Progetto Architetture: GameSetupHub

▼ FASE 0 - Introduzione

GameSetupHub è una piattaforma web per la condivisione, ricerca e valutazione di configurazioni di gioco. Gli obiettivi progettuali sono modularità, scalabilità, resilienza ai guasti e consistenza dei dati. Il sistema è pensato per favorire la collaborazione tra utenti e la diffusione di best practice nel setup dei videogiochi.

Descrizione del Sistema

Il sistema è organizzato secondo un'architettura a microservizi containerizzata, che garantisce separazione delle responsabilità e facilità di evoluzione.

▼ FASE 1 – Analisi dei Requisiti

1.1 Requisiti Funzionali (RF)

I requisiti funzionali descrivono ciò che il sistema **deve fare**.

RF1 – Registrazione e autenticazione

- L'utente può registrarsi e creare un profilo.
- L'utente può autenticarsi con username e password.

RF2 – Upload configurazioni

- L'utente autenticato può caricare una configurazione per un gioco specifico.
- Una configurazione comprende:
  - Testo descrittivo
  - Parametri chiave-valore (es. `resolution: 1920×1080`)
  - Tag (es. "competitive", "RTX", "modded")

    - Eventualmente, file di testo o il contenuto dei file

RF3 – Visualizzazione

- Gli utenti possono visualizzare tutte le configurazioni di un determinato gioco.
- Per ogni configurazione si vedono:
    - Descrizione
    - Parametri
    - Autore
    - Data
    - Valutazione media
    - Commenti (eventuale)

RF4 – Ricerca e filtraggio

- È possibile cercare configurazioni:
    - Per nome del gioco
    - Per tag
    - Per popolarità
    - Per data

RF5 – Valutazioni e commenti

- Gli utenti autenticati possono:
    - Mettere un “like” o un “dislike” (oppure dare da 1 a 5 stelle)
    - Commentare le configurazioni

1.2 Requisiti Non Funzionali (RNF)

I requisiti non funzionali descrivono **come** il sistema deve comportarsi.

RNF1 – Scalabilità

- Il sistema deve poter gestire da subito un centinaio di utenti contemporanei, con margine per reggere picchi più alti.

RNF2 – Prestazioni

- La ricerca e la visualizzazione devono restituire risultati in meno di 2 secondi per l'80% delle richieste.

RNF3 – Affidabilità

- I dati caricati devono essere persistenti.

RNF4 – Sicurezza

- I dati degli utenti devono essere protetti (autenticazione, hashing password).
- Le API devono essere protette da accessi non autorizzati (es. token JWT).

RNF5 – Manutenibilità e modularità

- Il sistema deve essere modulare, con componenti facilmente sostituibili o scalabili.

RNF6 – Portabilità

- Il sistema deve essere eseguibile su qualsiasi macchina con Docker installato.

RNF7 – Usabilità

- Le API devono essere ben documentate e facili da usare per un frontend o applicazioni esterne.

1.3 Attori e Casi d'Uso principali

Attore	Caso d'uso principale
Utente	Registrazione, login, visualizzazione configurazioni.
Utente autenticato	Upload configurazione, valutazione, commenti.
Sistema	Gestione dati, indicizzazione.

1.4 Assunzioni

- Ogni utente è identificato in modo univoco (username/email).
- I dati delle configurazioni sono semi-strutturati (JSON o simile).

- Le operazioni CRUD sono esposte tramite API REST.
- I servizi saranno eseguiti come container Docker.

▼ FASE 2 – Progettazione Architetturale

2.1 Architettura Generale

Adottiamo un'**architettura a microservizi**, dove ogni servizio è indipendente e containerizzato. I servizi comunicano tra loro principalmente tramite **API REST**. I sistemi di messaggistica/eventi (di futura implementazione) saranno integrati con tecnologie come **Kafka o RabbitMQ**.

2.2 Componenti Principali

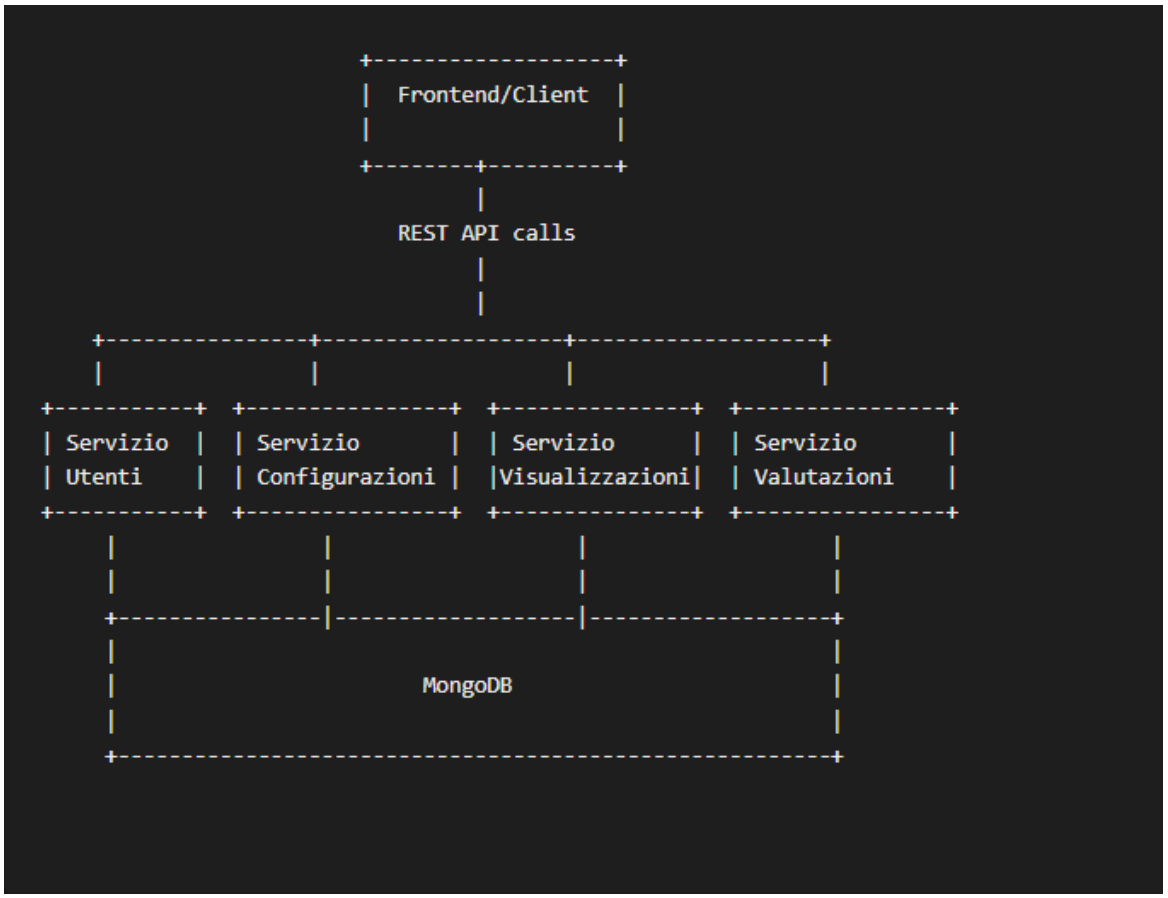
Microservizio	Responsabilità
Servizio Utenti	Gestione registrazione, login, autenticazione JWT, profili utente
Servizio Configurazioni	Gestione CRUD delle configurazioni: upload, modifica, visualizzazione
Servizio Visualizzazione	Ricerca full-text per gioco, tag, popolarità. Appoggio alla visualizzazione corretta nelle schede
Servizio Commenti e Valutazioni	Like/dislike, rating, commenti
Database	Storage persistente con DB non relazionale in MongoDB
Frontend/Client	Interfaccia web sviluppata in HTML, CSS e JS.

2.3 Scelte Tecnologiche

Scopo	Tecnologia suggerita	Motivazione
Linguaggio	Python	Per rapidità di sviluppo e librerie mature.
Framework	**FastAPI** (Python) per **API RESTful**	Leggere, performanti, facili da containerizzare
Base dati configurazioni	**MongoDB**	Supporta dati semi-strutturati, facile da scalare, adatto a JSON
Gestione utenti	**MongoDB**	Mongo per uniformità JSON
Containerizzazione	**Docker**	Standard per la portabilità

Scopo	Tecnologia suggerita	Motivazione
Message Broker (eventuale)	**RabbitMQ** o **Kafka**	Per gestione eventi (es. nuovo commento, nuova configurazione)
Autenticazione	**JWT**	Token-based authentication semplice e sicura

2.4 Diagramma dei Componenti

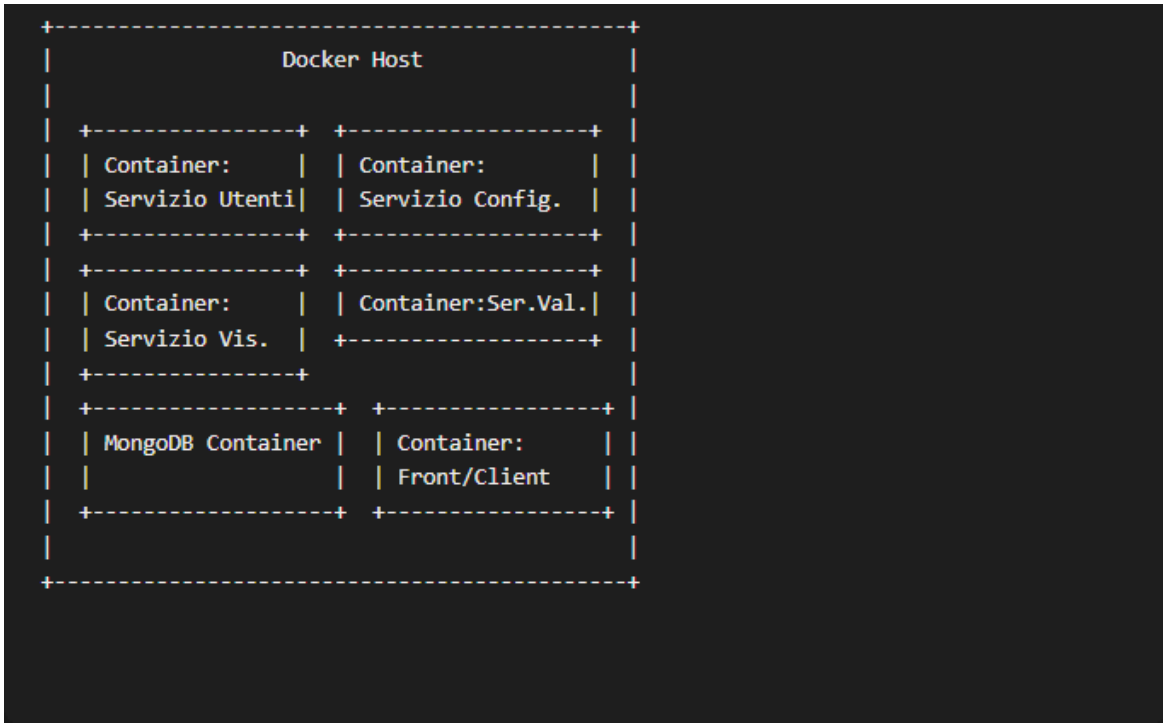


2.5 Pattern distribuiti adottati

Aspetto architetturale	Soluzione adottata
Scalabilità	I microservizi sono scalabili indipendentemente, grazie all'utilizzo di Docker Compose. MongoDB è scalabile orizzontalmente.
Bilanciamento del carico	Docker compose come orchestratore

Aspetto architetturale	Soluzione adottata
Consistenza	Eventual consistency su like/commenti; strong consistency nei dati utente e configurazioni
Tolleranza ai guasti	Retry automatici (retry : on-failure), persistenza su Mongo
Resilienza	HealthCheck [Se implementato docker swarm bisogna aggiungere roba a questa cella]
Containerizzazione	Ogni microservizio è containerizzato con Docker

2.6 Diagramma di Deployment



2.7 Diagramma di Sequenza (Caso d'uso: Creazione configurazione)

Utente → API → Servizio Utenti: verifica JWT
Servizio Utenti → API: OK
API → Servizio Configurazioni: POST /configurazione
Servizio Configurazioni → API → MongoDB: save(config)
MongoDB → Servizio Configurazioni: success
Servizio Configurazioni → API → Utente: 201 Created

2.8 Considerazioni per implementazioni future

Per supportare un elevato numero di richieste simultanee:

- **Frontend e API** possono essere replicati dietro bilanciatori (NGINX, LoadBalancer).
- **Performance:** Possibile integrazione di cache distribuita per migliorare le performance
- **Resilienza:** Timeout, circuit breaker da implementare con Docker Swarm.
- **Tolleranza ai guasti:** Replica dei servizi da implementare con Docker Swarm.
- **Consistenza dei dati:** quorum di lettura/scrittura con MongoDB replica set.
- **MongoDB** può essere scalato con **replica set** o **sharding**.
- **Scalabilità:** il sistema in futuro potrà crescere fino a supportare migliaia di utenti simultanei.

2.9 Sviluppi futuri

- Espansione verso nuove tipologie di configurazioni (aggiunta immagini)
- Integrazione di notifiche push e social features

▼ FASE 3 – Piano di Sviluppo

🎯 Obiettivo del PoC

Realizzare un sistema funzionante e containerizzato che permetta:

- la registrazione/login utente,
- l'upload e la visualizzazione di una configurazione,
- la ricerca base per nome del gioco, per tag o per popolarità,
- il like, la valutazione e il commento per una configurazione,
- la consultazione tramite API REST.

📦 Funzionalità Minime del PoC

Codice	Funzionalità	Stato
F1	Registrazione e login (JWT)	✅
F2	Upload configurazione (testo + chiavi-valori)	✅
F3	Visualizzazione configurazioni per gioco	✅
F4	Ricerca per nome gioco (match esatto o parziale)	✅
F5	Like/valutazione/commento di una configurazione	✅
F6	Containerizzazione di almeno 2 servizi	✅

⏱️ Priorità e ordine di sviluppo (sequenziale)

1. **Autenticazione utente (JWT)**
2. **Upload configurazione**
3. **Visualizzazione configurazioni**
4. **Like,valutazione e commento configurazioni**
5. **Containerizzazione Servizi**
6. **Creazione frontend**
7. **Test**

A questo punto, si è pronti per la **FASE 4 – Sviluppo del PoC**, in cui si andrà a definire il progetto base (es. struttura cartelle, esempio di API, Dockerfile) e si verrà mostrata l'implementazione dei microservizi.

▼ FASE 4 – Sviluppo del PoC

📁 Struttura del progetto PoC

```
gamesetuphub/
│
├──.vscode
├──configs_service
```

```
│   └──app
│      └──auth.py
│      └──database.py
│      └──main.py
│      └──models.py
│      └──routes.py
│      └──utils
│      └──Dockerfile
├──docs
│   ├──diagrammi
│   └──relazioni
├──frontend
│   └──js
├──infra
│  └──docker-compose.yml
├──tests
├──users_service
│   └──app
│      └──__pycache__
│      └──auth.py
│      └──database.py
│      └──main.py
│      └──models.py
│      └──routes.py
│      └──utils
│      └──Dockerfile
├──valutation_service
│   └──app
│      └──auth.py
│      └──database.py
│      └──main.py
│      └──models.py
│      └──routes.py
│      └──utils
│      └──Dockerfile
└──visualizations_service
│   └──app
│      └──auth.py
```

```
│       └─────database.py
│       └─────main.py
│       └─────models.py
│       └─────routes.py
│       └─────utils
│       └─────Dockerfile
```

Code Map esplicativa Funzione per Funzione

MAPPA DEL CODICE — GameSetupHubProject-Official

Questo documento descrive gli artefatti di codice principali del repository, organizzati per cartella e funzione. La struttura segue la suddivisione in microservizi, frontend, infrastruttura e documentazione.

Documenti top-level

- `README.md` — Panoramica del progetto e istruzioni per avvio locale.
- `CODE_MAP.md` — Mappa del codice (questo documento).
- `ComandsFile.txt` — Elenco comandi utili o script.
- `requirements.txt` — Dipendenze Python globali.
- `start_services.bat` / `start_services.sh` — Script per avvio rapido dei servizi (Windows/Unix).

Documentazione

- `docs/diagrammi/` — Diagrammi architetturali e di sequenza.
- `docs/relazioni/` — Relazione tecnica finale.

Infrastruttura

- `infra/docker-compose.yml` — Definizione dei servizi Docker per sviluppo locale (MongoDB, frontend, microservizi).

Frontend (sito statico servito da nginx)

- `frontend/index.html` — Dashboard principale.

- `frontend/configuration.html` — Dettaglio configurazione.
- `frontend/configurations.html` — Elenco configurazioni.
- `frontend/search.html` — Ricerca avanzata.
- `frontend/upload.html` — Form upload configurazione.
- `frontend/style.css` — Stili globali.
- `frontend/Dockerfile` — Image nginx per lo statico.

JS (frontend/js)

- `api.js` — Client HTTP centrale, gestione chiamate ai microservizi e autenticazione.
  - `auth.js` — Gestione login, registrazione, sessione utente.
  - `configuration-detail.js` — Logica pagina dettaglio configurazione.
  - `configurations.js` — Logica pagina elenco configurazioni.
  - `dashboard.js` — Stato servizi e configurazioni recenti.
  - `search.js` — Ricerca avanzata e filtri.
  - `upload.js` — Gestione form upload e parametri.

Descrizione funzione-per-funzione (già espanso per i file principali)

- `frontend/js/api.js` —
  - `getConfig(configId)`: recupera la configurazione base dal `configs_service`.
  - `getConfigurationDetails(configId)`: recupera la vista arricchita dal `visualizations_service`.
  - `addComment(configId, comment)`: POST per aggiungere un commento.
  - `addRating(configId, rating)`: POST per inviare una valutazione.
  - `toggleLike(configId)`: aggiunge o rimuove like.
  - `login` / `register` / `getCurrentUser`: gestione autenticazione e token.

    - helper: `formatDate`, `truncateText`, `handleResponse`, `showToast`, `showError`, `showSuccess`.
- `frontend/js/configuration-detail.js` — Metodi principali: `init`, `getConfigIdFromUrl`, `setupEventListeners`, gestione rating/commenti, caricamento dettagli, rendering, utility.
- `frontend/js/auth.js` — Metodi principali: `handleLogin`, `handleRegister`, `logout`, `updateNavbar`, `showProfile`, `isLoggedIn`, `requiresAuth`, `checkAuthRequired`, `getCurrentUser`, `getToken`, `showLoginModal`, `showRegisterModal`.
- `frontend/js/configurations.js` — Metodi principali: `init`, `setupEventListeners`, `loadConfigurations`, `applyFilters`, `updatePagination`, `renderPagination`, `renderConfigurations`, `generateConfigurationCard`, `getEmptyState`, `updateStats`, `handleFilter`, `clearFilters`.
- `frontend/js/dashboard.js` — Metodi principali: `checkServicesStatus`, `updateServiceStatus`, `loadRecentConfigurations`, `generateConfigCard`, `testServiceConnection`.
- `frontend/js/search.js` — Metodi principali: `setupEventListeners`, `loadAvailableData`, `setupAdvancedFilters`, `performSearch`, `performAdvancedSearch`, `performBasicSearch`, `applyLocalFilters`, `displayResults`.
- `frontend/js/upload.js` — Metodi principali: `init`, `setupEventListeners`, `addParameter`, `setupParameterEvents`, `updateParameterInput`, `removeParameter`, `syncParametersData`, `getParametersFromSimpleForm`, `validateJson`, `updateTagPreview`, `addTag`, `showPreview`, `getFormData`, `handleSubmit`.

Microservizi (Python / FastAPI)

Ogni microservizio segue la struttura:

- `Dockerfile` — Image per deploy.
- `requirements.txt` — Dipendenze specifiche.
- `app/` — Codice sorgente:
- `main.py` — Avvio FastAPI, mount router.

- `routes.py` — Endpoints REST.
- `models.py` — Modelli Pydantic per request/response.
- `database.py` — Connessione MongoDB e collections.
- `auth.py` — Autenticazione JWT (se presente).
- `utils.py` — Funzioni di utilità (se presente).

configs_service

- Gestione configurazioni di gioco: upload, ricerca, lettura.
- Endpoints principali: `upload_config`, `get_config`, `search_configs`.

users_service

- Gestione utenti: registrazione, login, info utente.
- Endpoints principali: `register`, `login`, `get_me`, `get_user_by_id`.

valutation_service

- Gestione commenti, likes, valutazioni.
- Endpoints principali: `add_comment`, `add_like`, `add_valutation`, `delete_comment`, `delete_rating`.

visualizations_service

- Arricchimento configurazioni con commenti, rating, likes, info autore.
- Endpoints principali: `get_configuration_details`, `get_game_configurations`, `search_configurations`.

Database & schemi (sintesi)

- Ogni servizio ha `app/database.py` che espone `MongoClient` e le collection necessarie (es: `configs_collection`, `comments_collection`, `ratings_collection`, `likes_collection`, `users_collection`).
- I modelli Pydantic si trovano in `app/models.py` per ciascun servizio e definiscono shape di request/response.

Test & automazione

- `tests/` — Test di integrazione e unitari:
- `main.py` — Partenza automatica dei test.
- `test_all_services.py` — Test end-to-end su tutti i servizi.
- `test_auth_simple.py` — Test autenticazione base.
- `test_complete_workflow.py` — Test flusso completo (register → upload → comment/like/rating → visualizations).
- `test_integrated_search.py` — Test ricerca integrata.
- `test_quick.py` — Test rapido.

▼ Conclusioni

L'obiettivo di modularità è stato raggiunto grazie alla divisione in microservizi; [la resilienza è stata verificata tramite fault recovery e replica]; la consistenza è garantita.

Tuttavia, rimanegono aperti questioni riguardo la scalabilità, tolleranza dei guasti, resilienza etc. Il progetto dimostra la fattibilità di un sistema distribuito per la gestione di configurazioni di gioco, con garanzie di consistenza, [resilienza] e disponibilità, ma suggerisce possibili evoluzioni verso architetture più scalabili e ricche dal punto di vista dell'esperienza utente.