Documento di Analisi e Scelta della Soluzione di Multi-Tenancy per il progetto MS3

Indice

1 Introduzione

Nel corso del nostro progetto software, abbiamo implementato e analizzato due soluzioni distinte di multi-tenancy per la segregazione dei dati:

1. **Single Database, Separate Schemas for Tenant**
2. **Database per Tenant**

Il presente documento descrive il processo di implementazione, le scelte adottate per la gestione del tenant ID e l'isolamento dei dati, e infine motiva la decisione finale su quale approccio adottare.

---

2 Implementazione della Multi-Tenancy

2.1 Memorizzazione del Tenant ID

Per identificare il tenant a cui appartiene una determinata richiesta, abbiamo scelto di memorizzare il tenant ID all'interno di un **claim JWT (JSON Web Token)**.

Vantaggi di questa scelta:

- Il token incapsula il tenant ID in modo sicuro e compatto, poiché è cifrato e firmato.
- Mantiene l'URL pulito, evitando di esporre il tenant ID direttamente nelle richieste.
- Si integra perfettamente con il framework di sicurezza (Spring Security) che stiamo utilizzando, facilitando la gestione delle autorizzazioni.

```
public String generateToken(CustomUserDetails userDetails) {  1 usage   ralisin
    Map<String, Object> claims = new HashMap<>();
    claims.put("role", userDetails.getAuthorities().stream()  Stream<capture of
            .map(GrantedAuthority::getAuthority)  Stream<String>
            .collect(Collectors.toList()));
    claims.put("current_tenant", userDetails.getTenant().toLowerCase());

    return createToken(claims, userDetails.getUsername());
}
```

2.2 Determinazione del Tenant ID ad ogni richiesta

Il backend determina il tenant ID estraendolo dal claim del JWT ricevuto nella richiesta. Questo processo garantisce:

- **Efficienza**: l'estrazione del claim è veloce e non richiede query aggiuntive.
- **Sicurezza**: il token è firmato e non può essere alterato senza invalidarne l'autenticità.
- **Scalabilità**: il meccanismo si adatta bene a sistemi distribuiti, riducendo la dipendenza da lookup esterni.

Una volta estratto il Tenant ID dal JWT, lo salviamo in una variabile **ThreadLocal**.

2.3 Partizionamento e Isolamento dei Dati

Abbiamo implementato e analizzato due diverse strategie per garantire l'isolamento dei dati:

2.3.1 Approccio 1: Single Database, Separate Schemas for Tenant

- Ogni tenant ha un proprio schema all'interno dello stesso database fisico.
- Ogni tenant dispone di un utente database configurato con privilegi limitati sul relativo schema.
- Le query vengono eseguite nel contesto dello schema specifico del tenant.
- Gestione centralizzata delle connessioni e della manutenzione del database.

Meccanismo:

- Spring/Hibernate può passare dinamicamente da uno schema all'altro in base al tenant, determinato durante il runtime.
- Il provider delle connessioni seleziona l'utente appropriato in base all'identificatore del tenant, garantendo l'isolamento a livello di query SQL e operazioni database.

Vantaggi:

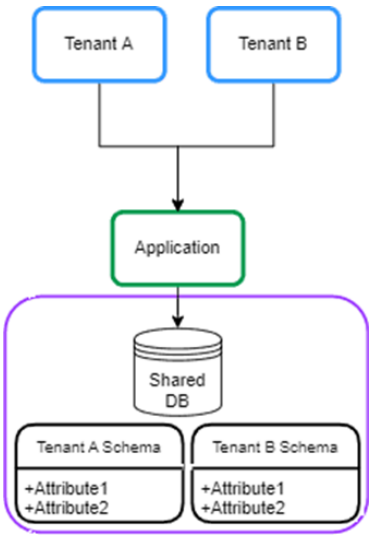
- **Minori costi infrastrutturali** rispetto alla soluzione con database separati.
- **Facilità di gestione**: tutti i dati sono contenuti in un unico database, semplificando backup e manutenzione.
- **Buon isolamento**: ogni tenant ha il proprio schema, riducendo il rischio di accessi non autorizzati ai dati di altri tenant.
- Più semplice eliminare o migrare un singolo tenant.

Svantaggi:

- La crescita del numero di tenant può portare a una gestione più complessa degli schemi.
- Il carico alto di un tenant può influenzare gli altri tenant.
- Potenziali limitazioni di scalabilità del database.

Implementazione:

- **Connessioni dinamiche personalizzate**: La classe *DataSourceConfig* crea connessioni con credenziali specifiche per ogni tenant.
- **Gestione dinamica dei tenant**: La classe *SchemaSwitchingConnectionProviderPostgreSQL* imposta il contesto SQL per isolare le query al solo schema del tenant.
- **Integrazione con Hibernate**: Hibernate utilizza un *MultiTenantConnectionProvider* per gestire le connessioni multiple e un *CurrentTenantIdentifierResolver* per risolvere dinamicamente l'identificatore del tenant; inoltre, è configurato sulla strategia "SCHEMA".
- **Migrazioni**: Manualmente per ciascuno schema.



2.3.2 Approccio 2: Database per Tenant

- Ogni tenant ha il proprio database dedicato.
- Ogni tenant dispone di un utente database configurato con privilegi limitati sul relativo database.
- Le connessioni sono gestite in base al tenant ID.
- Massima segregazione tra i dati dei vari tenant.

Meccanismo:

- Spring/Hibernate può passare dinamicamente da un database all'altro in base al tenant, determinato durante il runtime.
- Il provider delle connessioni seleziona l'utente appropriato in base all'identificatore del tenant, garantendo l'isolamento a livello di query SQL e operazioni database.

Vantaggi:

- **Massimo isolamento**: i dati di un tenant non possono essere accidentalmente esposti ad altri tenant.
- **Scalabilità**: è possibile allocare risorse specifiche per ogni tenant.
- **Flessibilità nelle configurazioni**: ogni tenant può avere ottimizzazioni specifiche a livello di database.

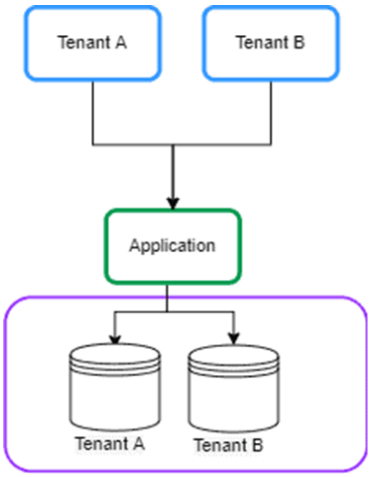
Svantaggi:

- **Costi infrastrutturali più elevati**, poiché ogni tenant ha un proprio database.
- All'aumentare del numero di tenant aumenta anche la complessità del sistema.
- **Maggiore complessità nella gestione delle connessioni**.
- **Difficoltà nella gestione di operazioni globali** (ad esempio, reporting trasversale tra tenant).

Implementazione:

- **Connessioni dinamiche personalizzate**: La classe *DataSourceConfig* crea connessioni con credenziali specifiche per ogni tenant.

- **Gestione dinamica dei tenant**: La classe *MultiTenantConnectionProviderImpl* imposta il contesto SQL per identificare il corretto database su cui operare.
- **Integrazione con Hibernate**: Hibernate utilizza un *MultiTenantConnectionProvider* per gestire le connessioni multiple e un *CurrentTenantIdentifierResolver* per risolvere dinamicamente l'identificatore del tenant; inoltre, è configurato sulla strategia “DATABASE”.
- **Migrazioni**: Manualmente per ciascun database.



---

3 Scelta della Soluzione Finale

Dopo aver analizzato entrambe le soluzioni, abbiamo scelto di adottare **l'approccio Single Database, Separate Schemas for Tenant** per i seguenti motivi:

1. **Equilibrio tra isolamento e costi**: garantisce una buona segregazione dei dati senza richiedere la gestione di più database separati.
2. **Efficienza nella gestione**: la manutenzione e le operazioni di backup sono più semplici rispetto alla gestione di un database per ogni tenant.
3. **Scalabilità accettabile**: il modello può supportare un numero elevato di tenant con un'efficace gestione degli schemi.

Questa soluzione si è rivelata adatta alle nostre esigenze attuali, garantendo sicurezza, prestazioni e un costo infrastrutturale sostenibile.

---

4 Conclusione

Abbiamo implementato una soluzione di multi-tenancy utilizzando JWT claim-based per la gestione del tenant ID e abbiamo adottato il modello **Single Database, Separate Schemas for Tenant** per la segregazione dei dati. Questa scelta ci permette di bilanciare sicurezza, scalabilità e costi operativi, mantenendo un'architettura manutenibile nel tempo.