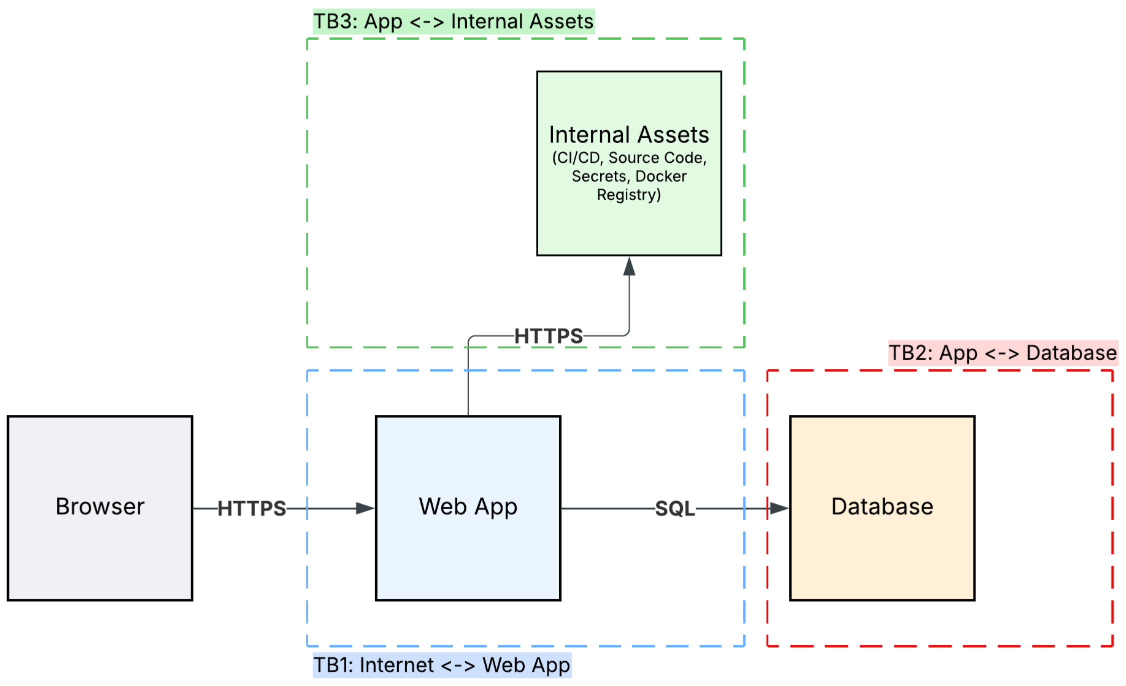
WebSocket Real time messages: Card choices, turn actions, real-time score updates, match-result broadcasts

DB: User records, password hashes, card definitions, active game state, turn history, match history, leaderboards.


REST API: Public HTTPS endpoints exposed
/login, /register, /create
/join, /cards, /state /leaderboard
WebSocket API: Real-time game communication
/game/{gameId}
DB: Backend read/write operations
User profiles, password hashes,
the active game state, turn logs,
card definitions, match history.
Public Endpoints
Requests/Responses
Web App
(Front + Backend)
SQL
Database

4) Trust Boundaries - ZM2

TB1: **Internet ↔ Web App;** TB2: **Web App ↔ Database;** TB3: **Web App ↔ Internal Assets**


TB3: App <-> Internal Assets
Internal Assets
(CI/CD, Source Code, Secrets, Docker Registry)
HTTPS
TB2: App <-> Database
Browser
HTTPS
Web App
SQL
Database
TB1: Internet <-> Web App

5) Identify Threats (STRIDE)

Threats are grouped under STRIDE categories and reference assets and boundaries.

S – Spoofing

- **T1:** Credential stuffing against **/api/users/login** leads to account takeover (affects **A2, A1**; boundary **TB1**).
- **T2:** Weak or stolen WebSocket tokens allow a user to impersonate another player in a game session (affects **A2, A3**; **TB1**).
- **T3:** CI/CD credential leakage allows attackers to impersonate internal developers and inject malicious code (affects **A7, A6**; **TB3**).

T – Tampering

- **T4:** Forged or modified WebSocket messages allow illegal moves, card manipulation, or score tampering (affects **A4, A5**; **TB1**).
- **T5:** Manipulated REST requests (ID change, gameId change) allow joining or modifying games the user does not own (affects **A4, A5, A1**; **TB1**).

- **T6:** Supply-chain tampering of dependencies, builds, or Docker images compromises backend logic (affects **A7, A6**; **TB3**).

R – Repudiation

- **T7:** Missing or mutable audit logs prevent attribution of gameplay actions, match results, or administrative changes (affects **A5**; **TB2/TB3**).

I – Information Disclosure

- **T8:** Insecure direct object reference (IDOR) exposes other players' profiles, match histories, or stats (affects **A1**; **TB1**).
- **T9:** WebSocket traffic without proper authentication reveals real-time game state to unauthorized observers (affects **A3, A4**; **TB1**).
- **T10:** Verbose error messages or stack traces leak secrets or database structure (affects **A6**; **TB1**).

D – Denial of Service

- **T11:** High-frequency REST or WebSocket spam floods game logic, preventing matches from starting or progressing (affects **A8**; **TB1**).
- **T12:** Heavy leaderboard or game-state DB queries overload the database, degrading all gameplay (affects **A8**; **TB2**).

E – Elevation of Privilege

- **T13:** Broken access control on game endpoints allows players to access or modify other users' resources (affects **A5, A1**; **TB1**).
- **T14:** CI/CD misconfigurations allow attackers to escalate privileges and push modified builds to production (affects **A7, A6**; **TB3**).

6) Mitigate Threats

Authentication and Sessions (T1–T3)

- Rate limiting and breached-password checks on **/login**.
- Strong password hashing (BCrypt).
- Short-lived JWT access tokens; HttpOnly + Secure cookies if using cookie-based auth.

- WebSocket handshake must validate the same token/session as REST.
- Session revocation and forced logout after token theft or suspicious activity.
- Rotate JWT signing keys and CI/CD access tokens regularly.

Authorization and Access Control (T5, T8, T13)

- Consistent authorization checks on all REST endpoints (game ownership, user ownership).
- Deny-by-default: a player cannot access another user's game, match, or profile data.
- Prevent IDOR by validating all IDs against the authenticated user.
- Comprehensive authorization tests for join-game, play-turn, and match-history endpoints.

Input Validation and Game Logic Integrity (T4, T5)

- Server-side validation for every move: card must belong to the user, turn order must match game state.
- Reject client-sent fields that attempt to override game logic (scores, state, opponent data).
- Add message sequence numbers or nonces to WebSocket events to prevent replay or reordering.
- Enforce authoritative game rules strictly on backend only.

Secrets, Supply Chain, and CI/CD Security (T3, T6, T10, T14)

- Store all sensitive secrets (DB credentials, JWT keys) in a secure secret manager or encrypted environment.
- No secrets in repositories or CI logs.
- Pin and verify dependencies; run dependency vulnerability scanning.
- Sign Docker images; enable artifact integrity verification.
- Use least-privilege access for CI runners and production deploy keys.

- Enable multi-factor authentication for repository and CI/CD administrative access.

Data Protection (T8, T9)

- Enforce TLS for all REST and WebSocket traffic.
- Avoid exposing unnecessary user information in game responses.
- Limit detailed game state visibility to authenticated players only.
- Sanitize error messages before returning them to clients.

Logging, Monitoring, and Resilience (T7, T11, T12)

- Centralized, immutable logging for matches, turns, and player actions.
- Store audit logs with restricted access; prevent modification.
- Rate limiting for WebSocket and REST endpoints.
- Detect gameplay anomalies (rapid message spam, invalid sequences).
- Resource quotas and circuit breakers to protect the backend and database from overload.
- Backups encrypted and tested regularly; clear incident response runbooks.

Generative AI

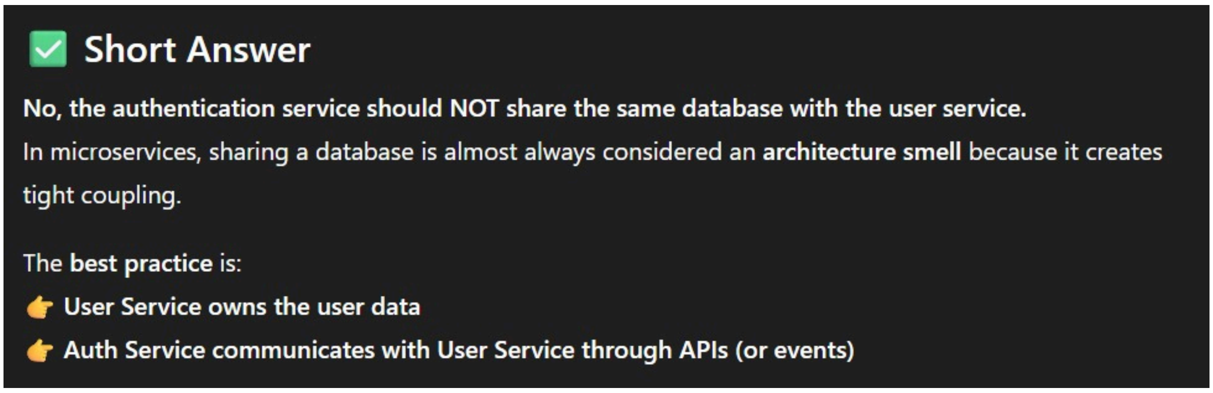
Generative AI tools were used mainly as a **learning and support aid** during the project. The primary system used was **ChatGPT**.

AI was used to:

- understand complex software engineering concepts by asking questions such as *“Can you explain this in a simple way?”*,
- clarify architectural topics related to microservices, REST APIs, testing, and security.
- explore **possible architectural and design choices** for the project and understand their advantages and disadvantages.

For example: when designing authentication, we evaluated whether the **Auth service should retrieve user information by calling the User Service** or by using a **replicated user database or sharing the same database with User Service**.

We used AI to understand the trade-offs of those approaches, such as data consistency, coupling between services, and maintenance complexity. Based on this analysis, we chose to connect the Auth service directly to the User Service instead of maintaining a duplicated database. AI Response was:



✅ Short Answer

No, the authentication service should NOT share the same database with the user service.
In microservices, sharing a database is almost always considered an **architecture smell** because it creates tight coupling.

The **best practice** is:
👉 **User Service owns the user data**
👉 **Auth Service communicates with User Service through APIs (or events)**

This decision was later discussed with the professor, who confirmed the same reasoning and conclusions that were previously identified using AI. Similar questions were asked throughout the project to better understand design choices and their implications.

Another example of AI usage was during the design of the card system, where AI was used to help brainstorm and structure 25 different card types and names, ensuring variety and consistency while respecting the predefined game rules and balance constraints.

A limitation of using AI is that when it does not have complete and detailed information about the project, it may provide answers or suggestions that do not fully align with the actual project design, requiring careful validation and correction.

The AI did not make final decisions or implement functionality. All architectural choices and project decisions were made by the team after evaluating the provided explanations.

Overall, Generative AI helped improve understanding and reasoning, but required careful validation to ensure that all information matched the actual project design and constraints.

Additional Features

1	**As a player, I want to see who turn it is so that I know if I can play or not**
2	**As a player, I want to be able to see the cards in my hand so that I know that card I can play next**

Additional Feature 1

This feature allows a player to see whose turn it currently is during a match. It prevents confusion and ensures that players know when they are allowed to play, improving game flow and user experience.

The backend maintains the current turn state for each game and exposes it through the game state API and WebSocket updates.

The frontend displays the active player’s turn in real time, updating automatically when the turn changes.

Additional Feature 2

This feature allows a player to view the cards currently available in their hand. It enables players to make informed strategic decisions by knowing which cards they can play next.

The backend provides the player’s current hand as part of the game state, accessible through REST endpoints or WebSocket messages.

The frontend renders the player’s hand visually.