Università degli Studi di Milano Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

Processo e Sviluppo del Software

Assigment 3: Software Development

Autore:
Michele Salanti 793091
Ivan Donati 781022

Anno Accademico 2019–2020

Indice

Capitolo 1

Introduzione

1.1 Logo

(a) Logo dell'applicazione

1.2 Autori

Salanti Michele - 793091

Ivan Donati - 781022

1.3 Repository

Come da consegna per l'assignment è stato fatto uso dello strumento di file versioning `git` gestito da *GitLab*. Tutto il materiale finale del progetto è presente nel branch master del repository raggiungibile al seguente collegamento:

`https://gitlab.com/meliurwen/2019_assignment3_MiVan`

La radice del branch raffigurata in basso presenta un'organizzazione semplice e minimale:

- **mivan:** È la cartella che contiene il sorgente dell'applicazione.
- **doc:** È la cartella che contiene il sorgente della documentazione scritta in LaTeX.
- **assignment3_mivan.pdf:** È la versione compilata in pdf della documentazione.
- **Dockerfile** e **docker-compose.yml:** Sono i file necessari per eseguire la versione containerizzata dell'applicazione.
- **LICENSE** e **README.md:** Sono rispettivamente la licenza (MIT) assegnata per il progeto ed una sua breve introduzione.

```
<repository>
├─mivan/
├─doc/
├─.gitignore
├─Dockerfile
├─docker-compose.yml
├─LICENSE
├─README.md
└─assignment3_mivan.pdf
```

1.4 Applicazione

L'applicazione oggetto di questo assigment è **MiVan**.

Si tratta di un'applicazione **back-end only** che *gestisce prestiti di libri di un sistema bibliotecario* con *una o più sedi*. Essa, oltre ad essere in grado di gestire i prestiti, possiede la capacità di gestire i libri, la loro *posizione*, lo *staff* che ne amministra i *prestiti* e gli *utenti* che ne fanno richiesta.

Grazie a questa applicazione è possibile *creare, visualizzzare, modificare e rimuovere* in tempo reale lo stato di cessione dei libri. I prestiti sono descritti da una *data di inizio*, *di fine*, uno *stato*, un *libro*, un *utente* ed un *operatore*. Si ritiene che sia importante sottolineare che durante la progettazione si è presa la decisione che un prestito debba consistere in esattamente una unità di libro, in maniera tale che questo livello di granularità permetta all'utente, in caso di prestito "contemporaneo" (in realtà avviene in rapida sequenza dal punto di vista del backend) di più libri, di non doverli restituire tutti in blocco.

Altra caratteristica è la *separazione* tra *"concetto di libro"* ed *"unità di libro"*, questo per gestire in maniera efficiente il caso molto frequente in cui il sistema bibliotecario possieda più copie dello stesso libro; nella nostra implementazione ogni singola unità (descritta come *"Item"*) corrisponderebbe in maniera univoca all'unità fisica corrispondente. In altre parole, se il sistema bibliotecario è in possesso di n libri identici (stesso ISBN) ognuno di essi è identificabile univocamente.

Una feature degna di nota (che andrebbe a soddisfare il requisito del *self-loop*) è la possibilità di sapere se nel sistema bibliotecario è presente il *prequel* (ammesso che esista) di un determinato libro assieme al suos tato di disponibilità.

Allo stato attuale l'applicazione è stata pensata per essere utilizzata nell'area metropolitana di *Brescia* e *Novara*, con un ***target iniziale*** ristretto agli utenti delle ***sedi del sistema blibliotecario comunale*** delle rispettive città.

A seconda della trazione che potrebbe ricevere una volta lanciata, si potrà valutare un'eventuale *espansione* del territorio coperto e degli enti (sia pubblici che privati) interessati.

Capitolo 2

Implementazione

2.1 Progettazione

Nella fase iniziale di progettazione, invece che iniziare direttamente con la stesura di un *diagramma UML delle Classi* si è ritenuto più comodo sviluppare prima l'idea su carta disegnando un semplice *diagramma ER* (Entity Relationship), mostrato in figura 2.1.

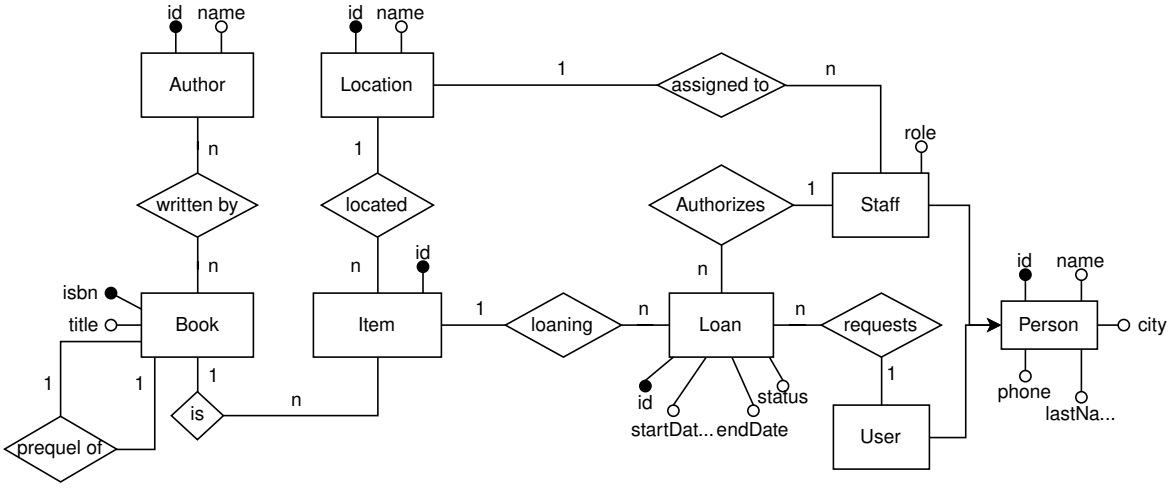

Figura 2.1: Diagramma ER

Una volta stesa una bozza definitva e chiara sulle entità, relazioni e relativi attributi da definire si è passati a trasporre in una forma più dettagliata e *più comoda* per noi da tenere come riferimento, ossia un *EER* (*Enhaced Entity Relationship*) disegnato con il tool *MySQL Workbench*.

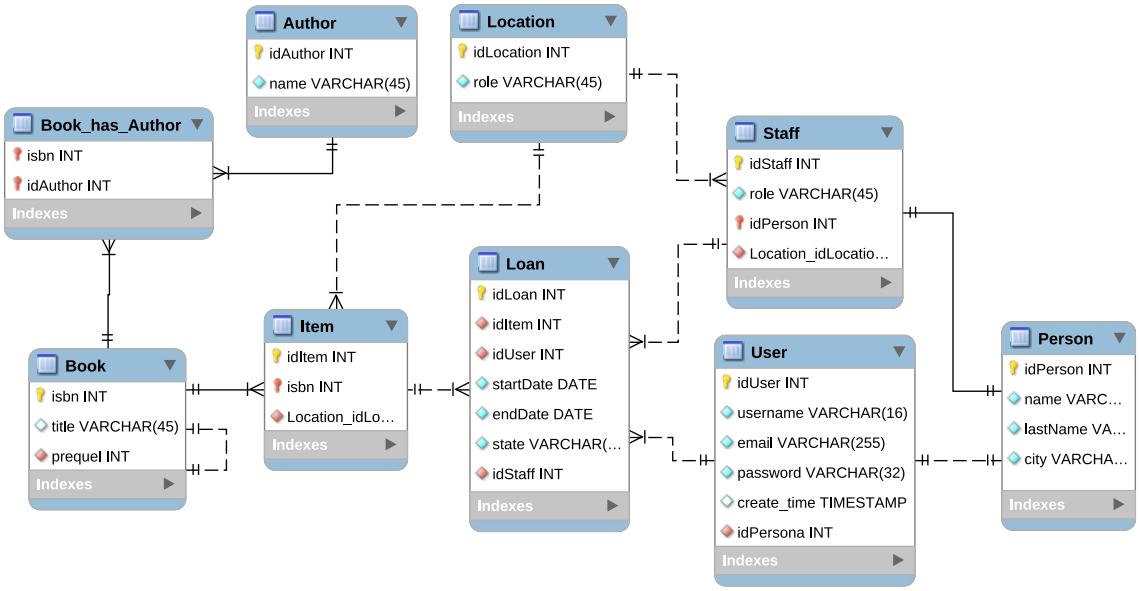

Figura 2.2: Diagramma EER

2.2 Struttura

L'applicazione è strutturata in diversi *package* con ognuno funzionalità specifiche:

- **com.mivan.model:** In questo package sono presenti tutte le entità del modello dati dell'applicazione, implementate in classi *Author*, *Book*, *Location*, *Staff*, *User*, *Loan*, *Item* e *Item*. Per gestire

la persistenza dei dati di un database relazionere, per tali classi sono state utilizzate le annotazioni delle *JPA* [1] (*Java Persistence API*).

- **com.mivan.repository:** In questo package sono implementate le query per l'interrogazione al database.

2.3 Test

Per verificare l'effettivo funzionamento del programma sono stati sviluppati dei test d'integrazione, posizionati all'interno della cartella dell'applicazione `mivan/src/test/java/mivan/`.
Le classi presenti nella cartella sono in seguito elencate:

- **AuthorTest:** Consente di testare l'*inserimento*, la *modifica* e l'*eliminazione* (implicitamente anche *lettura* nei primi due test) degli *autori*.
- **BookTest:** Consente di testare l'*inserimento*, la *modifica* e l'*eliminazione* (implicitamente anche *lettura* nei primi due test) dei libri.
- **AuthorBookTest:** Consente di testare l'*aggiunta*, la *modifica* e l'*eliminazione* (ed anche *lettura* nei primi due test) degli *autori* ai *libri* e viceversa.
- **LocationTest:** Consente di testare l'*inserimento*, la *modifica* e l'*eliminazione* (implicitamente anche *lettura* nei primi due test) delle *sedi* del sistema bibliotecario.

- **StaffTest:** Consente di testare l'*inserimento*, la *modifica* e l'*eliminazione* (implicitamente anche *lettura* nei primi due test) dello *staff*.
- **UserTest:** Consente di testare l'*inserimento*, la *modifica* e l'*eliminazione* (implicitamente anche *lettura* nei primi due test) degli *utenti*.
- **ItemTest:** Consente di testare l'*aggiunta*, la *modifica* e l'*eliminazione* (implicitamente anche *lettura* nei primi due test) delle singole *unità di libro* (*Item*).
- **LoanTest:** Consente di testare l'*aggiunta*, la *modifica* e l'*eliminazione* (implicitamente anche *lettura* nei primi due test) dei *prestiti*.

Come è possibile osservare nell'elenco sovrastante i *JUnit test* volti a verificare la *corretta esecuzione* delle operazioni *CRUD* (*Create, Read, Update, Delete*) è stato fatto uso dell'engine *H2*, il quale consente di eseguire tali operazioni su un *database temporaneo* caricato in memoria (*RAM*).

Tale approccio consente *arginare* del tutto il problema di intaccare il database persistente dai dati fittizzi dei test.

Capitolo 3

Esecuzione

3.1 Requisiti

L'applicazione è stata sviluppata, eseguita e testata su sistemi *UNIX Like*, in particolare sulla distribuzione *GNU/Linux Debian*.

Per questo questo motivo le istruzioni che seguono saranno incentrate su questo ambiente, ma dovrebbero valere per tutti gli altri sistemi.

Per rendere la propria macchina pronta ad eseguire l'applicazione è necessario installare i pacchetti `openjdk` (la versione 8 è sufficiente) e `mvn` corrispondenti rispettivamente a *Open Java Development Kit* ed al tool *Apache Maven* [2]; in caso di Debian o derivate si usa il seguente comando:

```
$ sudo apt-get install openjdk-8-jdk mvn
```

In caso si volesse usare *Docker* allora le dipendenze sono *docker-compose*[3], ed ovviamente *docker* stesso. *Compose* è presente nella

maggior parte delle repo delle distro, il problema è che non è sempre aggiornato, per cui si ovvierà a questo possibile problema per mezzo di `pip`. [4]

La serie di comandi è la seguente, in caso di distro diversa da Debian usare il relativo gestore di pacchetti in sostituzione ad apt:

```
$ sudo apt-get install docker-ce python3-pip
$ sudo pip3 install docker-compose
```

3.2 Test

Per eseguire i test è necessario spostarsi all'interno della cartella del sorgente dell'applicazione e poi lanciare il relativo comando:

```
$ cd mivan
$ ./mvnw clean verify
```

3.3 Build

L'operazione di build genera un file `.jar` all'interno della cartella di nome `target`, la quale se non è già presente *verrà creata a runtime*.

Per eseguire la build è necessario spostarsi all'interno della cartella del sorgente dell'applicazione e poi lanciare il wrapper `mvnw`:

```
$ cd mivan
$ ./mvnw clean package spring-boot:repackage
```

3.4 Avvio tramite linea di comando

L'avvio immediato dell'applicazione, utile durante lo sviluppo si esegue con un solo comando. Si ricorda che prima è necessario spostarsi all'interno della cartella del sorgente dell'applicazione:

```
$ cd mivan
$ ./mvnw spring-boot:run
```

3.5 Avvio tramite Docker

Per poter eseguire l'applicazione per mezzo di Docker container è necessario soddisfare i requisiti indicati all'inizio di questo capitolo. Il vantaggio di usare Docker è che semplifica *notevolmente* sia la fase di sviluppo che di *deploy* dell'applicazione, specialmente per l'ultimo punto che ne riduce in maniera sensibile sia il tempo che la complessità.

> **Nota:**
> Per poter utilizzare docker è necessario avere i privilegi di root od essere nel gruppo `docker`!

Un comodo strumento di cui faremo uso per gestire i container è *Compose*, di cui, dato il file `.yml` già compilato alla radice della repository eseguiamo il comando di build:

```
$ sudo docker-compose build
```

E poi, una volta buildate l'immagine del container, lo lanciamo:

```
$ sudo docker-compose up
```

Bibliografia

[1] Spring. Using jpa. *https://docs.spring.io/spring-boot/docs/2.2.2.RELEASE/reference/htmlsingle/#boot-features-jpa-and-spring-data*, 2019.

[2] Apache Software Foundation. Apache maven documentation. *https://maven.apache.org/guides/index.html*, 2019.

[3] Docker Inc. Docker documentation. *docs.docker.com*, 2019.

[4] PyPA. The python package installer documentation. *pip.pypa.io/en/stable/*, 2019.