Università degli Studi di Milano - Bicocca

Progetto e Sviluppo del Software

Approfondimento MVC: Sviluppo di una WebApp con Spring MVC, RnW.

Riccardo Ghilotti - 879259

r.ghilotti@campus.unimib.it

Indice

Elenco delle figure

1 Introduzione: funzionalità di RnW

RnW, o *Read n' Write*, è una webapp ispirata ai social media, che permette agli utenti di condividere testi ed ad altri di commentarli.
Un utente in particolare può:

1. Scrivere dei testi.
2. Rendere i testi che ha scritto pubblici/privati.
3. Commentare i propri testi e quelli di altri utenti.
4. Cancellare i testi che ha creato.
5. Segnalare i testi che può visualizzare.
6. Modificare il nome e la password del proprio account.
7. Cancellare il proprio account.

Alcuni utenti hanno privilegi di amministratore, che consentono loro di:

1. Cancellare gli account degli altri utenti.
2. Cancellare testi di altri utenti.
3. Cambiare la visibilità(pubblica/privata) di altri testi oltre ai propri.
4. Cambiare nomi di altri utenti.
5. Risolvere segnalazioni.

Infine, i testi hanno dei pre-requisiti da rispettare, ovvero:

1. Un testo deve essere composto da tre macro-sezioni: introduzione, corpo e conclusione.
2. Ogni macro-sezione non può essere vuota.
3. Ogni testo deve avere un titolo.

Le tre macro-sezioni possono essere composte da una o più sotto-sezioni.

2 Introduzione: cambiamenti durante lo sviluppo

Durante lo sviluppo della webapp sono stati fatti dei cambiamenti sia all'architettura dell'applicazione, che al tipo di database utilizzato per contenere i dati.

2.1 Cambiamento del modello di database

L'applicazione è passata da un modello relazionale, ovvero MySQL, ad un modello NoSQL, ovvero MongoDB. Questo perchè il modello NoSQL è più adatto a memorizzare contenuti testuali di varie dimensioni.
Inoltre, salvare testi con i pre-requisiti definiti in precedenza è molto più semplice con quest'ultimo, in quanto non bisogna fare alcuna trasformazione strutturale delle macro-sezioni: queste possono essere semplicemente archiviate come vettori, la cui lunghezza e struttura è equivalente al numero ed al contenuto delle relative sotto-sezioni.

2.2 Cambiamento nell'architettura

Durante lo sviluppo dell'app sono stati introdotti due strati per separare meglio i compiti delle diverse componenti:

- Uno strato di servizio.
- Uno strato di mapper.

In breve, lo strato di servizio gestisce la logica di business del progetto, mentre lo strato di mapper interagisce con il database e mappa i documenti che gli vengono ritornati nelle classi che fanno parte del modello. Entrambi gli strati e le classi che fanno parte di essi verranno esplorati meglio nella sezione del documento che mostra come è strutturata l'applicazione.

3 Applicazione MVC: Il pattern MVC

Il pattern MVC, Model-View-Controller, è un pattern architetturale che prevede la divisione dell'applicazione in tre strati principali:

- La View: rappresenta lo strato più vicino all'utente. In essa è presente tutta la logica di presentazione, ovvero la logica che permette all'utente di visualizzare l'applicazione e le sue funzionalità ed interagirci.
- Il Model: rappresenta lo strato in cui è contenuta la maggior parte della logica dell'applicazione, composta da interazione con il database, controlli di validazione, definizione del dominio e logica di business. Nel caso di RnW il Model è molto semplice, questo perchè si affida ad uno strato di servizio, per gestire la logica di business e di validazione, ed uno strato di accesso ai dati, che gestisce l'interazione con il database. In questo modo esso si occupa solamente di definire gli oggetti del dominio, rimanendo snello. Nel documento verrà anche chiamato "Modello".
- Il Controller: rappresenta lo strato intermediario tra Modello e View; qui vengono raccolte le chiamate della View, viene chiamato il Modello per fornire una risposta e quest'ultima viene formattata in modo che sia leggibile dalla View. Nel documento verrà anche chiamato "strato di Controllo".

Come già anticipato, in RnW sono stati introdotti due strati aggiuntivi:

- Lo strato di Servizio: aggiungere questo strato smorza la complessità del Modello e aiuta a separare i compiti ad esso assegnati. Lo strato di servizio si occupa della logica di validazione e di business, gestendo le interazioni tra il Controller e lo strato che interagisce con il database.
- Lo strato di mapper: Questo strato interagisce con il database per mappare gli oggetti persistenti in classi Java e trasformare gli oggetti del dominio in documenti che possono essere salvati nel Database. Nel documento verrà anche chiamato "strato DAO".

4 Applicazione MVC: RnW

In questa sezione verranno esplorati meglio i differenti strati che compongono l'applicazione, mostrando i compiti delle varie componenti della webapp. Per ogni strato verranno presentati degli esempi.

4.1 La View

Questo strato contiene i diversi file `.jsp` che compongono la View. In particolare, essi forniscono all'utente la possibilità di visualizzare e interagire facilmente con i contenuti della webapp.
Le pagine sono scritte utilizzando HTML, CSS e Javascript:

- HTML fornisce uno scheletro per le diverse pagine.
- CSS, tramite il framework Bootstrap[1], ne modifica lo stile.
- Javascript sfrutta il DOM per modificare le diverse pagine, oltre ad effettuare chiamate al Controller tramite AJAX.

4.1.1 Esempio: Pagina del profilo di un utente

In questa pagina, tramite dei controlli effettuati lato server, vengono aggiunti o rimossi dei bottoni che garantiscono agli utenti funzionalità aggiuntive, come la possibilità di cambiare il nome del profilo visualizzato.
Quando viene caricato il profilo di un utente, tramite javascript DOM, verranno aggiunti alla pagina i testi scritti da quest'ultimo, recuperati dal controller.
La lista di testi è diversa se l'utente che visita l'account è anche il suo possessore, mostrando pure i testi privati, però la pagina della View non è al corrente di ciò, in quanto si limita solo a mostrare lista di testi che gli è stata ritornata dal Controller.
Sempre tramite valori booleani passati dal Controller, la pagina permetterà all'utente di effettuare diverse operazioni:

1. Se l'utente non è il possessore dell'account, o non ha effettuato l'accesso, allora potrà solamente visualizzare i testi pubblici dell'account. 1
2. Se l'utente è il possessore dell'account, potrà visualizzare, oltre ai testi pubblici, anche i testi privati. In più appariranno dei bottoni che gli permettono di cambiare il proprio nome, la propria password e scrivere un nuovo testo. 2
3. Se l'utente è un amministratore, che sia il possessore dell'account o meno, può accedere ad una pagina speciale, che gli permette di gestire gli utenti. Inoltre, può cambiare il nome dell'account. 3

[1] https://getbootstrap.com/

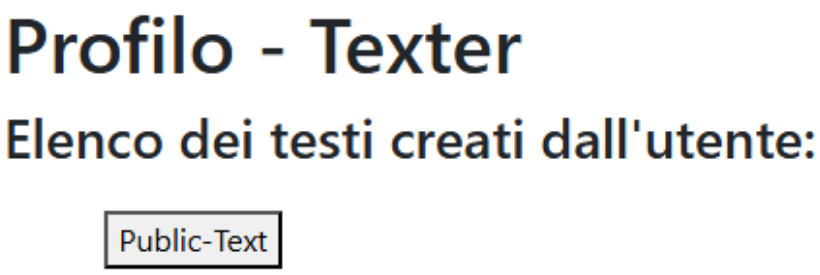


Figura 1: vista di un utente che non ha effettuato l'accesso che visita il profilo di un utente

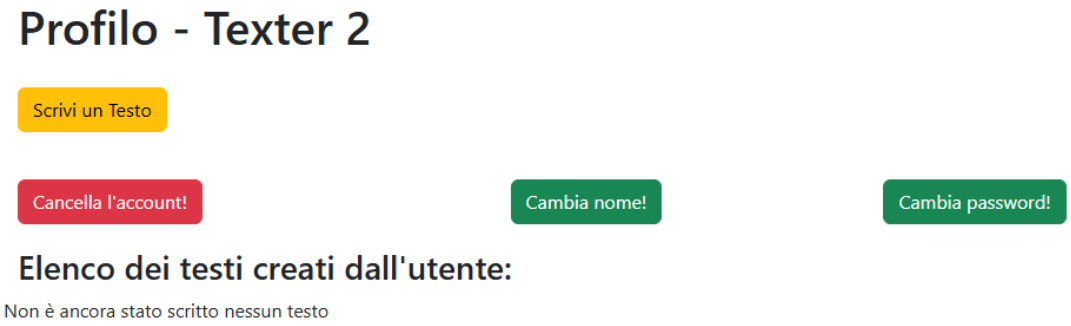


Figura 2: vista di un utente che ha effettuato l'accesso che visita il proprio profilo

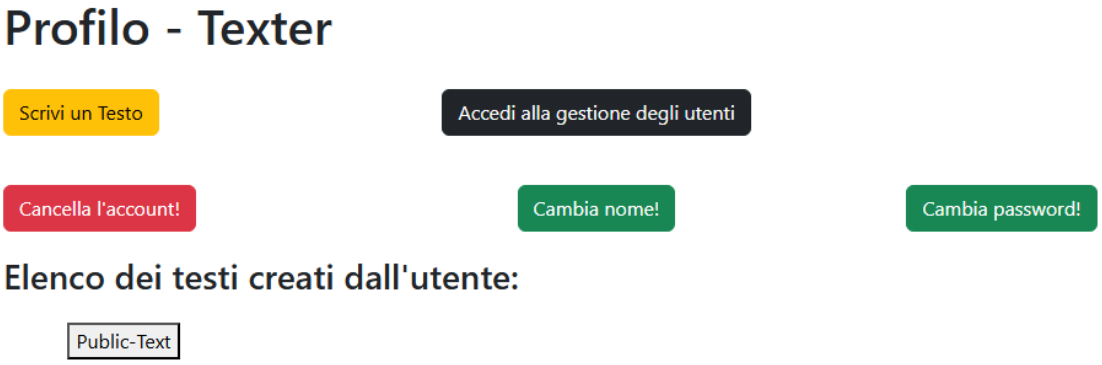
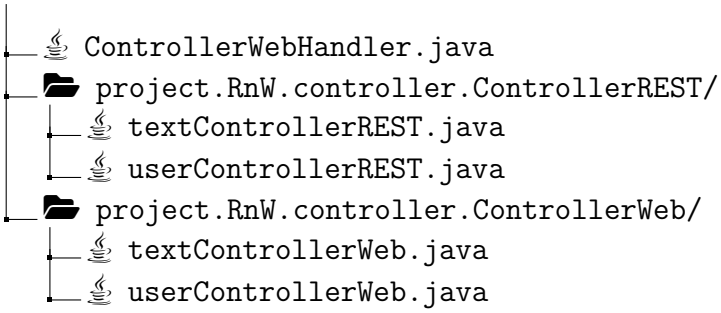


Figura 3: vista di un utente con i permessi da amministratore che visita il proprio profilo

4.2 Il Controller

Lo strato che si occupa di coordinare le interazioni tra Model e View e di gestire la logica di formattazione è strutturato nel seguente modo:

```
project.RnW.controller/
    ControllerRESTHandler.java
    ControllerUtils.java
```



Il ruolo delle diverse classi

Le classi che gestiscono le chiamate che arrivano dalla View sono `textControllerREST`, `userControllerREST`, `textControllerWeb` e `userControllerWeb`. Esse sono divise per tipo di chiamata (chiamata REST o chiamata che reindirizza l'utente ad una nuova pagina) e per fine della chiamata (chiamata che effettua un'operazione che ha come protagonista un testo o chiamata che effettua un'operazione che ha come protagonista un utente). Inoltre, queste classi trasformano le eccezioni che vengono propagate dalle altre componenti dell'applicazione fino allo strato di Controllo in messaggi visualizzabili dalla View.
La classe `ControllerUtils` contiene la maggior parte, se non tutta, la logica di formattazione dell'applicazione. Questa classe, infatti, contiene solamente metodi che vengono utilizzati all'interno delle altre classi che fanno parte di questo strato, rendendole più semplici da leggere e diminuendo le ripetizioni di codice.
Infine, le due classi `ControllerRESTHandler` e `ControllerWebHandler`, sono classi *@ControllerAdvice*, ovvero il cui obiettivo è gestire eccezioni che vengono lanciate dai controller.

4.2.1 Esempio: Accesso alla pagina di gestione degli utenti.

In questa sezione verranno mostrate le operazioni effettuate da questo strato quando riceve una chiamata dalla View.
Si supponga che l'utente prema il bottone per passare alla pagina di gestione degli utenti e la View invii una chiamata, alla quale lo strato di Controllo risponderà con il metodo `adminView` 1:

```
1         adminView(String userId){
2
3         handlingPage = newPage()
4         user = serviceUser.getUser(userId)
5
6         if(user.isAdmin()){
7                 handlingPage.add(ControllerUtils.getUsersIds();
8                 handlingPage.add(ControllerUtils.getReports();
9         }
10        else{
11            handlingPage = home;
12            handlingPage.add("Non hai accesso a questa parte dell'app");
13        }
14        return handlingPage;
15    }
```

Listing 1: Pseudocodice per adminView

Lo strato di Controllo quindi, comunica con altre tre classi per fornire all'utente la nuova View. Inizialmente contatta lo strato di Servizio per recuperare l'oggetto che corrisponde all'utente che effettua la chiamata (`serviceUser.getUser(userId)`), poi contatta il Modello, che controllerà

se l'utente è un admin oppure no (`user.isAdmin`) e infine, chiama ControllerUtils per eseguire delle operazioni di supporto al metodo, ovvero `getUsersIds()` 2 e `getReports()` 3.

```
getUsersIds(){
    userList = serviceUser.getAllUsers();
    userIds = new List();
    for (User u in userList) {
        temp = {u.getId(),
                u.getName()};
        userIds.add(temp);
    }
    return userIds;
}
```

Listing 2: Codice per getUsersId

```
getReports(){
    reportsList = serviceReport.getReports();
   infoReportsList = new List();
    for(Report r in reportsList) {
        Text t = r.getReported();
        temp = {t.getId().toString(),
                t.getTitle(),
                t.getAuthor().getId().toString(),
                t.getAuthor().getName(),
                r.getReporter().getId().toString(),
                r.getReporter().getName(),
                r.getContent(),
                r.getId().toString()};
        infoReportsList.add(temp);
    }
    return infoReportsList;
}
```

Listing 3: Codice per getUsersId

Come si può vedere dai frammenti di codice 2 e 3, questi appartengono allo strato di Controllo perché entrambi i metodi preparano i dati per inviarli alla View.

4.3 Lo strato di Servizio

Lo strato di Servizio si occupa di coordinare le classi che definiscono il dominio e lo strato DAO, per rispondere al Controller quando l'operazione richiesta dall'utente coinvolge della logica di business. Inoltre, se i dati inviati dall'utente devono essere convalidati, quest'operazione viene svolta in questo strato.
Lo strato di servizio, come i prossimi due strati, è diviso in quattro classi, ognuna per oggetto del dominio. Ogni classe porta a termine le operazioni che riguardano quello specifico oggetto. Dunque le quattro classi sono `serviceUser`, `serviceText`, `serviceReport`, `serviceComment`.

4.3.1 Esempio: salvare un testo

Nella seguente sezione verranno presentati i metodi dello strato di Servizio che trattano il salvataggio di un testo.
In precedenza, un metodo dello strato di Controllo avrà già chiamato `serviceText.saveText`, fornendogli i parametri necessari per svolgere l'operazione. In 4, si può vedere che il metodo

si assicura di salvare i caratteri speciali nelle stringhe, convertendone il formato, trasforma le diverse macro-sezioni da stringhe di array Json a liste e se il testo è nuovo chiama `saveNewText` 5, altrimenti chiama `updateText` 6.

```
    saveText(String id, String title, String intro,
      String corpus, String conc, String userId){

        intro = new String(intro.changeFormat());
        corpus = new String(corpus.changeFormat());
        conc = new String(conc.changeFormat());


    List introList = null;
    List corpusList = null;
    List concList = null;
    List commentList = null;
    try {
      introList = intro.toList();
      corpusList = corpus.toList();
      concList = conc.toList();
    } catch (JsonProcessingException e) {
      print("ERROR: " + e.toString());
    }

    if(introList.isEmpty() || corpusList.isEmpty() || concList.isEmpty() ||
   title.isEmpty())
      throw new EmptyMacroSectionsException(
          "Macro-sections cannot be empty");

    if(/**text is new*/) {
      Text t = saveNewText(title, introList, corpusList,
          concList,  true, mapperUser.getUser(userId));
        }
    else {
      User u = mapperUser.getUser(new ObjectId(userId));
      Text t = mapperText.getText(new ObjectId(id));
      if(t.title() != title)
            throw new ModifiedTitleException("Title has been modified");
      updateText(introList, corpusList, concList, u, t);
    }
    }
```

Listing 4: Codice per saveText

```
saveNewText(String title, List intro, List corpus, List conclusion, boolean
   isPrivate, User author){
      ObjectId id = mapperText.insert(
          title, intro, corpus, conclusion, isPrivate, author);
      if(id == null)
        throw new TextUnsavedException("Text was not updated");
      Text t = new Text(
          id,
          title,
          intro,
          corpus,
          conclusion,
          null,
          isPrivate,
```

```
            author);
        return t;
    }
```

Listing 5: Codice per saveNewText

```
updateText(List intro, List corpus, List conc, User u, Text t){
        t.changeIntro(intro, u);
        t.changeCorpus(corpus, u);
        t.changeConclusion(conc, u);
        if(!mapperText.update(t.getId(), intro, corpus, conc))
          throw new TextUnsavedException("Text was not updated");
    }
```

Listing 6: Codice per updateText

4.4 Il Modello

Il Modello è formato da classi molto semplici, popolate soprattutto da Getter e Setter. Queste classi, però, modellano gli oggetti che l'applicazione utilizza per compiere il suo scopo, quindi sono assolutamente necessarie e fondamentali per il funzionamento dell'applicazione stessa.
Queste classi inoltre definiscono alcune regole di dominio, come il fatto che l'unico utente che può modificare le macro-sezioni è l'autore del testo.

4.5 Lo strato dei Mapper

Lo strato dei mapper gestisce tutte le interazioni con il database e mappa i documenti che gli vengono ritornati da esso negli oggetti Java che appartengono al Modello.
Come per il Modello e lo strato di Servizio, lo strato dei mapper contiene solamente quattro classi, una per rispettivo oggetto nel Modello (`mapperUser`, `mapperText`, `mapperComment`, `mapperReport`).

4.5.1 Esempio: il recupero di un utente

Quando lo strato di Servizio deve recuperare un utente chiama la classe `mapperUser`, la quale possiede due metodi per la ricerca dell'utente in questione: `getUser` 7, che lo cerca tramite il suo id, e `getAndMapUser` 8, che lo cerca tramite mail e password.

```
getUser(ObjectId id2){
        docUser = Database.find(eq("_id",id2));
        return mapUser(docUser);
    }
```

Listing 7: Codice per getUser

```
getAndMapUser(String mail, String password){
    Document docUser = mapperUser.getUser(mail, password);
    return mapUser(docUser);
  }
```

Listing 8: Codice per getAndMapUser

Entrambi i metodi si affidano a `mapUser` 9 per fare l'operazione di mappatura da `Document` a `User`.

```
mapUser(Document docUser){
    if(docUser == null)
      throw new AccountNotFoundException("No user was found");
    ObjectId id = docUser.getId("_id");
      String name = docUser.getString("name");
      boolean admin = docUser.getBoolean("admin", false);

      return new User(id, name, admin);
  }
```

Listing 9: Codice per mapUser

Come si può vedere dall'esempio, questi metodi cercano gli oggetti nel database e si occupano di ritornare l'oggetto mappato.

5 I test

Infine sono stati creati dei test per controllare il corretto funzionamento dell'applicazione.

5.1 I test di Integrazione

Questo tipo di test coinvolge più componenti di un'applicazione.
In particolare questi test si assicurano che:

- L'utente venga reindirizzato alla pagina corretta.
- Le eccezioni lanciate dagli strati sottostanti non vengano ritornate come tali, ma mostrate come messaggi che permettono all'utente di comprendere cosa sta succedendo.
- Le informazioni ritornate dai controller siano formattate in maniera corretta.
- Gli attributi che la View utilizza per modificare la composizione delle diverse pagine siano restituiti correttamente.

Testando multiple componenti dell'applicazione essi falliscono in caso di conflitti tra i metodi che fanno parte dell'applicazione, procedure incoerenti tra diversi strati, errori del database ed eccezioni gestite scorrettamente.

5.2 I test unitari

I test unitari invece si concentrano sulla logica dei metodi degli strati sottostanti al controller. Questo per verificare che la loro esecuzione in un ambiente isolato non porti ad errori. Se ci sono problemi con i test di integrazione, i test unitari possono aiutare ad identificare quale componente fallisce, in quanto possono sorgere errori solo se la logica del metodo stesso è sbagliata.
Nonostante ciò, non sono stati stilati molti test unitari per le classi appartenenti allo strato di Controllo e che rispondono alle chiamate della View, in quanto la loro logica è spesso poco complessa e viene già testata nei test di integrazione.
In particolare molti test unitari si concentrano sul corretto funzionamento in caso di errore.

6 Conclusione e sviluppi futuri

RnW quindi presenta un'architettura a strati che divide le classi per separare i compiti e rendere il codice più modulare e comprensibile. Questo semplifica l'implementazione di nuove feature e la correzione di errori.
Inoltre, se devono essere fatte modifiche al codice già esistente, nuovi errori possono essere facilmente riscontrati tramite i test.
Siccome l'ordine, l'alta leggibilità del codice e l'alta testabilità sono punti di forza della webapp, in futuro potrebbero essere implementate funzionalità complesse che arricchiscono l'applicazione, cercando di mantenere le qualità sopra menzionate. Si potrebbe anche pensare di modificare funzionalità già presenti, o inserirne di molto simili, che riutilizzano il codice degli strati adiacenti per aggiungere percorsi d'azione specifici per alcuni utenti.