**Contesto**

La soluzione è stata sviluppata in relazione ai requisiti funzionali e architetturali indicati, simulando una piattaforma di gestione documentale e approvazione **TrustFlow**.

**Architettura a microservizi**

Il progetto è composto da **servizi Spring Boot separati**, ognuno con responsabilità ben definite:

| **Servizio** | **Responsabilità principale** |
| --- | --- |
| auth-service | Autenticazione utente tramite username/password e SPID mock |
| document-service | Upload, gestione e modifica dei documenti con workflow |
| integration-layer | Mock di servizi esterni protetti tramite OAuth2 (firma, PEC…) |
| audit-service | Logging asincrono e persistente degli eventi applicativi |

I microservizi comunicano tra loro via REST e scambiano token JWT per propagare il contesto di autenticazione.

**Persistenza con PostgreSQL**

Le informazioni applicative strutturate (utenti, documenti, workflow, configurazioni) sono memorizzate in **PostgreSQL**.

| **Servizio** | **Database** | **Utilizzo principale** |
| --- | --- | --- |
| auth-service | authdb | Utenti, credenziali, tenant |
| document-service | documentdb (es.) | Metadata documenti, stato, tracking workflow |

Sono stati definiti script SQL per la creazione delle tabelle.

La scelta di PostgreSQL è motivata da compatibilità con JPA e per un setup locale leggero (avviato tramite Docker).

**Sicurezza e autenticazione**

* **JWT (JSON Web Token)**: autenticazione stateless tra servizi, prodotto da auth-service.
* **OAuth2 Authorization Server**: integrato integration-layer, emette il token per simulare le chiamate ai servizi esterni
* **SPID Mock**: simulazione del login federato tramite codice fiscale e redirect su pagina applicativa.
* **Filtro JWT**: implementato nei servizi.

**Documenti e workflow**

Il document-service permette:

* Caricamento di documenti (metadata in PostgreSQL)
* Visualizzazione e aggiornamento dello stato
* Avvio automatico del **workflow approvativo** (configurabile per tenant)
* Approvazione del documento da parte del singolo utente, a seconda della configurazione (SEQUENZIALE o PARALLELA)

**Audit asincrono**

Per garantire tracciabilità e decoupling:

* Gli eventi audit (es. caricamento documento) vengono inviati su **RabbitMQ** in formato JSON
* audit-service riceve i messaggi e li persiste in **MongoDB**
* Ogni log include: IP, utente, tenant, tipo evento, descrizione evento, messaggio e timestamp

Questo approccio permette scalabilità e audit centralizzato per tutti i servizi.

**Tecnologie utilizzate**

* **Maven**
* **Spring Boot 3.x**
* **Spring Security + OAuth2 Authorization Server**
* **Spring Data JPA**
* **Logback (logstash encoder)**
* **MongoDB** per l'audit log
* **PostgreSQL** per i dati transazionali (utenti, documenti, workflow)
* **RabbitMQ** per gestione del logging asincrono
* **Tools**: IntelliJ IDEA + GitHub Copilot, DBeaver, Postman, GitHub, Docker (WLS)

**Scelte progettuali**

* **Separazione logica** in servizi indipendenti, per facilitare testabilità e deployment isolato
* **Asincronia nell’audit** migliora performance e disaccoppia i componenti

**Considerazioni per scenario su larga scala**

Tenendo conto delle stime presentate (> 50 milioni di documenti, ~500 tenant, ~200 utenti/tenant, ~50gb/giorno di documenti caricati), la soluzione presenta accortezze per predisporne la scalabilità:

* **Architettura a microservizi**
* **RabbitMQ per la gestione asincrona di eventi su coda**
* **Audit su MongoDB**
* **Gestione multi-tenant su** **DB e JWT**

L’architettura sopra indicata dovrebbe essere integrata con ulteriori scelte orientate alla scalabilità, come ad es.:

* **Storage documentale su file system distribuito o oggetto (S3):** disaccoppiamento documenti dai metadati
* **Possibili shard o replica su su db PostgreSQL per crescita futura**
* **Kubernetes per scalabilità infrastrutturale del singolo microservizio**
* **Docker per containerizzazione dei servizi**