UNIVERSITÀ DI FIRENZE

DIPARTIMENTO DI INFORMATICA

Advanced Techniques and Tools for Software Development

Report of ToDoMultiDB, a Spring Boot web application

Pietro Bernabei

ANNO ACCADEMICO 2022/2023

1 Introduzione

ToDoMultiDB, è una web application che permette di distribuire le informazioni su due database relazionali distinti. Nello specifico la web app è strutturata secondo un'architettura a layer (Repository, Service, REST Controller, e Web Controller) , la quale impiega due database MySQL indipendenti non duplicati per persistere le informazioni trasmesse da parte dell'utente. Come anticipato in precedenza il client può interagire con il backend dell'applicazione attraverso due interfacce: una grafica, attraverso una comune interfaccia HTML, e una attraverso le api REST. Le informazioni condivise sono strutturate in due semplici modelli in relazione 1 a n: User e Todo rispettivamente. Per il testing e la stessa esecuzione dell'applicazione e dell'intero contesto, è stata utilizzata la tecnologia di containerizzazione offerta da Docker.

2 Tecniche e framework:

Rispetto alle tecniche e ai framework presentati durante il corso, è stato integrato principalmente il pattern DTO e ad abilitata la gestione personalizzata delle transazione su Spring Boot. - DTO è un design pattern, che definisce un oggetto che trasporta i dati tra i processi per ridurre il numero di chiamate al metodo. Questa tecnica è stata introdotta nel progetto per risolvere un problema particolarmente tedioso riscontrato durante lo sviluppo come sarà poi esposto successivamente. - Transaction: Una transazione di database è una sequenza di azioni che vengono trattate come una singola unità di lavoro. Queste azioni devono essere completate nella loro interezza o non avere alcun effetto. La gestione delle transazioni è una parte importante delle applicazioni orientate agli RDBMS per garantire l'integrità e la coerenza dei dati. Il concetto di transazione può essere descritto con le seguenti quattro proprietà chiave, descritte come ACID:

- Atomicità - Una transazione deve essere trattata come una singola unità di operazione, il che significa che l'intera sequenza di operazioni ha successo o non ha successo.
- Consistenza - Rappresenta la consistenza dell'integrità referenziale del database, chiavi primarie uniche nelle tabelle, ecc.
- Isolamento - Possono esserci molte transazioni che elaborano lo stesso set di dati nello stesso momento. Ogni transazione deve essere isolata dalle altre per evitare la corruzione dei dati.
- Durata - Una volta completata una transazione, i risultati di questa devono essere permanenti e non possono essere cancellati dal database a causa di un guasto del sistema.

Un vero sistema di database RDBMS garantisce tutte e quattro le proprietà per ogni transazione. La visione semplicistica di una transazione inviata al database tramite SQL è la seguente.

- Iniziare la transazione con il comando begin transaction.
- Eseguire varie operazioni di cancellazione, aggiornamento o inserimento utilizzando le query SQL.
- Se tutte le operazioni hanno successo, si esegue il commit, altrimenti si esegue il rollback di tutte le operazioni.

Il framework Spring fornisce un livello astratto in cima alle diverse API di gestione delle transazioni. Spring supporta sia la gestione programmatica che quella dichiarativa delle transazioni.

Transazioni locali e globali Le transazioni locali sono specifiche di una singola risorsa transazionale, come una connessione JDBC, mentre le transazioni globali possono estendersi a più risorse transazionali, come le transazioni in un sistema distribuito.

- La gestione delle transazioni locali può essere utile in un ambiente informatico centralizzato in cui i componenti e le risorse dell'applicazione si trovano in un unico sito e la gestione delle transazioni coinvolge solo un gestore di dati locale in esecuzione su un'unica macchina. Le transazioni locali sono più facili da implementare.

- La gestione globale delle transazioni è necessaria in un ambiente informatico distribuito, dove tutte le risorse sono distribuite su più sistemi. In questo caso, la gestione delle transazioni deve essere effettuata sia a livello locale che globale. Una transazione distribuita o globale viene eseguita su più sistemi e la sua esecuzione richiede il coordinamento tra il sistema di gestione delle transazioni globali e tutti i gestori di dati locali di tutti i sistemi coinvolti.

Programmatico vs. dichiarativo
Spring supporta due tipi di gestione delle transazioni:

- Gestione programmatica delle transazioni: ovvero la transazione deve essere gestita con l'aiuto della programmazione. Ciò offre un'estrema flessibilità, ma risulta di difficile manutenzione.
- Gestione dichiarativa delle transazioni: la gestione delle transazioni è separata dal codice attraverso l'uso di annotazioni o configurazioni basate su XML. Come si vedrà in seguito, in questo progetto, per la modellazione del comportamento delle transazioni sulle capacità dell'applicazione è stato scelto l'uso dell'approccio dichiarativo nella sua forma di annotazioni.

3 Design e implementation choices:

La web app è stata sviluppata come specificato nell'introduzione secondo una struttura a strati. Di seguito sarà proposta una descrizione di ciascuno strato partendo dal Repository layer al WebController layer

3.1 Repository layer

Lo sviluppo di questo layer è stato principalmente effettuato attraverso il motore di generazione di query integrato nella piattaforma. Anche se queste funzionalità sono generate autonomamente dal framework Spring, sono stati comunque sviluppati i test (Unit e Integration Test) delle varie funzionalità al fine di capirne e testare le funzionalità.

3.1.1 Database Connection

In comunione al repository layer sono stati implementati i meccanismi di connessione ai database MySQL attraverso interfacce JDBC e il relativo meccanismo di cambio della connessione. Secondo l'introduzione iniziale sulle transazioni, è stata sviluppata una gestione delle transazioni di tipo locale, gestendo direttamente ciascuna connessione. Nello specifico, questa gestione delle connessione verte attorno all'implementazioni di tre classi:

- DataSourceContextHolder,
- DataSourceRoutingConfiguration,
- DataSourceOneConfig/DataSourceTwoConfig

DataSourceContextHolder

```
1  @Component
2  public class DataSourceContextHolder {
3
4      private static ThreadLocal<DataSourceEnum> context = new ThreadLocal<>();
5      ...
6      public void set(DataSourceEnum data) {
7          context.set(data);
8      }
9      ...
10     public DataSourceEnum setDatabase(int ctx) {
11         if (ctx <= 1)
12             this.set(DataSourceEnum.values()[0]);
13         else
14             this.set(DataSourceEnum.values()[(ctx - 1) % 2]);
15         return this.getDataSource();
```

```
    }
}
```

DataSourceContextHolder è una classe di particolare interesse siccome ha lo scopo di mantenere e gestire le connessioni instanziate nella struttura ThreadLocal¡DataSourceEnum¿. Attraverso quest'ultima struttura dati è possibile associare a un thread un risorsa solo a lui accessibile, come l'identificativo del database di destinazione. La gestione indipendente a livello di thread della risorsa DataSourceEnum permette di associare a una transazione la destinazione dei database. Essendo una transazione incapsulata in un thread quando viene inizializzata, è necessario associargli determinate informazioni immutabili esternamente allo stesso. Per quanto riguarda i test di questa classe, questi non risultano essere di particolare interesse non essendo stato necessario implementare una logica particolarmente complessa.

DataSourceRouter DataSourceContextHolder fornisce un interfaccia per la manipolazione dell'identificativo del database di destinazione, attraverso il settaggio di un particolare DataSourceEnum. In questa sezione invece sarà descritta la classe che registra il comportamento del DataSourceContextHolder.

```
public class DataSourceRouter extends AbstractRoutingDataSource {

    @Autowired
    DataSourceContextHolder dataContext;

    @Override
    protected Object determineCurrentLookupKey() {
        return dataContext.getDataSource();
    }

}
```

Con l'estensione della classe AbstractRoutingDataSource, viene permesso di descrivere una metodologia personalizzata nel determinare la destinazione delle comunicazioni. Principalmente questa classe permettere di determinare a runtime la scelta dell'utente. **DataSourceRoutingConfiguration** Prima di proseguire con la discussione dell'implementazione della classe è necessario fornire una breve descrizione delle annotazioni associate:

- Configuration: si specifica a SpringBoot che la seguente classe è una configurazione
- Profile: La seguente configurazione deve essere utilizzata quando non viene eseguito l'ambiente con il profilo repository abilitato. L'uso di questa annotazione è stata necessaria durante lo sviluppo dei test di integrazione con le classi repository.
- ComponentScan: per specificare all'applicazione da dove iniziare a ricercare le classi annotate come component o derivate. Questa annotazione è stata associata alla classe siccome, il sistema inizia la scansione delle componenti dalla directory di package in poi. Essendo la seguente classe in .routing.config e alcune componenti di interesse in .routing quest'utlime non sono indicizzate di default.
- EnableTransactionManagement: abilita l'inizializzazione e l'esecuzione personalizzate delle transazioni definite con la notazione @Transaction associate a metodi e classi.

```
package com.example.todoappmultidb.routing.config;

@Configuration
@Profile("!repository")
@ComponentScan("com.example.todoappmultidb")
@EnableTransactionManagement
public class DataSourceRoutingConfiguration {

    private static final String SCHEMA_MYSQL_SQL = "schema-mysql.sql";
    @Autowired
```

```
    private DataSourceOneConfig dataSourceOne;
    @Autowired
    private DataSourceTwoConfig dataSourceTwo;
    private ResourceDatabasePopulator resourceDatabasePopulator = new ResourceDatabasePopulator();
    private DataSourceRouter dataRouter = new DataSourceRouter();
...
     @Bean
    public DataSource dataSource() {
        dataRouter.setTargetDataSources(genereta TargetDataSources());
        dataRouter.setDefaultTargetDataSource(dataSourceOne.getDataSource());
        return dataRouter;
    }
```

Attraverso il metodo dataSource() sono impostate le connessioni ai database, di cui una di default. Di seguito invece viene mostrato il metodo sviluppato per l'inizializzazione della connessione e del relativo Database attraverso un istanza della classe ResourceDatabasePopulator con il relativo schema mysql. Quest'ultimo è frutto del processo di popolamento con i modelli Java da parte di Hibernate.

```
 ...
 @Bean
public DataSourceInitializer dataSourceOneInitializer() {

    DataSourceInitializer dataSourceInitializer = new DataSourceInitializer();
    resourceDatabasePopulator.addScript(new ClassPathResource(SCHEMA_MYSQL_SQL));
    dataSourceInitializer.setDatabasePopulator(resourceDatabasePopulator);
    dataSourceInitializer.setDataSource(dataSourceOne.getDataSource());
    return dataSourceInitializer;

}
}
```

Rimanendo nel ambito delle configurazione delle connessioni ai database, le informazioni necessarie all'instauramento delle connessioni attraverso JDBC sono state strutturate in due tipologie di classi:

DataSourceConfig

```
 public class DataSourceConfig {

    private String url;
    private String password;
    private String username;

    ... setter e getter generici...

    public DataSource getDataSource() {
        return new DriverManagerDataSource(url, username, password);
        }
```

Classe di metodi getter e setter tranne per l'ultimo che come da documentazione: *Semplice implementazione dell'interfaccia standard JDBC DataSource, che configura il vecchio JDBC DriverManager tramite le proprietà del bean e restituisce una nuova connessione a ogni chiamata getConnection.*

DataSourceOneConfig

```
@Component
@ConfigurationProperties("datasourceone.datasource")
public class DataSourceOneConfig extends DataSourceConfig {
    public DataSourceOneConfig() {
        super();
        }
}
```

Semplice classe che estende la precedente, ma con la particolarità di essere inizializzata attraverso le proprietà definite nel file application.properties presente nella cartella src/main/resources. Siccome gli

Unit Test richiedono che una classe sia testata in un ambiente isolato da tutte le interazione con classi esterne, è stato necessario identificare una modalità di inizializzazione della classe senza un riferimento esplicito all'application.properties. La soluzione individuata è documentata nel prossimo snippet, nel quale si può notare l'uso di due diverse annotazioni:

- EnableConfigurationProperties: che permette di definire quale classe annotata con Configuration abilitare per questo test. Annotazione particolarmente utile per rispettare il test slicing dell'unit test.
- TestPropertySource: la quale permette di instanziare la Configuration Class con i dati di testing, garantendo così il disaccoppiamento dell'ambiente del test rispetto alle reali configurazioni.

```
@RunWith(SpringJUnit4ClassRunner.class)
@EnableConfigurationProperties(value = DataSourceOneConfig.class)
@TestPropertySource("classpath:server.properties")
public class DataSourceOneConfigTest {
    @Autowired
    private DataSourceOneConfig data;
    @Test
    public void datasourceOneConfigBindingValue_test() {
        assertThat(data.getUrl()).isEqualTo("localhost");
        assertThat(data.getPassword()).isEqualTo("password");
        assertThat(data.getUsername()).isEqualTo("root");
        }
```

3.2 Service Layer

Siccome l'applicazione si interfaccia con due database, tra cui deve smistare le informazioni inoltrate dall'utente, è necessaria una gestione più minuziosa delle trasmissioni. Per questo motivo a livello di Service è stato sviluppato il sistema di transazioni, dove: le attività di lettura sui database sono incapsulate in transazioni con le sole capacità di lettura, mentre quelle che svolgono un attività di scrittura prevedono sia la capacità di scrivere sia quella di essere incapsulate in una nuova transazione. Prima di descrivere le due differenze stilistiche scelte con gli esempi successivi, è importante fornire una breve descrizione del comportamento della annotazione Transactional. Il transactional manager, se abilitato (già mostrato in precedenza), permette l'incapsulamento dell'esecuzione dei metodi in transazioni di cui è possibile definirne comportamenti specifici. Annotando un metodo con l'annotazione @Transactional, identifichiamo al transaction manager la necessità di incapsulare in una transazione tutto il flusso di esecuzione che scaturisce dal metodo. Più precisamente al momento è possibile identificare se un'esecuzione debba necessitare una nuova transazione o possa essere accoppiata ad un altra se già esistente. In questo progetto sono state utilizzate solamente le due precedenti configurazioni, ma ne esistono altre con comportamento simile. Di seguito è stato inserito una delle tante implementazioni della logica descritta in precedenza. Possiamo notare infatti che la funzione getAllUser è annotata con una notazione Transactional semplice. La dicitura rollbackFor permette di ripristinare lo stato di esecuzione a uno stato precedente alla esecuzione della transazione nel momento in cui si scaturisca una eccezione. Il comportamento di default dell'annotazione Transactional prevede che le transazioni abbiano la sola capacità di leggere, ottimizzandone l'esecuzione per questo scopo, e che sia incapsulata in una transazione nuova se non ne è già presente una (propagation.REQUIRED). Caso diverso invece è insertNewUser, per la quale è stata disabilità l'opzione di sola lettura e imposto l'esecuzione della scrittura in una transazione nuova. Attraverso questa opzione si impedisce di eseguire due attività di scrittura all'interno della stessa transazione limitando possibili conflitti.

```
@Service
public class UserService {
...
    @Transactional(rollbackFor = NotFoundException.class)
    public List<UserDTO> getAllUser() throws NotFoundException {

        List<User> userFound = userRepository.findAll();
        if (userFound.isEmpty())
```

```
9           throw new NotFoundException("Not found any User");
10          return userFound.stream().map(this::toDTO).collect(Collectors.toList());
11  }
12  ...
13      @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW,
14       rollbackFor = IllegalArgumentException.class)
15      public UserDTO insertNewUser(UserDTO userToSave) {
16          userToSave.setId(null);
17          verifyNullField(userToSave);
18          return toDTO(userRepository.save(new User(userToSave)));
19      }
20  ...
21  }
22
```

Sempre per rimanere in tema transazioni, UserService è stata predisposta per svolgere un attività di interfaccia nei confronti del DataSourceContextHolder, la classe che implementa effettivamente l'interfaccia alla logica del cambio della database di destinazione. Alla base di questa scelta ci sono stati i seguenti motivi ovvero: una sua implementazione a un livello inferiore come repository, avrebbe richiesto di spostare l'intero sistema di creazione delle transazioni a questo livello, sovraccaricando lo stesso livello con un ulteriore compito. Se così non fosse (a livello di service sono create le transazioni e a livello di repository viene impostato il database), si generebbe un effetto indesiderato riscontrato durante lo sviluppo. Durante l'inizializzazione di una transazione, viene generato un thread che gestisce l'intero flusso delle esecuzioni dalla loro chiamata fino al database, al quale vengono associate le configurazioni delle connessioni in maniera immutabile. In questo frangente quando viene eseguita la logica del layer repository all'interno di una transazione il cambio del database di destinazione non può essere eseguito siccome andrebbe in contrapposizione alle proprietà ACID.**Questo particolare meccanismo è stato riscontrato durante la fase di integration testing tra Service e Repository, siccome fu prima annotato erroneamente con Transactional il metodo setDatabase della classe UserService, incapsulandolo in una transazione prima di poter impostare la destinazione desiderata, poi all'interno degl'altri metodi transactional i quali generarono lo stesso effetto. Per ottemperare a questa mal configurazione è necessario impostare la connessione desiderata prima di creare la transazione, siccome una volta che quest'ultima viene instanziata è immutabile.** La gestione di questo meccanismo a un livello superiore come REST o WEB Controller avrebbe richiesto di duplicare le capacità per ciascuno o avrebbe accoppiato due livelli distanti. Un'altra funzionalità associata al Service Layer è quello di tradurre da Model (o DAO) a DTO. Questa scelta è motivata dalla necessità di non duplicare la stessa funzionalità di traduzione sui due Controller Layer, e di non inserirlo nel Repository Layer data la sua natura.

3.3 Model

Prima di proseguire con gli ultimi livelli, è necessario soffermarci nella descrizione del modello dei dati implementati. La web app gestisce due diversi modelli: User e ToDo. Mentre il primo implementa due semplici campi: nome e email di tipo String (il campo email non implementa nessun tipo di controllo sull'effettiva formattazione del valore immagazzinato per semplicità), il secondo implementa due attributi diversi: un LocalDateTime e un Map con chiave di tipo String e con Boolean come valore associato. Oltre a questi semplici campi, User e ToDo sono in relazione tra loro attraverso una relazione di 1 a n. Nello specifico la loro relazione è stata configurata con la rimozione degli orfani, ovvero viene cancellata dal database l'intera collezione di ToDo, al momento della rimozione di un User. Di seguito la configurazione delle annotazioni associate ai due campi che permettono di collegare le rispettive classi in una relazione 1 a m di tipo unidirezionale. La differenza tra una relazione di tipo unidirezionale e bidirezionale, è dovuta alla configurazione degli owning side. La definizione di un owning side, permette di identificare i riferimenti sulle operazioni di update nelle relazione bidirezionali sui database. Soffermandoci sulle annotazioni della classe ToDo, oltre all'annotazione @ManyToOne che è autoesplicativa, l'annotazione successiva identifica la colonna per un associazione di entità o di collezioni di elementi. Nello specifico questa annotazione permette di identificare il campo come una foreign key, con il nome di idUser, non annullabile. Quest'ultimo valore, fa si che all'inserimento di

una nuova istanza nel database, sia generato un eccezione quando il campo idOfUser sia un riferimento a un user non presente nel sistema.

```
@Entity
public class ToDo {
...
     @ManyToOne
     @JoinColumn(name = "idUser", nullable = false)
     private User idOfUser;
...
```

Lato User, la relazione è definita attraverso le annotazioni associate a list ToDo. Per quanto riguarda l'annotazione OneToMany, l'assegnazione del valore a mappedBy, identifica nella classe ToDo, l'ownening side, mentre l'abilitazione dell'opzione orphanRemoval, permette di eliminare tutte le entry ToDo dal database, quando il loro riferimento al User viene posto a null o eliminato lo stesso riferimento. Quando un User viene caricato dal database, i suoi campi vengono inizializzati. L'ultima annotazione identifica la strategia con la quale vengono caricati gli oggetti della relazione dal database. La potenza di Java persistence permette di definire la modalità di loading di tutti quegli oggetti collegati all'istanza richiesta. Esistono diverse strategie tra cui JOIN e SELECT. Hibernate con la prima strategia, carica dal database tutte le istanze richieste attraverso un unica query (eagle), mentre con la seconda opzione viene impostata una strategia lazy, ovvero gli elementi collegati sono caricati al momento di una loro lettura, eseguendo così per ciascuno una query separata.

```
@Entity
public class User {
...
    @OneToMany(mappedBy = "idOfUser", orphanRemoval = true)
    @Fetch(value = FetchMode.JOIN)
    private List<ToDo> todo;
...
```

A differenza della precedente relazione, che ha visto coinvolti due modelli per i quale è stato necessario configurare entrambi i lati, di seguito viene mostrato invece un esempio di relazione tra una classe e una struttura dati, di cui si definisce la relazione attraverso le annotazioni su un solo un lato.

```
 @ElementCollection
 @CollectionTable(name = "todo_mapping", joinColumns = {
 @JoinColumn(name = "todo_map_id", referencedColumnName = "id") })
 @MapKeyColumn(name = "todo_action")
 @Column(name = "doit")
 @Fetch(value = FetchMode.JOIN)
 private Map<String, Boolean> actionList;
```

Brevemente:

- ElementCollection: è una annotazione JPA, che identifica l'attributo associato come un insieme di elementi Embeddable o valore Basic.
- CollectionTable: permette di specificare la tabella che gestisce la collezione sul database.
- MapKeyColumn e Column definisco i nomi delle colonne all'interno della tabella rispettivamente delle chiave e dei valori della mappa.

3.4 REST Controller e Web Controller

Data questa natura binaria dei modelli dell'applicazione, per mantenere una separazione dei ruoli e delle competenze in ogni layer sono stati sviluppati una componente per ciascun modello il più indipendente dal altro. Questa scelta è stata seguita dal livello repository fino al Web e REST Controller. Ciascun delle interfacce esposte da questi ultimi sono in generale strutturate come di seguente:

- Setting del database sul quale operare attraverso l'interfaccia di UserService.
- Esecuzione dell'operazione richiesta, attraverso i metodi offerti dal Service Layer relativo.

- Invio all'utente i risultati prodotti dall'esecuzione.

Durante lo sviluppo una delle problematiche riscontrate è la manipolazione dei modelli in relazione tra loro. Un esempio di questo problema, può essere descritto semplicemente prendendo in esame l'esecuzione del metodo toString su un oggetto. Il metodo toString una volta chiamato, esegue su ogni attributo dell'oggetto la funzione toString. Prendendo in considerazione la struttura dei modelli, nel momento in cui su un oggetto di tipo User viene richiamato il metodo toString, quest'ultimo concatenerebbe tutti i risultati dei metodi toString come descritto in precedenza. Brevemente, all'esecuzione di toString su un oggetto ToDo questo richiamerà a sua volta il metodo toString sul suo attributo User. Come si può notare è facile incappare in un esecuzione senza termine quando non viene implementata una gestione oculata dei flussi ricorsivi. Detto questo la soluzione implementata è risultata essere la limitazione della profondità delle chiamate. Questa soluzione però non risulta applicabile a quegli oggetti che vengono inviati ai client attraverso i Controller, per i quali è difficile limitare l'effetto indesiderato, se non riscrivendo i converter da Oggetto a JSON. Cercando di trovare una soluzione che evitasse la scrittura di un converter, la soluzione è giunta modificando la struttura delle informazioni inviate al utente, passando all'utilizzo dei DTO, Data Transfer Object. I DTO sono rappresentazioni di uno o più modelli o anche di semplici informazioni, in strutture più semplici studiate appositamente per essere inviate a terzi. Con questa rappresentazione è stato possibile eliminare la struttura ciclica dei modelli, mantenendo comunque i riferimenti del loro rapporto. Quindi sia i REST che i web Controller comunicano con il client attraverso i DTO, come anticipato nel sezione Service. In realtà durante lo sviluppo della applicazione è stato necessario riscrivere un converter per la conversione da String a Map di stringhe e booleani. Questa necessità è dovuta alla mancanza dell'implementazione del pacchetto Jackson utilizzato da Spring Boot del converter, che non era in grado di effettuare questa operazione. Cito anche che la stessa soluzione è stata adottate per la conversione da stringa a LocalDate. Una volta testati come ogni semplice funzione, i converter sono stati registrati nella classe WebConfig. Riprendendo la descrizione della gestione della scelta del database a livello Controller, il REST Controller gestisce la scelta del database su cui operare attraverso la codifica della destinazione all'interno del URL mentre il Web Controller, ne prende coscienza come parametro di percorso. Mentre per il REST Controller questa scelta non ha influito particolarmente, per quanto riguarda il WebController, la scelta di passarlo come parametro, ha permesso attraverso Thymeleaf di gestire la scelta del database direttamente nella pagina HTML senza passarlo come valore effettivo. Un esempio è mostrato di seguito:

3.4.1 REST Controller

Di seguito le API del REST controller sviluppato con i formati degli input associati.

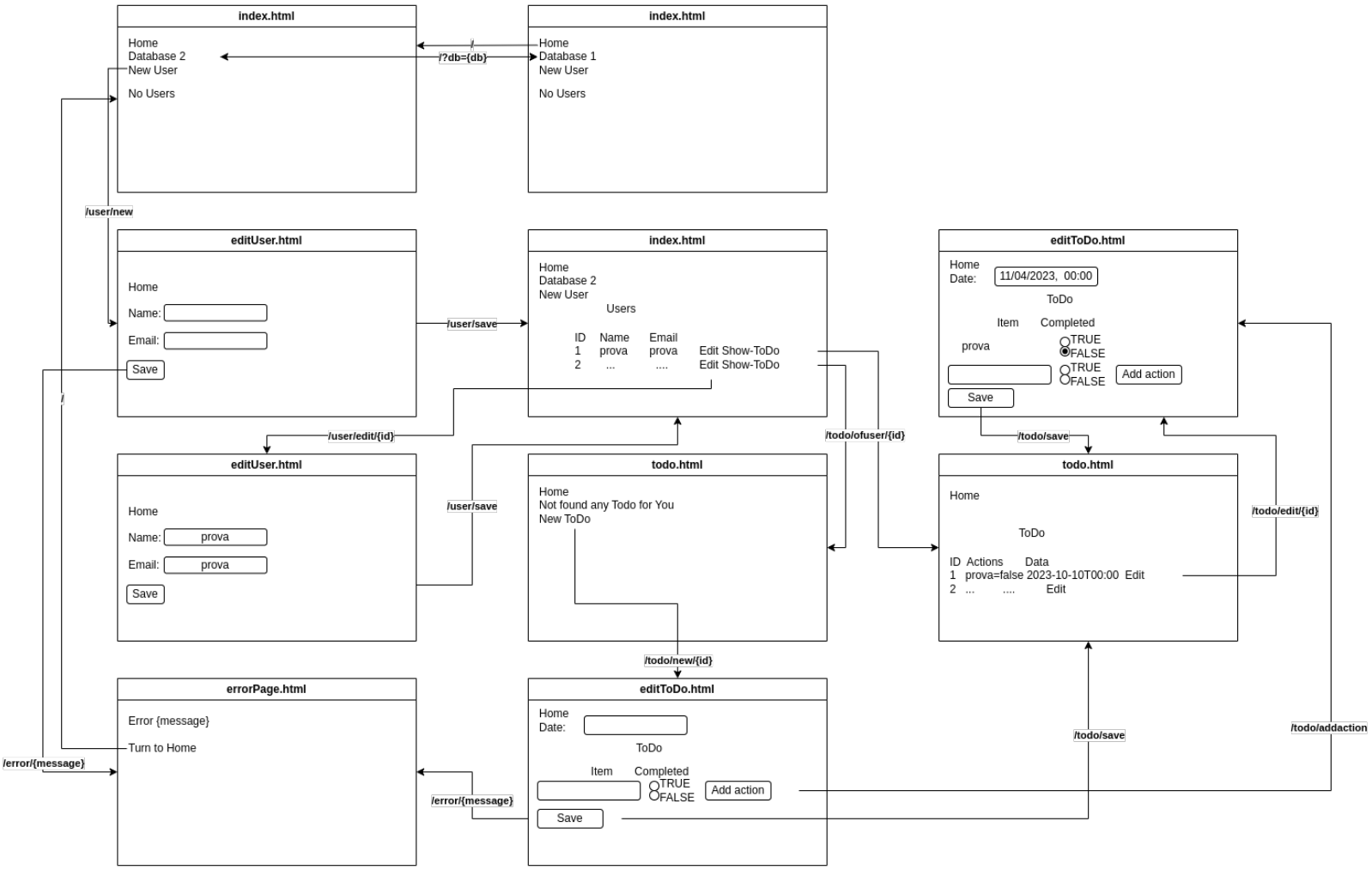
```
Formato User per il corpo della richiesta REST:
{
 "id": "2",
 "name": "prova",
 "email": "prova"
}
GET
 /api/users/{db} - Ottieni tutti gli user nel db {db}
 /api/users/{db}/id/{id} - Ottieni un singolo user con id {id} nel db {db}
PUT
 /api/users/{db}/update/{id} - Sovrascrivi i campi del user con id {id}
POST
 /api/users/{db}/new - Inserisci un nuovo user.
```

```
Formato ToDo per il corpo della richiesta REST:
{
 "id": "1",
 "actions": {
 "prova": true,
 {chiave}: {boolean}
 },
 "date": "2023-10-10T00:00:00",
 "idOfUser": "1"
```

```
}
GET
 /api/todo/{db} - Ottieni tutti i todo nel db {db}
 /api/todo/{db}/id/{id} - Ottieni un singolo todo con id {id} nel db {db}
 /api/todo/{db}/ofuser/{id} - Ottieni tutti i todo del user con id {id} nel db {db}
PUT
 /api/todo/{db}/update/{id} - Sovrascrivi i campi del todo con id {id}
POST
 /api/todo/{db}/new - Inserisci un nuovo todo.
```

3.4.2 Web Controller

Di seguito invece viene mostrata una mappa dell'applicazione lato web con tutte le varie pagine html implicate e i relativi indirizzi associati.



Alcune note:

- Come sarà presentato nella sezione successiva, è presente una pagina html, errorPage.html, che è stata predisposta attraverso un Controller dedicato alla gestione di specifici eccezzioni, mostrando a video la motivazione dell'errore.
- Come da illustrazione la scelta del database è presente solo nella pagina index.html, questo semplicemente per motivi di design e di usabilità del sito. Una scelta diversa, come quella di permettere all'utente in qualsiasi momento della navigazione la modifica del database, avrebbe richiesto all'utente o di riniziare la navigazione dalla prima pagina, scegliendo o inserendo un

utente, o di inserire l'id dello stesso. cosa simile nel caso in cui si fosse nella sezione riferita ai todo. Su questo presupposto niente impedisce all'utente di specificare alla fine di un URL il parametro "?db=" con il relativo codice del database (1 o 2), per cambiare destinazione.

3.5 Errori

Durante lo sviluppo al progetto, è stata presa in considerazione anche se in forma molto minima la gestione delle eccezioni su entrambe le interfacce. Principalmente le eccezioni generate, nascono dal Service Layer dato il suo stesso compito di gestire le transazioni con i database. Queste una volta generate, sono inviate al relativo layer superiore che ha richiamato il servizio. Il controller inerente all'operazioni avrà lo scopo di incapsulare debitamente l'informazione generate e condividerla con il Client. Per quanto riguarda l'interfacce REST, i messaggi e i relativi codice di errore non sono comunicati al client ma semplicemente salvati nei file di log, così da disposizione di default dello stesso framework per ragioni di sicurezza. Questo al fine di condividere meno informazioni possibili sensibili in caso di problemi. Lato Web invece la situazione è gestita in maniera diversa. A seconda della tipologia dell'eccezione viene gestita in maniera diversa: alcuni sono semplicemente integrati all'interno delle pagine "normali" come messaggi di stato, per esempio l'assenza di record in un database, altri invece sono gestiti da una pagina e da un controller dedicati. Il controller in questione è ErrorWebController, il quale semplicemente risponde a una chiamata GET con la relativa pagina errorPage.html inserendoci un messaggio, personalizzato o generico se non presente altri. A nota di quanto descritto sopra, per abilitare la gestione personalizzata degli errori lato Web è necessario semplicemente un controller dedicato al compito che implementi l'interfaccia ErrorController e che sia abilitata la seguente proprietà nel file application.properties.

```
server.error.path=/error
```

4 Come eseguire il programma

Download dell'applicazione:

```
HTTPS:
 git clone https://github.com/BernabeiPietro/todomultidb.git
SSH:
 git clone git@github.com:BernabeiPietro/todomultidb.git

 cd todomultidb/ToDoApplicationMultiDB/
```

Esecuzione test:

```
Unit e IT Test:
 mvn clear verify
E2E Test:
 mvn clean verify -PE2E-test
```

Esecuzione dell'applicazione con docker-compose:

```
 mvn clean package -Plocal-prod
 docker-compose build
 docker-compose up
```

Esecuzione distribuita:

```
- Inizializzazione dei due database MySQL.
- Creazione del JAR con le informazioni sui due database:
 mvn clean package -Ddb.docker.dns1=DB1_DNS_NAME -Ddb.docker.dns2=DB2_DNS_NAME -Ddb.docker.port1=DB1_port -Ddb.d
```