Università degli Studi di Milano Bicocca

Dipartimento di Informatica

Corso di Laurea Magistrale

Processo e Sviluppo del Software

Assignment 3

ANNO ACCADEMICO 2024/2025

Alessandro Isceri - 879309

Indice

1. Introduzione

1.1 Obiettivo del Progetto

L'obiettivo principale del progetto è approfondire uno degli argomenti trattati durante le lezioni frontali, in particolare, verrà approfondito il pattern architetturale Model-View-Controller (MVC). Il progetto consiste nello sviluppo di un'applicazione web basata sul framework SpringMVC, al fine di esplorare a livello pratico le potenzialità di questo pattern.

1.2 Breve descrizione dell'app

1.2.1 Funzionalità implementate

L'applicazione sviluppata, **Listify**, è una web-app progettata per la gestione di to-do list. Le sue principali funzionalità includono:

- Registrazione, autenticazione ed eliminazione degli utenti.
- Creazione, rinomina ed eliminazione di una o più to-do list.
- Creazione, modifica ed eliminazione di attività associate alle to-do list.
- Visualizzazione di un grafico a torta che fornisce una panoramica dello stato delle attività per una to-do list (numero di attività To Do, In Progress, e Completed).

Rispetto ai requisiti originali, non è stata implementata la funzionalità che permette di visualizzare dei grafici per misurare la produttività dell'utente, come ad esempio il numero delle attività completate nell'arco di una settimana.

1.2.2 Descrizione delle entità

Ogni utente è identificato da un indirizzo e-mail e un username (entrambi devono essere univoci), ogni utente deve avere una password e può possedere più to-do list.
Ogni to-do list è identificata da un identificatore numerico, deve avere un nome e può contenere più attività.

Ogni attività è identificata da un identificatore numerico, deve avere un nome, può avere una data di scadenza (facoltativa) e una priorità (facoltativa); ad ogni attività è associata una categoria (To Do, In Progress, Completed).

La priorità di un'attività è rappresentata da un numero intero compreso tra 1 e 10: dove 1 indica la massima priorità, mentre 10 rappresenta la priorità minima. Se la priorità è impostata a -1, significa che non è stata definita (quell'attività non ha una specifica priorità).

Per maggiori dettagli sulla struttura delle entità nel database, si rimanda alla sezione 2.1.6.

1.3 Tecnologie utilizzate

Per lo sviluppo dell'applicazione sono state utilizzate le seguenti tecnologie:

- **Frontend**:
  - **JSP (JavaServer Pages)**: per la generazione dinamica delle pagine HTML.
  - **JavaScript**: per l'aggiunta di interattività e funzionalità lato client e per inviare richieste Ajax al server.
  - **Bootstrap**: framework CSS utilizzato per creare un'interfaccia utente responsive e moderna.
  - **FontAwesome**: libreria di icone utilizzata per migliorare l'aspetto grafico delle pagine.
- **Backend**:
  - **SpringMVC (Java)**: framework utilizzato per implementare il pattern MVC, garantendo una chiara separazione tra logica di business, interfaccia utente e gestione delle richieste.
- **Database**:
  - **MySQL**: database relazionale eseguito localmente per la persistenza dei dati.
- **Testing**:
  - **JUnit 5**: framework per test automatizzati, utilizzato per garantire la qualità e l'affidabilità del codice.
  - **Mockito**: libreria per il mocking di dipendenze, utilizzata per simulare comportamenti di componenti esterni durante i test unitari.

1.4 Guida all'installazione

Questa guida fornisce i passaggi necessari per configurare e avviare l'applicazione Listify.

1. **Installazione di XAMPP**:
    - Scaricare XAMPP per eseguire il database in locale.
    - Una volta installato, avviare i servizi `Apache` e `MySQL`.
    - Importare il database tramite questo script.
2. **Installazione di Spring Tools Suite**:
    - Scaricare e installare Spring Tools Suite per Eclipse.
3. **Clonazione del repository**:
    - Clonare il repository Git del progetto oppure scaricarne il contenuto.
    - Importare il progetto in Eclipse.
4. **Configurazione del progetto**:
    - Aprire le `Properties` del progetto:
        - **Java Compiler**: impostare `JDK compliance` su Use compliance from execution environment.
        - **Project Facets**: convertire il progetto in *faceted form* e selezionare:
            * `JavaScript 1.0`.
            * `Java 17`.
            * `Dynamic Web Module 5.0`.
    - Applicare le modifiche e salvare.
5. **Esecuzione dei test di integrazione**:
    - Assicurarsi che il database sia in esecuzione (dovrebbe essere già stato avviato durante il primo passaggio).
    - Eseguire il comando `Run As > Maven Install`.
    - Assicurarsi che la build abbia successo.
6. **Avvio del server**:
    - Eseguire il comando `Run As > Run On Server` selezionando il server Tomcat v11.0.
    - Se il comando non è disponibile, occorre configurarlo manualmente:
        - `Run As > Run Configurations > Tomcat v11.0 Server at localhost`.

- In caso di errore, verificare che le dipendenze di Maven vengano importate correttamente:
  - `Properties > Deployment Assembly > Add > Java Build Path Entries > Next > Maven Dependencies > Finish`.
- Salvare e chiudere.
- Riavviare il Server.

7. **Accesso all'applicazione**:
   - Una volta avviato il server, accedere all'applicazione tramite il browser all'indirizzo: `http://localhost:8080/listify/`.

2. Architettura del Sistema

2.1 Pattern MVC

2.1.1 Descrizione del pattern MVC

Il pattern MVC (Model-View-Controller) è pattern architetturale che separa un'applicazione in tre componenti principali:

- **Model**: gestisce i dati, la logica di business e le operazioni di accesso al database.
- **View**: si occupa della presentazione dei dati e dell'interfaccia utente.
- **Controller**: agisce come intermediario tra il Model e la View, gestisce le richieste dell'utente e invia le corrispondenti risposte.

Questa separazione consente di migliorare la manutenibilità, la modularità e la testabilità del codice.

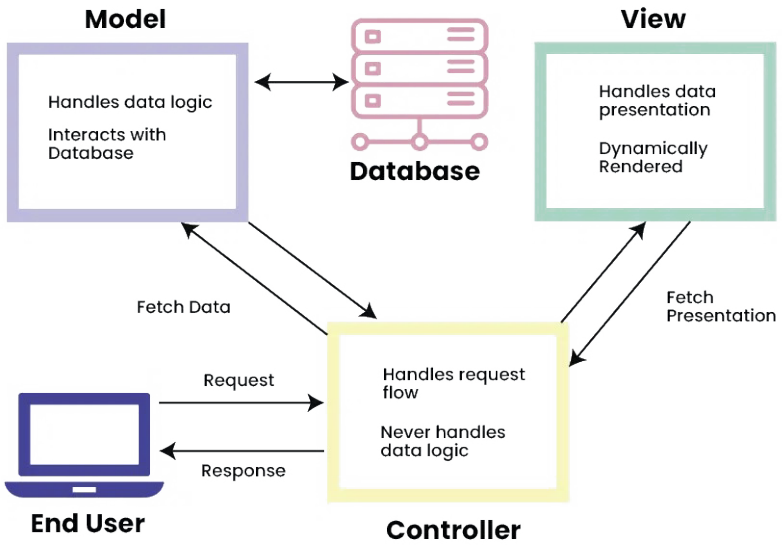

Figura 2.1: Architettura MVC

2.1.2 Architettura di Listify

In Listify l'architettura è stata implementata come segue:

1. **View**: Sono stati utilizzati dei file '.jsp': `index.jsp`, `home.jsp`, `errorPage.jsp`, `toDoList.jsp`, arricchiti con codice JavaScript per gestire eventi derivanti dall'interazione dell'utente.
2. **Controller**: Sono stati implementati due controller per gestire le richieste: `ViewController.java` e `APIController.java`.
3. **Model**:
   - Classi di dominio: `Activity.java`, `ToDoList.java`, `User.java`.
   - Logica di business: `ListifyService.java`.
   - Interazione con il database: `ActivityRepository.java`, `UserRepository.java` e `ToDoListRepository.java`.
4. **Database**: un database relazionale MySQL eseguito in locale.

Per quanto riguarda le classi Java, esse sono state organizzate nei seguenti package:

- Package **controllers**: Contiene i due controller dell'applicazione.
- Package **domain**: Include le classi di dominio.
- Package **repository**: Comprende le classi per l'interazione con il database.
- Package **services**: Contiene la classe che implementa la logica di business.
- Package **config**: Include le classi di configurazione di SpringMVC, utilizzate per specificare quali richieste gestire, dove si trovano le view e dove sono localizzate le risorse statiche.

2.1.3 View

Le view si trovano nella cartella `src/main/webapp/WEB-INF/views`, ad eccezione della pagina entry-point dell'applicazione, `index.jsp`, che si trova direttamente in `WEB-INF`.
Ogni file `.jsp` è associato a uno script `.js` e ad un file `.css` per la formattazione. Questi file si trovano nella cartella `src/main/webapp/resources/` rispettivamente nelle cartelle `scripts` e `styles`.
Di seguito vengono descritte le view dell'applicazione:

- **index.jsp**: rappresenta l'entry-point dell'applicazione, e presenta un form di login/registrazione per consentire all'utente di accedere alla propria area riservata.

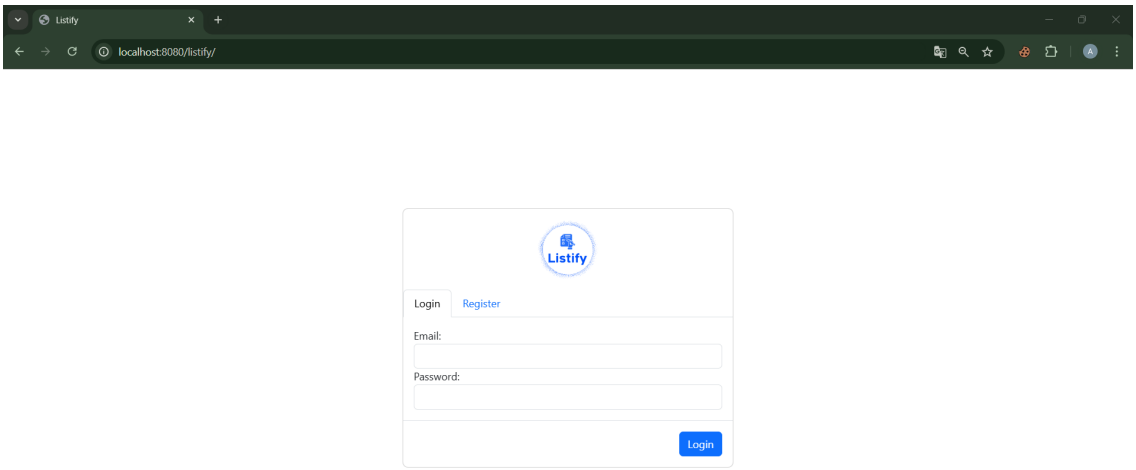

Figura 2.2: index.jsp

- **home.jsp**: se l'utente effettua il login o la registrazione correttamente, viene reindirizzato alla sua home page, dove può visualizzare le proprie to-do list (e ha la possibilità di crearne di nuove, rinominarle o eliminarle).

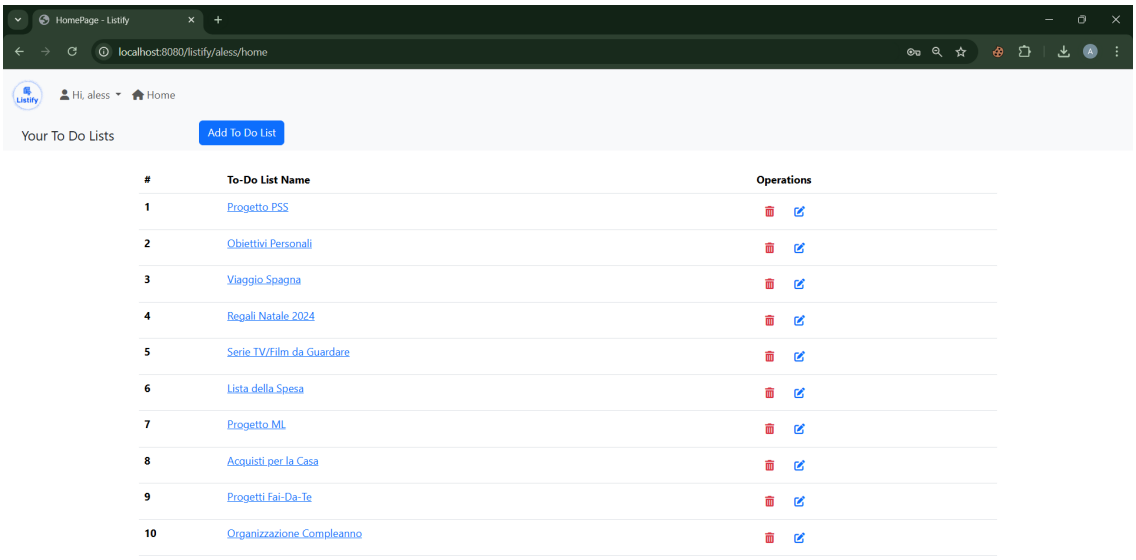

Figura 2.3: home.jsp

toDoList.jsp: cliccando sul nome di una lista nella home page, l'utente viene reindirizzato a questa pagina, dove può visualizzare le attività della lista selezionata, suddivise in To Do, In Progress e Completed. È presente un grafico a torta che mostra il numero di attività per categoria. L'utente può aggiungere nuove attività, eliminarle o modificarle cliccando su di esse. Le attività possono essere spostate tra le varie categorie tramite drag-and-drop.

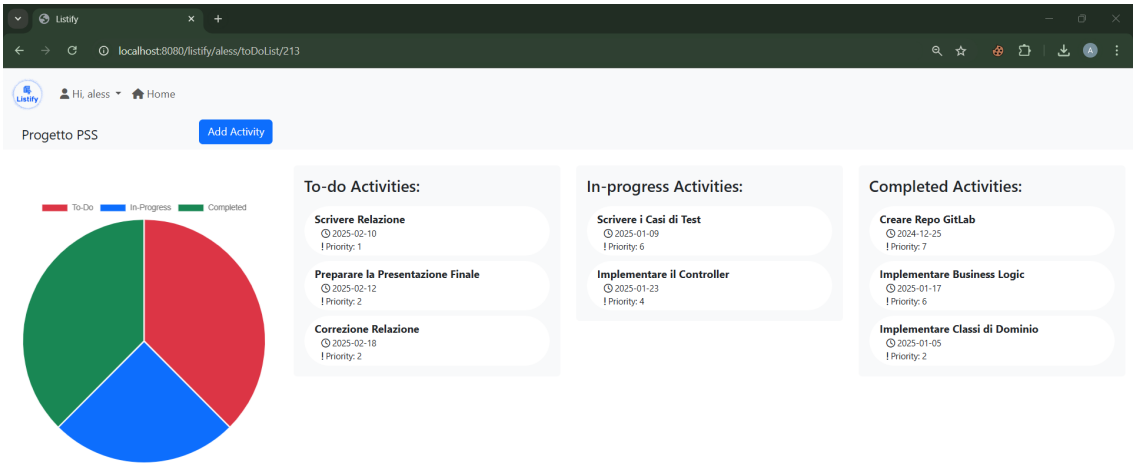

Figura 2.4: toDoList.jsp

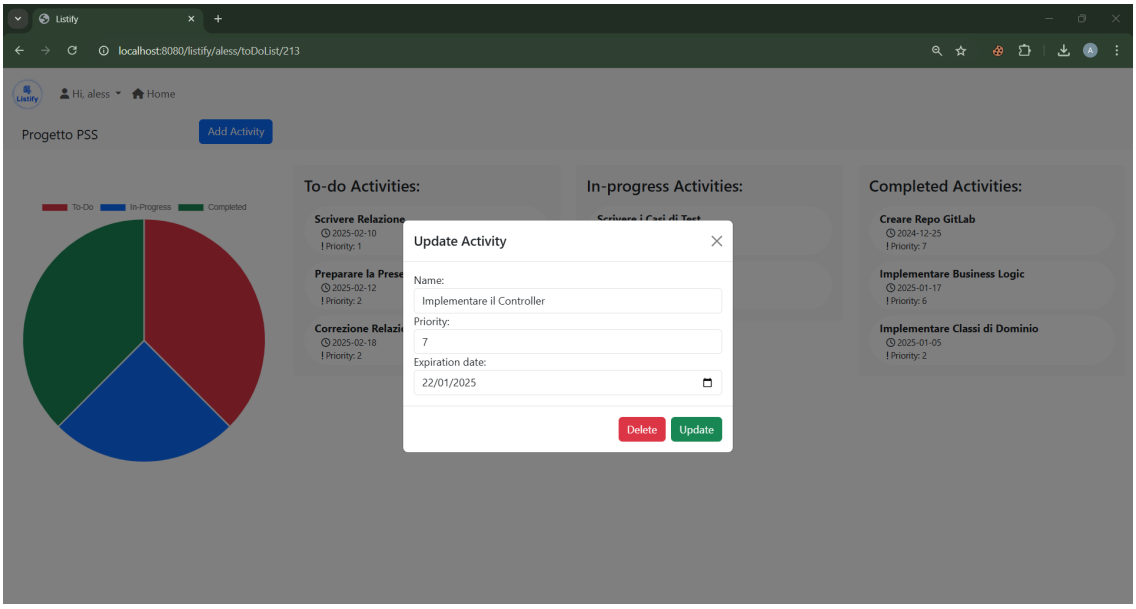

Figura 2.5: toDoList.jsp - Aggiornamento di un'attività

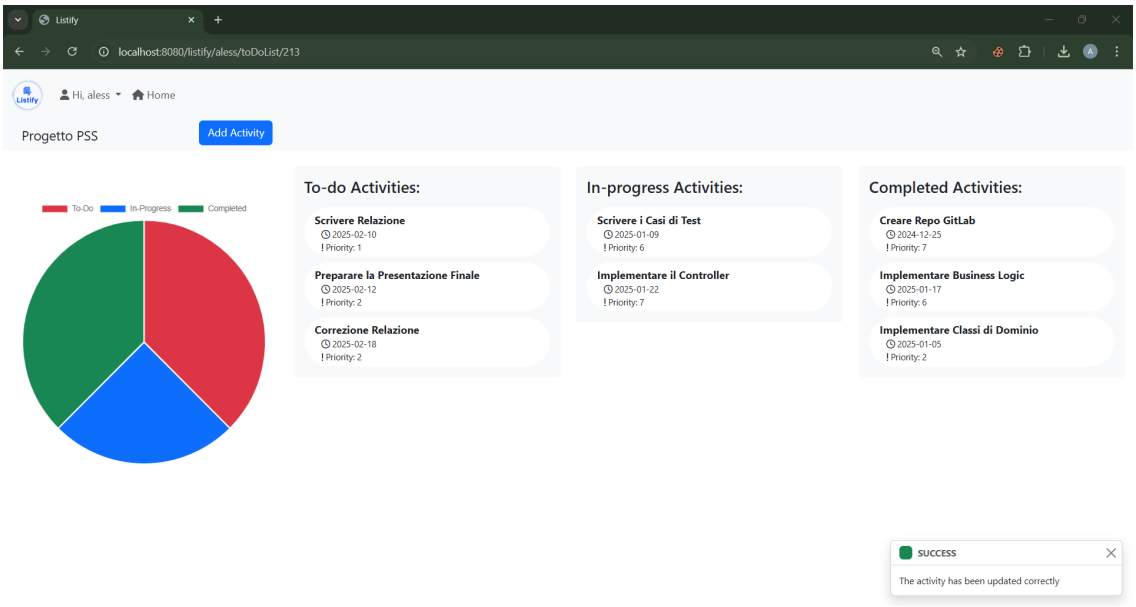

Figura 2.6: toDoList.jsp - Attività aggiornata con successo

- **errorPage.jsp**: questa pagina viene visualizzata quando un utente non loggato (o loggato con un altro account) tenta di accedere ad un'area riservata. Si tratta di una semplice pagina di errore.

In ogni pagina è presente una navbar che facilita la navigazione, permettendo all'utente di tornare alla home page, effettuare il logout o cancellare il proprio account.

Infine, ogni volta che un'operazione va a buon fine (o meno), viene visualizzato un messaggio di successo (o errore) in basso a destra della view tramite un toast.

2.1.4 Controller

Come accennato in precedenza, sono stati implementati due controller:

- **APIController**: gestisce le richieste Ajax, consentendo operazioni asincrone senza avere il bisogno di ricaricare continuamente la pagina.
- **ViewController**: gestisce le richieste di pagine jsp, restituendo le view richieste dall'utente.

Quando l'utente invia una richiesta, se è valida, uno dei due controller la riceve. Il controller verifica che l'utente sia correttamente loggato e abbia l'autorizzazione per accedere al contenuto richiesto. Se necessario, il controller delega l'esecuzione di alcune operazioni alla classe contenente la logica di business (`ListifyService.java`), che a sua volta potrebbe richiedere alle classi Repository

di aggiornare i dati in maniera persistente sul Database. Se tutte le operazioni hanno successo, il ViewController restituisce una nuova view all'utente. Nel caso di richiesta Ajax, l'APIController restituisce un codice di ritorno HTTP che indica la riuscita (o meno) di una determinata operazione.

2.1.5 Model

Il model dell'applicazione è composto dalle classi di dominio, dalla classe contenente la logica di business e dalle classi che interagiscono con il database.

- **Classi di dominio**: classi semplici, principalmente composte da metodi `getters` e `setters`, che rappresentano le entità principali dell'applicazione (`User`, `ToDoList`, `Activity`).
- **Logica di Business**: una singola classe che gestisce i dati a runtime. Quando il controller invoca questa classe per eseguire un'operazione, la classe prima interagisce con il database per eseguire l'operazione richiesta, se necessario[1]. Se l'operazione sul database ha successo, i dati vengono aggiornati a runtime e la logica di business restituisce il risultato al controller, indicando se l'operazione è andata a buon fine. In caso di successo, la view viene aggiornata; in caso contrario, viene restituito un messaggio di errore.
- **Interazione con il database**: per l'interazione con il database, non è stato adottato il pattern Object-Relational Mapping (ORM), poiché il focus del progetto è sull'approfondimento del pattern MVC e non sulla gestione avanzata del database. L'interazione con il database è gestita da classi specifiche che eseguono query SQL dirette (in base ai parametri ricevuti durante la chiamata).

Per semplicità, all'avvio del server viene caricato tutto il database in memoria. Questa operazione consente di ridurre il numero di accessi al database durante l'esecuzione, migliorando la velocità di risposta dell'applicazione. Tuttavia, questa scelta potrebbe essere un problema in caso di grandi volumi di dati.

Questa soluzione è stata adottata principalmente per permettere di gestire i dati in maniera persistente tra le diverse sessioni di un utente, ma senza dover affrontare la complessità di implementazione del pattern ORM.

[1] Non tutte le operazioni hanno bisogno di accedere al DB: ad esempio, per effettuare il login basta verificare se l'utente è registrato a runtime.

2.1.6 Database

Il database di Listify è composto da tre entità: `user`, `todolist` e `activity`, ognuna delle quali ha una specifica funzione nell'applicazione.

- **user**:
  - `email`: campo univoco che rappresenta l'email dell'utente.
  - `password`: password dell'utente.
  - `username`: campo univoco che rappresenta il nome utente (chiave primaria).
- **todolist**:
  - `id`: chiave primaria, generata automaticamente (AI).
  - `name`: nome della lista.
  - `username`: chiave esterna che fa riferimento all'utente che ha creato la lista.
- **activity**:
  - `id`: chiave primaria, generata automaticamente (AI).
  - `name`: nome dell'attività.
  - `priority`: priorità dell'attività.
  - `expirationDate`: data di scadenza dell'attività, può essere `NULL`.
  - `category`: categoria dell'attività (To Do, In Progress, Completed).
  - `list_id`: chiave esterna che fa riferimento alla lista a cui appartiene l'attività.

Per quanto riguarda i vincoli di integrità referenziale, sono stati impostati i seguenti vincoli:

- `ON UPDATE CASCADE`: se i dati relativi a un utente o una lista vengono aggiornati, anche le righe correlate nelle tabelle `todolist` e `activity` verranno aggiornate di conseguenza.
- `ON DELETE CASCADE`: se un utente o una lista vengono eliminati, tutte le relative to-do list e attività verranno automaticamente rimosse dal database.

3. Test

I test eseguiti simulano i seguenti scenari per APIController e ListifyService:

- Registrazione di un nuovo utente.
- Login/logout dell'utente.
- Creazione di una nuova to-do list.
- Rinomina di una to-do list esistente.
- Aggiunta di nuove attività a una to-do list.
- Modifica delle attività esistenti (nome, priorità, scadenza, categoria).
- Eliminazione di attività da una to-do list.
- Eliminazione di una to-do list.
- Eliminazione dell'account utente.

I test eseguiti simulano i seguenti scenari per ViewController:

- Richiesta della home page.
- Richiesta della to-do list page.

3.1 Test di Unità

3.1.1 Obiettivi dei test di unità

I test di unità sono stati implementati per verificare il corretto funzionamento delle singole componenti dell'applicazione.

Gli obiettivi principali dei test di unità sono:

- Verificare che i metodi dei controller (sia `ViewController` che `APIController`) restituiscano i risultati attesi. Per garantire un test unitario del web layer, è stato effettuato un mocking della classe `ListifyService`.
- Assicurare che la logica di business, implementata nella classe `ListifyService`, gestisca correttamente i dati a runtime. Per garantire un test unitario del service layer, sono state effettuate delle operazioni di mocking per le classi `Repository`.

3.1.2 Descrizione dei test principali

Sono stati scritti test di unità per le seguenti componenti:

- **ViewController**: i test verificano che ogni richiesta restituisca la view corretta e che tutti gli attributi del modello siano presenti e correttamente valorizzati.
- **APIController**: i test verificano lo status code delle risposte, l'eventuale body della risposta e i valori degli attributi di sessione.
- **ListifyService**: i test verificano che i dati vengano gestiti correttamente a runtime. Ad esempio, si controlla che, se viene creato un nuovo utente, questo risulti correttamente presente a runtime, o che l'eliminazione di una lista o di un'attività avvenga come previsto.

3.2 Test di Integrazione

3.2.1 Obiettivi dei test di integrazione

I test di integrazione sono stati realizzati per verificare il corretto funzionamento dell'applicazione, focalizzandosi sull'interazione tra le diverse componenti.

Gli obiettivi principali dei test di integrazione sono:

- Assicurarsi che le richieste inviate all'`APIController` vengano correttamente gestite e che i dati siano elaborati e salvati nel database e a runtime tramite il `Service`.
- Verificare che il `Service` interagisca in modo corretto con il database senza anomalie o comportamenti inattesi.

3.2.2 Descrizione dei test principali

Sono stati scritti test di integrazione per le seguenti componenti:

- **APIController**: a differenza dei test unitari, in questo caso viene simulato l'intero flusso di utilizzo dell'applicazione: un utente si registra, crea delle to-do list, le modifica, e così via. In questo modo, è stata testata la corretta gestione dei dati sia nel database che a runtime.

- **ListifyService**: a differenza dei test unitari, in questo caso i dati vengono effettivamente salvati nel database prima di essere gestiti a runtime. In questo modo è stato possibile testare l'integrazione e il corretto coordinamento tra il service e il database.

Questi test hanno consentito di verificare l'interazione tra il `Controller`, il `Service` e il `database`, simulando un flusso completo di utilizzo dell'applicazione.

4. Conclusione

4.1 Considerazioni sul Pattern MVC

L'adozione del pattern MVC si è rivelata una scelta efficace per lo sviluppo di questo progetto.
Tra i principali vantaggi dell'adozione del pattern MVC si possono evidenziare:

- **Separazione delle responsabilità**: ogni componente (Model, View, Controller) ha un compito ben definito. Questo riduce il rischio di sovrapposizione di responsabilità e rende il codice più leggibile e organizzato.
- **Facilità di manutenzione**: ad esempio, durante il progetto, è stato necessario modificare la logica di business; grazie alla separazione delle responsabilità, è stato possibile effettuare le modifiche senza impattare le altre componenti (Controller, View, o Database).
- **Testabilità**: grazie alla struttura modulare indotta dal pattern e alla possibilità di isolare le dipendenze (ad esempio, utilizzando *Mockito* per il mocking), è stato possibile testare in modo unitario e indipendente ogni componente.

In sintesi, il pattern MVC si è dimostrato particolarmente adatto allo sviluppo di questa applicazione web, poiché ha favorito una netta divisione tra le componenti principali, semplificando la gestione generale del progetto.

4.2 Possibili Sviluppi Futuri

Il progetto presenta diverse possibilità di espansione e miglioramento. Tra i principali sviluppi futuri identificati:

- Implementazione di grafici per misurare la produttività dell'utente, come il numero di attività completate su base settimanale o mensile.
- Introduzione di nuove personalizzazioni per le attività, ad esempio attraverso l'aggiunta di tags o colori.
- Possibilità per gli utenti di condividere una to-do list con altri utenti, permettendo collaborazioni su to-do list comuni.

Questi sviluppi consentirebbero di ampliare le funzionalità dell'applicazione e di migliorare l'esperienza utente, rendendo Listify uno strumento decisamente più completo.