Laurea in informatica - Università di Salerno
Corso di *Ingegneria del Software* - Prof.ssa F.Ferrucci, Prof. F.Palomba

Object Design Document

Riferimento	
Versione	1.0
Data	20/12/2021
Destinatario	Prof.ssa F Ferrucci, Prof. F.Palomba
Presentato da	ALESSIA AMATO, ALFONSO MADDALONI, ANTONIO GIORDANO, ENRIQUE CAMACHO GARCÍA, MARIANNA VUJKO, LUCA MORELLI, GIUSEPPE RAGOSTA, MARCO PALMISCIANO
Approvato da	ALESSIA AMATO, ALFONSO MADDALONI, ANTONIO GIORDANO, ENRIQUE CAMACHO GARCÌA, MARIANNA VUJKO, LUCA MORELLI, GIUSEPPE RAGOSTA, MARCO PALMISCIANO

Revision History

Data	Versione	Descrizione	Autori
15/12/2021	v0.1	Introduzione, stesura paragrafo "Packages"	Enrique Camacho Garcia
16/12/2021	v0.2	stesura paragrafo "Interfaccia delle classi"	Enrique Camacho Garcia
17/12/2021	v0.3	stesura paragrafo "class diagram"	Marianna Vujko, Giuseppe Ragosta
18/12/2021	v0.4	Definizione "design pattern utilizzati"	Antonio Giordano, Alessia Amato, Luca Morelli
20/12/2021	v1.0	Revisione e correzione	tutti

Sommario

1. Introduzione

Il sistema Book Club si prefigge lo scopo di avvicinare le persone all'hobby della lettura offrendo una piattaforma utile a confronto e discussione letteraria. L'idea nasce a causa della sempre minore propensione dei giovani ad avvicinarsi alla lettura nel tempo libero. Grazie a Book Club, la lettura viene resa più coinvolgente, dinamica e immersa nel sociale con il coinvolgimento di altre persone all'interno dei gruppi di lettura, facilitando ulteriormente gli incontri di chi già prima di usare questo sistema organizzava attività di questo genere.

1.1. Object Trade-off

Durante la fase di analisi e di progettazione del sistema abbiamo individuato diversi trade-off che sorgono nuovamente nella fase di Object Design e che, quindi, riporteremo di seguito.

Performance / Memoria: Il sistema deve garantire risposte rapide a discapito della memoria utilizzata. Per questo motivo saranno introdotte delle ridondanze per evitare interrogazioni costose (in termini di performance) a scapito di alcune operazioni di scrittura meno frequenti (per mantenere la coerenza dei dati).

Affidabilità / Tempo di risposta: Il sistema sarà implementato in modo tale da preferire l'affidabilità al tempo di risposta, garantendo la consistenza nella modifica e nella visualizzazione dei valori.

Costi di sviluppo / Tolleranza ai guasti: Il sistema dovrà essere parzialmente funzionante in caso di problemi con una funzionalità a media o bassa priorità, anche a costo di renderle non disponibili per un determinato periodo di tempo. In caso di malfunzionamenti a funzionalità più centrali, il sistema sarà messo in manutenzione fino alla risoluzione del problema.

Di seguito è riportata una tabella che mostra i design goal preferiti nei Trade-Off. Il grassetto indica la preferenza.

Trade-off	
Performance	Memoria
Affidabilità	Tempo di risposta
Tolleranza ai guasti	Costi di sviluppo

1.2. Linee guida per la documentazione delle interfacce

Di seguito sono definite le linee guida a cui un programmatore deve attenersi nell'implementazione del sistema.

1.2.1. Classi e interfacce

Lo standard nella definizione delle classi TypeScript e delle interfacce JavaScript è simile a quello tradizionalmente usato in Java (camel case, a capo dopo i simboli di graffa aperta, …).

1.2.2. File JavaScript

Gli script JavaScript seguono le linee guida definite nei file di configurazione di ESLint nel progetto front-end (.eslintrc.js).

1.2.3. File TypeScript

Gli script JavaScript seguono le linee guida definite nei file di configurazione di ESLint nel progetto back-end (.eslintrc.js).

1.3. Definizioni, acronimi, abbreviazioni

Definizioni

- **Book club** = gruppo di lettura creato da un fondatore
- **Utente** = persona che usa l'applicazione. Può essere:
  - **ospite**, ossia un utente che non ha ancora effettuato l'accesso tramite le credenziali del proprio account
  - **autenticato**, colui che ha effettuato il login sull'applicazione
  - **fondatore**, ossia un utente non ospite che ha creato almeno un book club: spesso si usa in relazione a un book club, per esempio "il fondatore del book club" in base al contesto
  - **partecipante**, ossia un utente non ospite che partecipa ad almeno un book club: spesso si usa in relazione a un book club, per esempio "il/i partecipante/i del book club" in base al contesto

Acronimi

ODD = Object Design Document

CD = Class Diagram

GUI = Graphical User Interface

1.4. Riferimenti

Il documento ODD farà riferimento a:

- Statement of Work;
- Requirements Analysis Document;
- System Design Document.

2. Packages

La struttura del sistema proposto è formata da un package che è costituito da 4 componenti:

- **Gestione utente autenticato**

  I servizi offerti sono riportati nel package creazione bookclub, ricerca libro e invito, i quali faranno uso dei controller: BookclubController, BookController e InviteController. La logica di business sarà gestita dalle classi dei service ovvero: BookclubService, BookService e InviteService. Per poter accedere ai dati presenti nel database usufruiremo delle entità Bookclub, Book e Invite. Le schermate utilizzabili dall'utente autenticato sono afferenti alle funzionalità a cui quest'ultimo può accedere: Catalogo, Ricerca, PaginaCreazione, infoLibro, che vanno a racchiudere le schermate per la creazione di un book club, BachecaSelection sempre disponibile seppur l'utente autenticato non parteciperà a nessun book club, InvitiRicevuti ovvero la schermata dove potranno essere accettati gli inviti ed altre pagine di utilità come ProfilePage, Sicurezza e ChiSiamo

- **Gestione partecipante**

  I servizi offerti sono riportati nel package PDL e bookclubs, i quali faranno uso dei PDLController e BookclubController gestendo la logica di business con le classi service: PDLService e BookclubService. Per poter accedere ai dati presenti nel database usufruiremo delle entità PDL e Bookclub. Il partecipante rispetto all'utente autenticato potrà usufruire della schermata InfoBookClubUser, la quale permetterà la modifica di un PDL di un libro

- **Gestione fondatore**

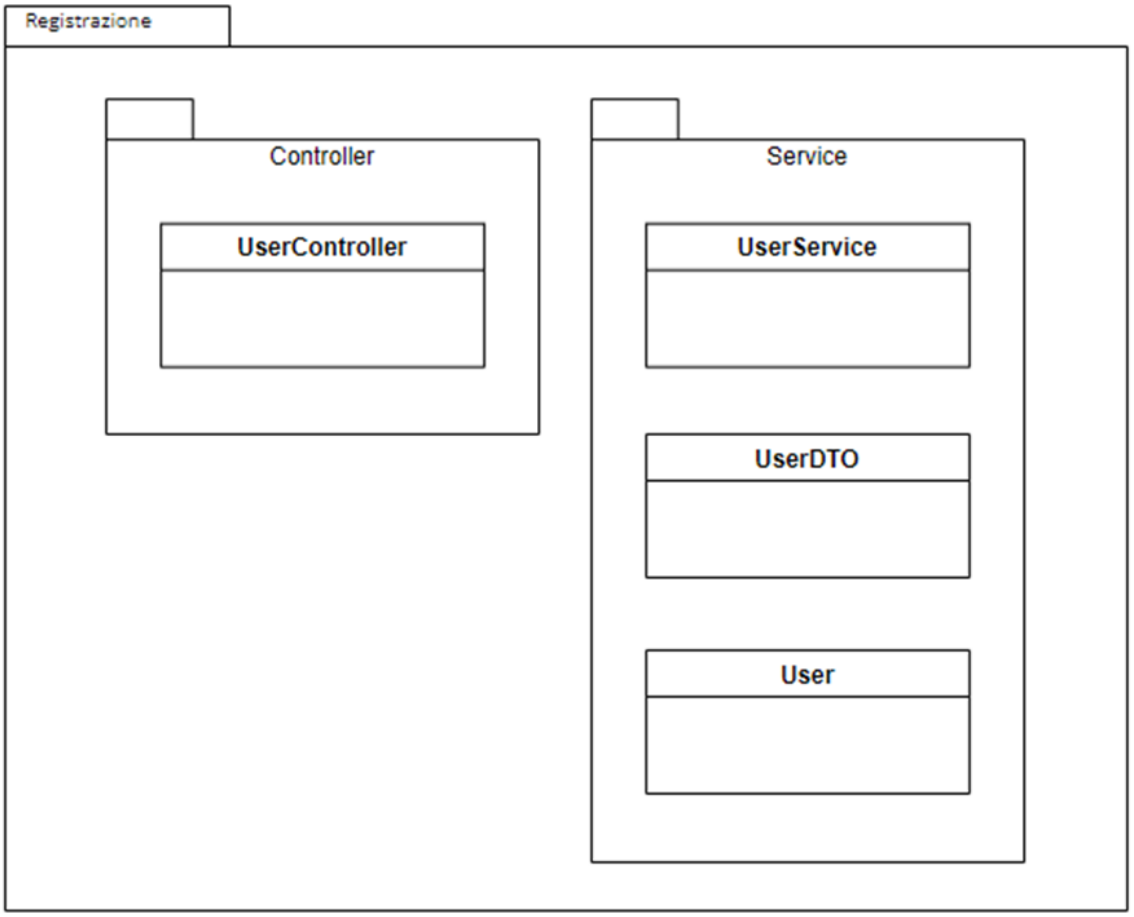
  I servizi offerti sono riportati nel package ODL e invite, i quali faranno uso dei ODLController e InviteController gestendo la logica di business con le classi service: ODLService e InviteService. Per poter accedere ai dati presenti nel database usufruiremo delle entità ODL e Invite. Il fondatore rispetto al partecipante potrà usufruire della schermata InfoBookClubFounder, la quale permetterà la modifica di un PDL di un libro ma anche l'ODL del book club, e alla schermata BookClubInvites,

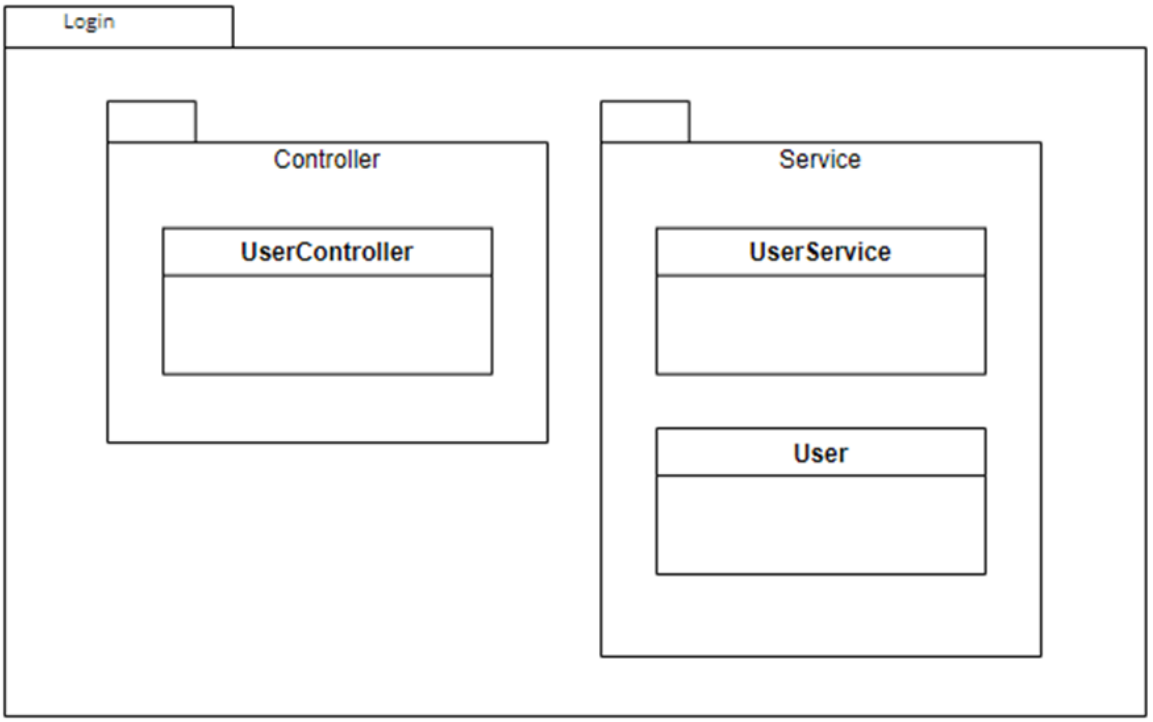
dalla quale potrà controllare lo stato dei suoi inviti e rifiutare gli inviti ancora in attesa

- **Gestione ospite**

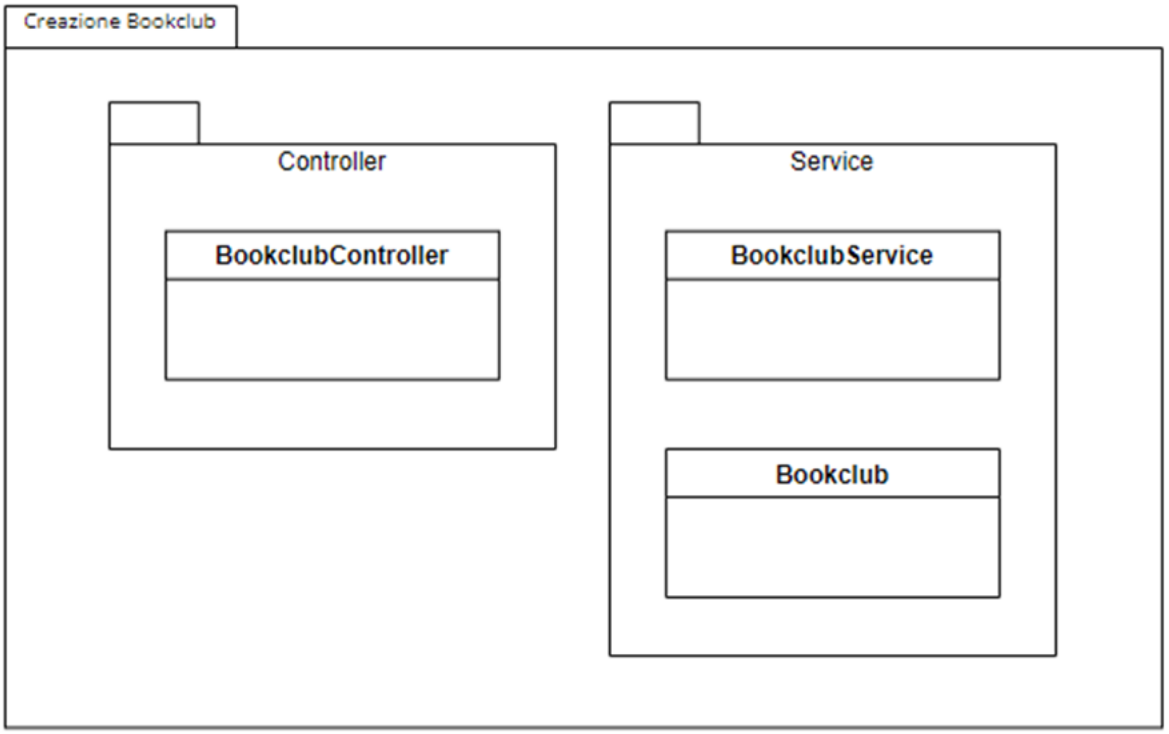
  I servizi offerti sono riportati nel package registrazione e login, i quali faranno uso rispettivamente dello UserController, dello UserService e dello User. Quest'ultimo descrive l'entità dello user all'interno del nostro sistema, il quale ci permette di accedere ai dati all'interno del nostro database. Le schermate con cui l'utente potrà interagire saranno: InitialPage, Registrazione, Login

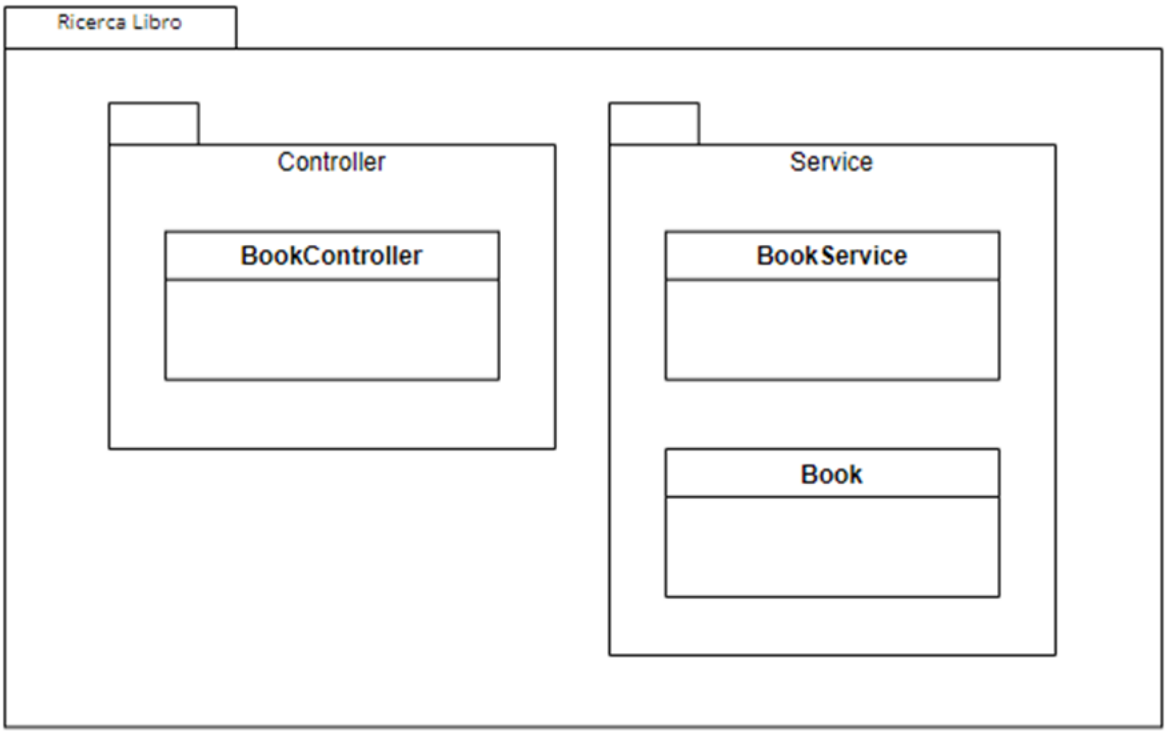
Gestione ospite

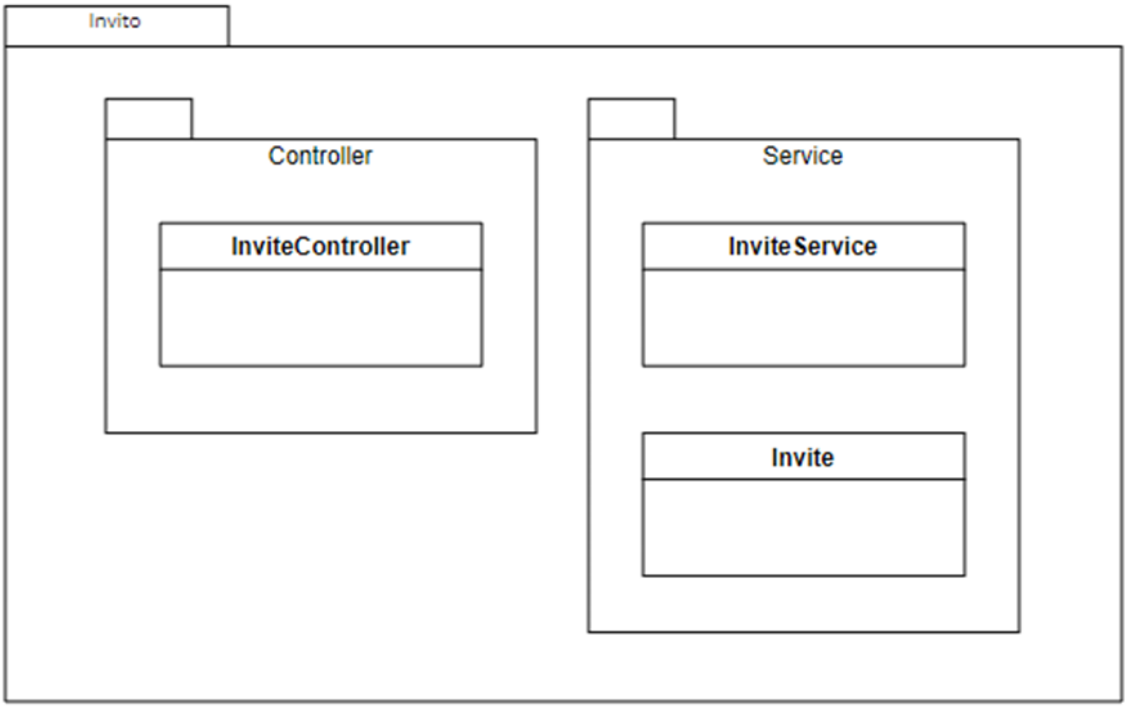




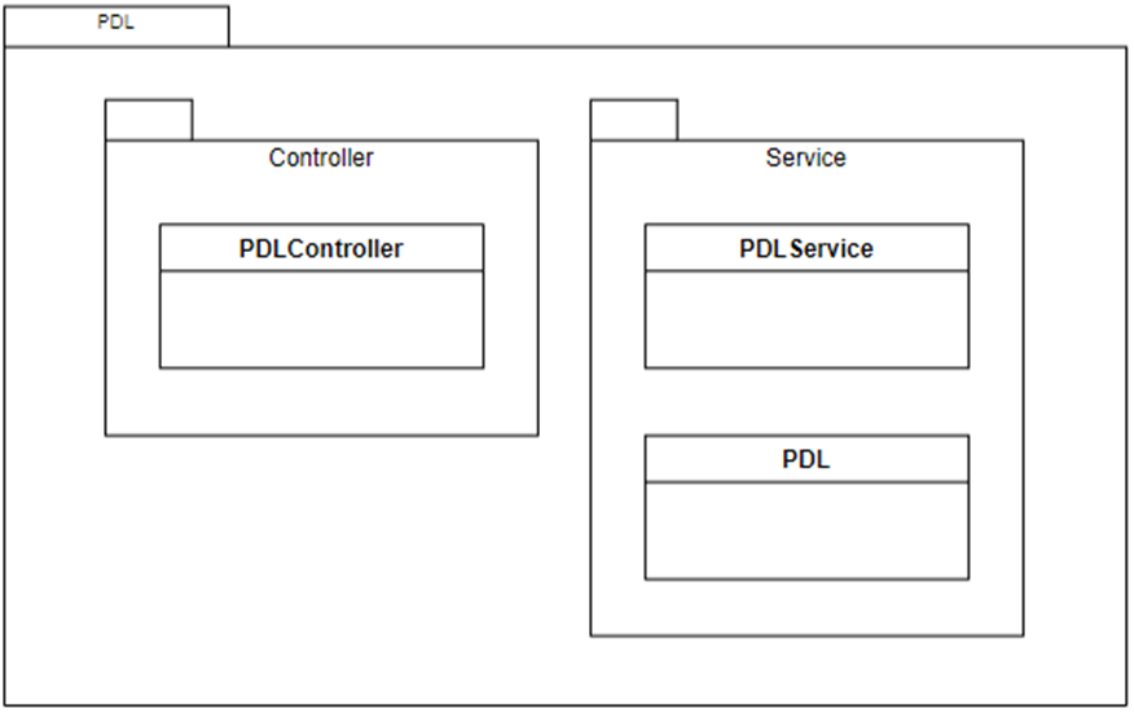
Gestione utente autenticato

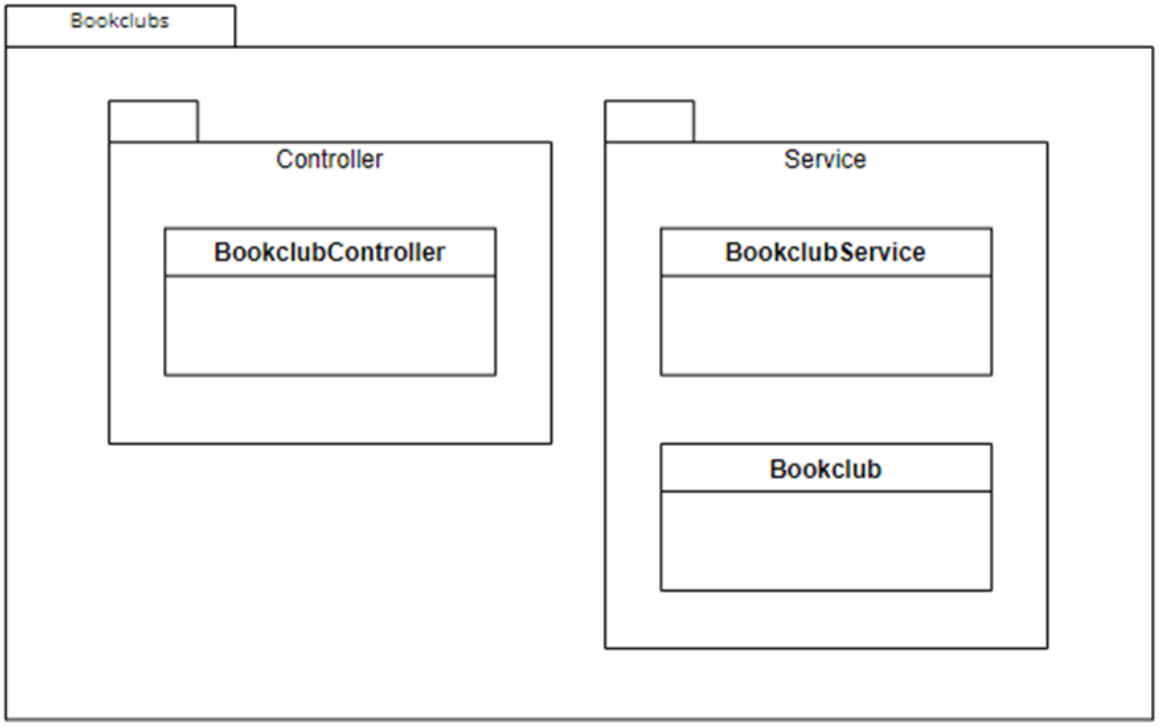



Ricerca Libro
Controller
BookController
Service
BookService
Book

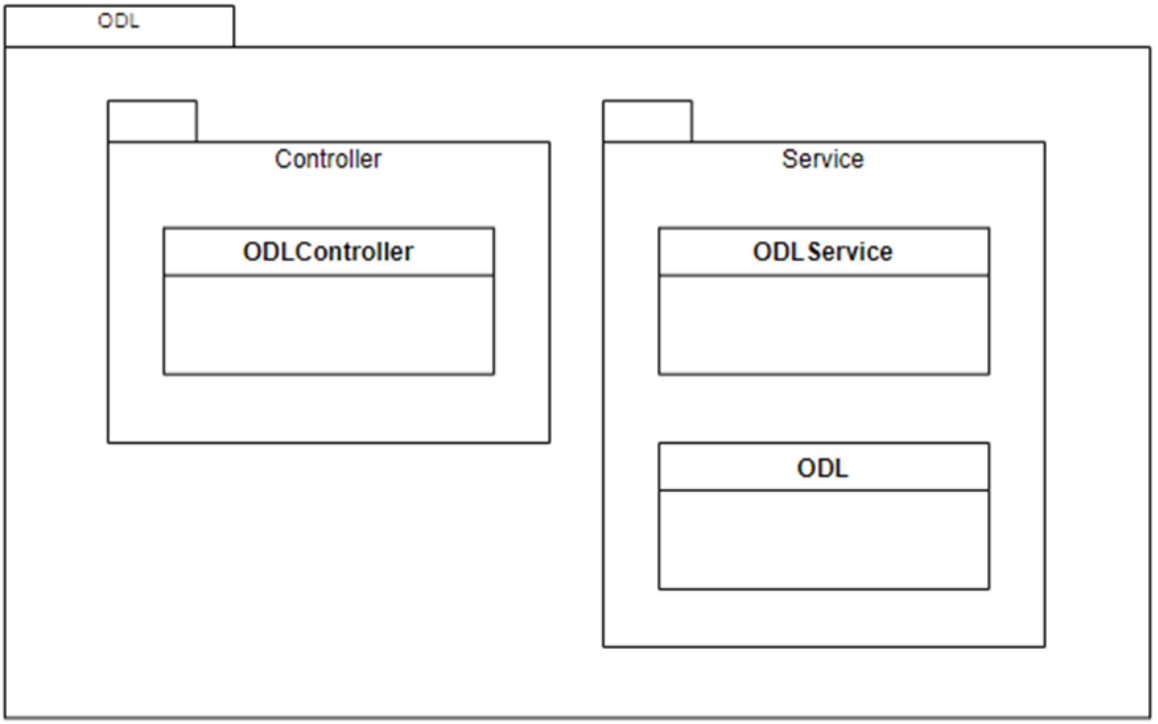

Invito
Controller
InviteController
Service
InviteService
Invite

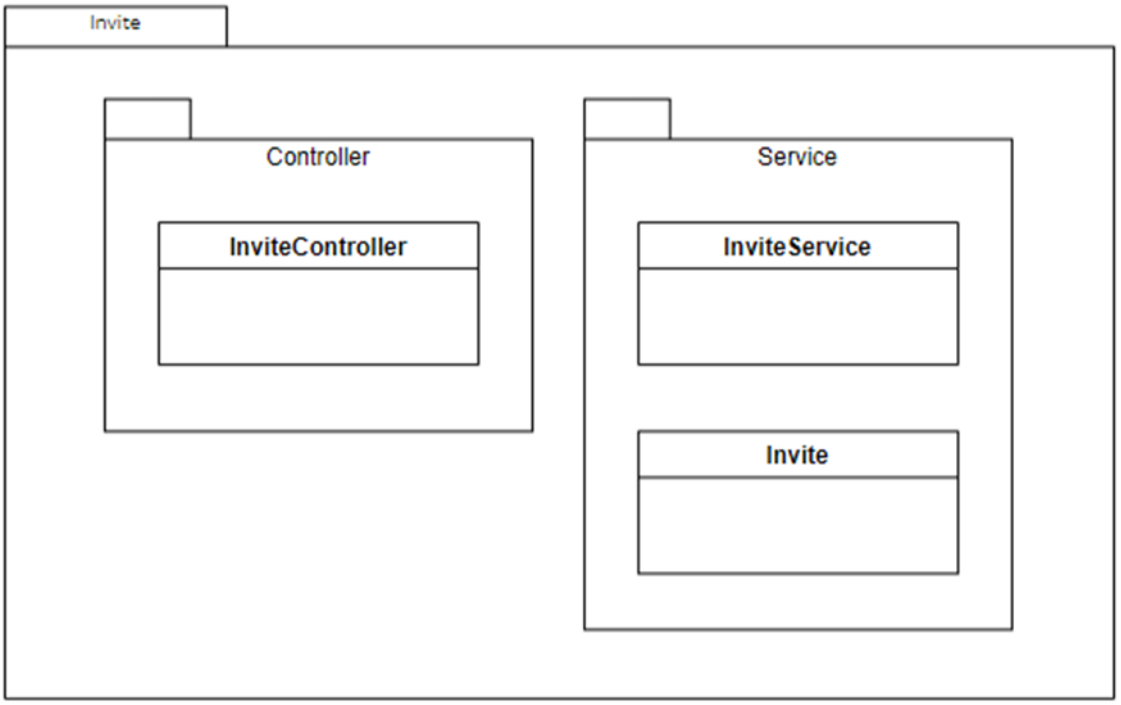
Gestione partecipante





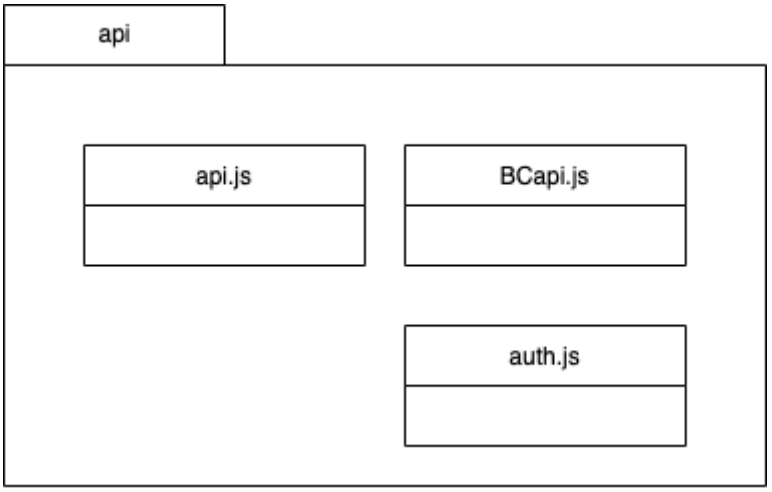
Gestione fondatore



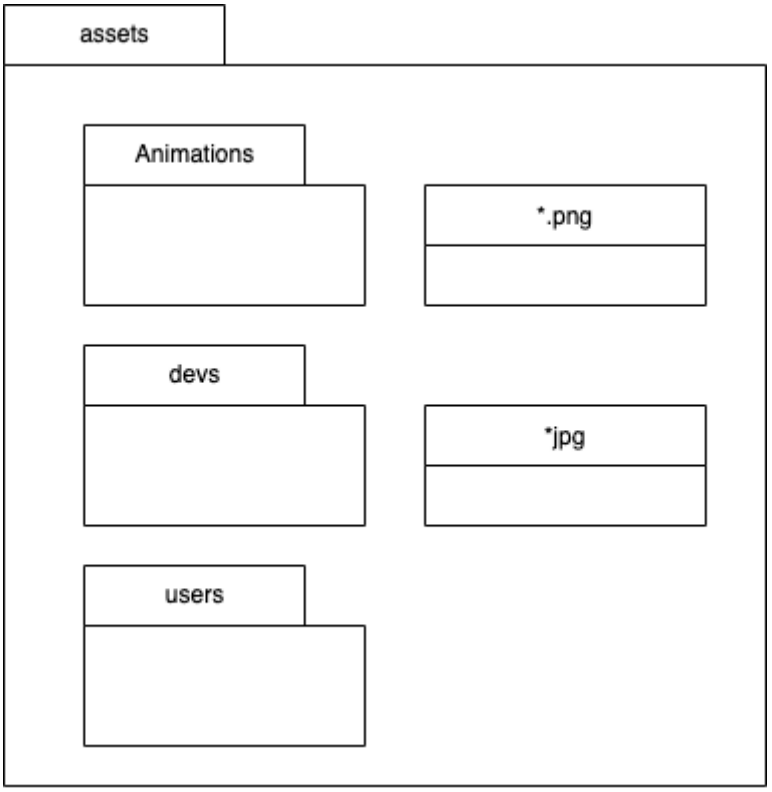


Package app

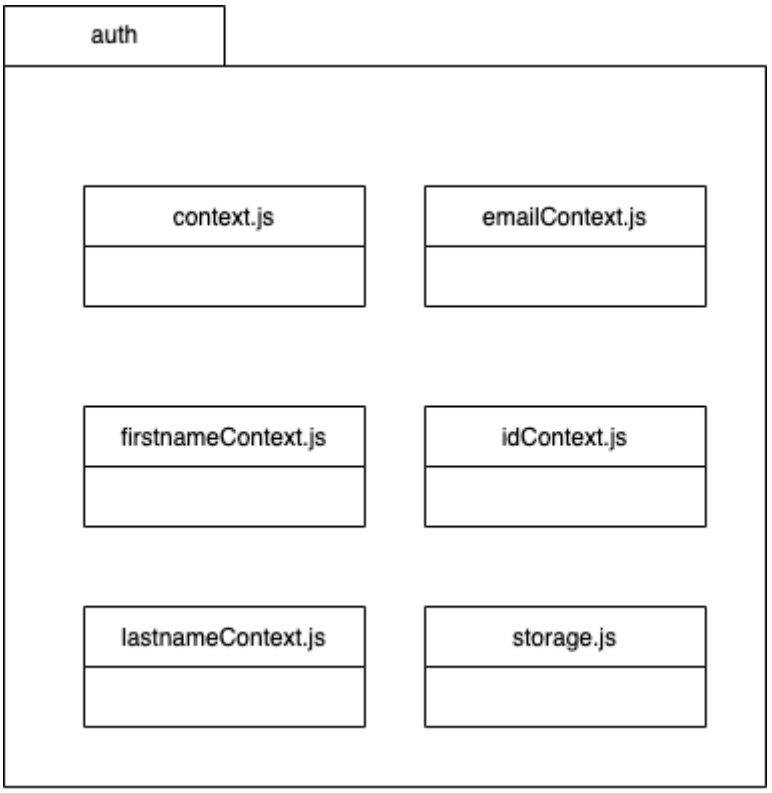
api: cartella che contiene tutti i documenti afferenti all'apiClient. api.js conterrà i metodi invocati successivamente alla registrazione, BCapi.js contiene l'indirizzo richiamato per l'operazione di login e registrazione



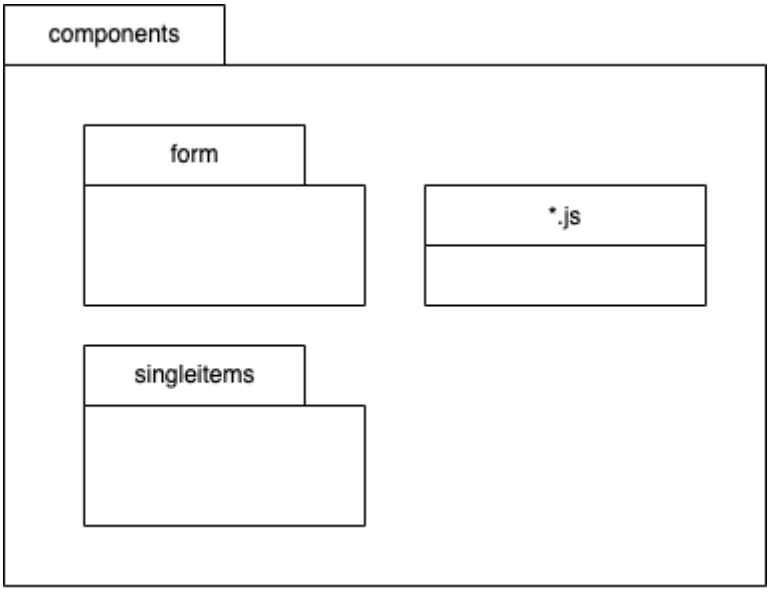
assets: la cartella contiene tutte le foto e animazioni statiche che non richiedono una renderizzazione o una chiamata tramite uri



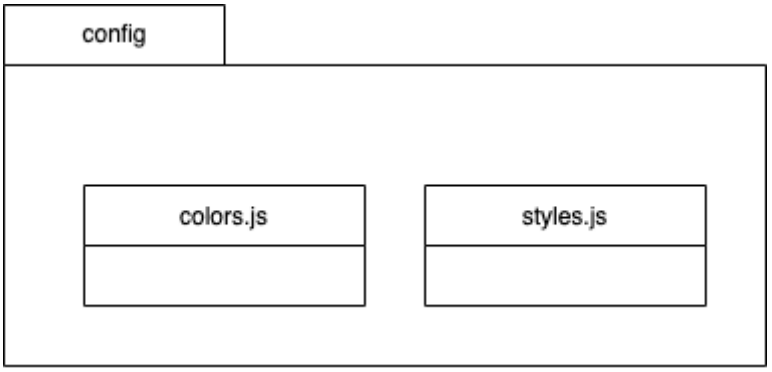
auth: file contenente le variabili salvate tramite useContext, come il token di registrazione, nome, cognome e email. in questi file sono inoltre presenti i metodi per aggiungere, cancellare o modificare uno dei valori sopra riportati



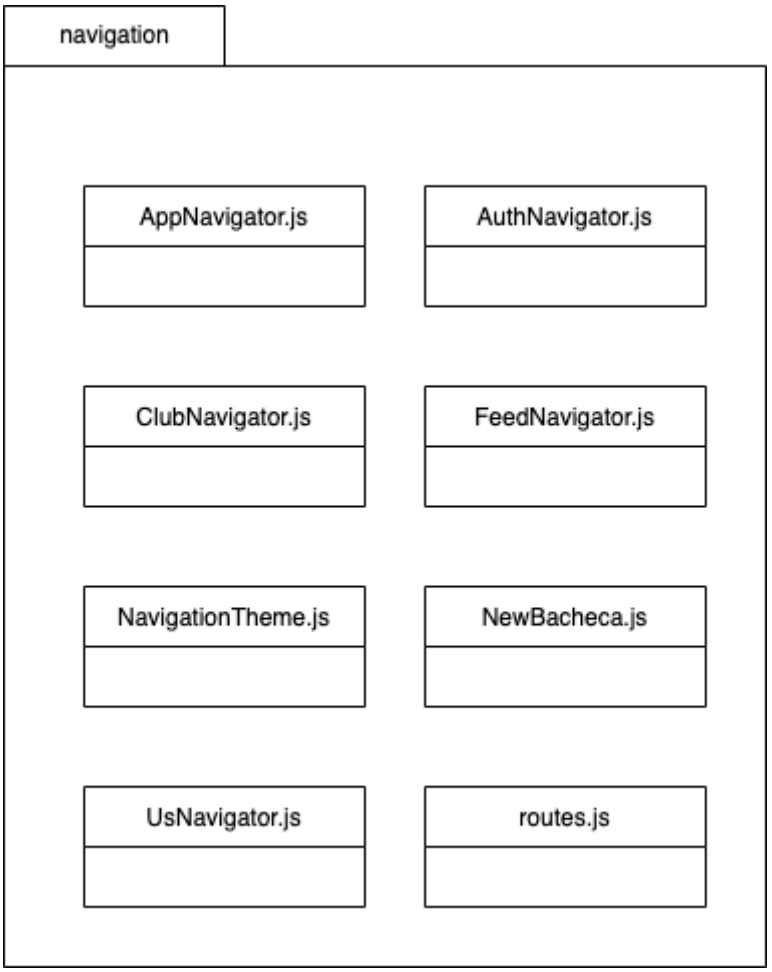
components: in questo file sono contenuti tutte le componenti riutilizzabili che verranno poi richiamate nelle schermate con cui l'utente potrà interagire. É stata inoltre dedicata una singola cartella per tutte le componenti relative ai form



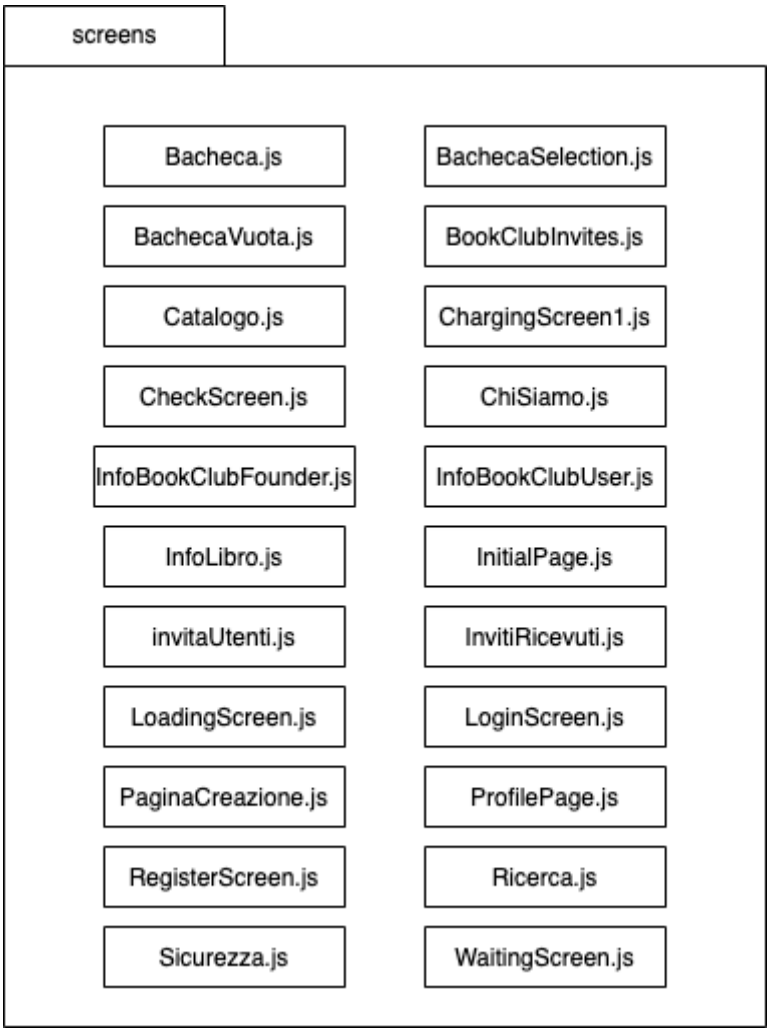
config: in questa cartella sono presenti i file dei colori e dello stile, esportati poi nelle varie componenti. Questi file permettono la modifica a livello globale dello stile dell'applicazione



navigation: in questo file è contenuta tutta la logica di navigazione interna all'applicazione. nel file routes son presenti le varie schermate che permettono la modifica unica e a livello globale di path in caso sia necessario



screens: in questo file sono racchiuse tutte le schermate a cui l'utente potrà accedere manualmente o automaticamente. all'interno di queste schermate saranno a loro volta contenute le componenti contenute nel file "components".



Di seguito viene mostrata la suddivisione del package src a livello back-end il quale contiene una cartella per ogni entità trattata all'interno del nostro sistema, al cui interno troviamo i controller e i service :

- **src**
    - **Book**
    - **Bookclub**
    - **Invite**
    - **Membership**
    - **ODL**
    - **PDL**
    - **ReadSession**
    - **User**
    - **app.module**
    - **main**

Di seguito viene mostrata la suddivisione del package app a livello front-end il quale contiene delle sottocartelle per la suddivisione delle componenti:

- **app**
    - **api/**
    - **assets/**
        - ➢ **animations/**
        - ➢ **devs/**
        - ➢ **users/**
        - ➢ ***.png**
        - ➢ ***.jpg**

- **auth/**
- **components/**
  - **forms/**
  - **singleItems/**
  - ***.js**
- **config/**
- **navigation/**
- **screens/**

3. Interfacce delle classi

Di seguito sarà illustrata la suddivisione del sistema nelle sue componenti con le interfacce di ciascun servizio offerto.

Gestione Ospite:

Servizio di registrazione

Nome Classe	UserService
Descrizione	Questa classe permette di gestire le operazioni relative alla registrazione
Metodi	+createUser(user UserDto):
Invariante	

Nome Metodo	+createUser(user UserDto)
Descrizione	Questo metodo consente di registrare un ospite
Pre-Condizione	**context:** UserService::createUser(user UserDto) **pre:** email.length(6,30) firstName.length(2,20) lastName.length(2,20) password.length(8,32)
Post-Condizione	**context:** UserService:: new User(user)

Servizio di login

Nome Classe	UserService
Descrizione	Questa classe permette di gestire le operazioni relative al login
Metodi	+retrieveTokenByCredential(email string, password string)
Invariante	

Nome Metodo	+retrieveTokenByCredential(email string, password string)
Descrizione	Questo metodo permette ad un ospite di fare il login
Pre-Condizione	**context:** UserService:: retrieveTokenByCredential(email string, password string) **pre:** email.length(6,30) password.length(8,32)
Post-Condizione	**contex:** UserService:: string

Gestione utente autenticato:

Servizio di creazione di un bookclub

Nome Classe	BookclubService
Descrizione	Questa classe permette di gestire le operazioni relative ad un bookclub
Metodi	+createBookclub(isbn string, bookclubName string, founder string)
Invariante	

Nome Metodo	+createBookclub(isbn string, bookclubName string, founder string)
Descrizione	Questo metodo permette ad un utente autenticato di poter creare un bookclub
Pre-Condizione	**contex:** BookclubService::createBookclub(isbn string, bookclubName string) **pre:** bookclubName.length()>1
Post-Condizione	**contex:** BookclubService::bookclub

Servizio di ricerca libro

Nome Classe	BookService
Descrizione	Questa classe permette di gestire le operazioni relative alla ricerca di un libro
Metodi	+findBooksByTitle(bookTitle string)
Invariante	

Nome Metodo	+findBooksByTitle(bookTitle string)
Descrizione	Questo metodo permette ad un utente autenticato di poter ricercare un libro tramite il titolo
Pre-Condizione	**contex:** BookService:: +findBooksByTitle(bookTitle) **pre:** bookTitle.length()>=2
Post-Condizione	**contex:** UserService:: Book[]

Servizio di gestione invito

Nome Classe	InviteService
Descrizione	Questa classe permette di gestire le operazioni relative agli inviti per un utente autenticato
Metodi	+getInvites(user string) +acceptInvite(inviteId number, user string) +refuseInvite(inviteId number,user string)
Invariante	

Nome Metodo	+getInvites(user string)
Descrizione	Questo metodo permette ad un utente autenticato di visualizzare la lista degli inviti ricevuti
Pre-Condizione	**contex:** InviteService::getInvites(user) **pre:** result = await this.loadPermissionsByTokenUser(token) if(result!='UNAUTHORIZED') return UnauthorizedException
Post-Condizione	**contex:** InviteService::Invites[]

Nome Metodo	+acceptInvite(inviteId number, user string)
Descrizione	Questo metodo permette ad un utente autenticato di accettare un invito ricevuto
Pre-Condizione	**contex:** InviteService::acceptInvite(user) **pre:** result = await this.loadPermissionsByTokenUser(token) if(result!='UNAUTHORIZED') return UnauthorizedException
Post-Condizione	**contex:** InviteService::Invite

Nome Metodo	+refuseInvite(inviteId number, user string)
Descrizione	Questo metodo permette ad un utente autenticato di rifiutare un invito ricevuto
Pre-Condizione	**contex:** InviteService::acceptInvite(user) **pre:** result = await this.loadPermissionsByTokenUser(token) if(result!='UNAUTHORIZED') return UnauthorizedException
Post-Condizione	**contex:** InviteService::Invite

Gestione partecipante:

Servizio del progresso di lettura

Nome Classe	PDLService
Descrizione	Questa classe permette di gestire le operazioni relative al progresso di lettura di un bookclub per un partecipante
Metodi	+addPDL(newPDL number, user string, id number)
Invariante	

Nome Metodo	+addPDL(newPDL number, user string, id number)
Descrizione	Questo metodo permette ad un partecipante di un bookclub di aggiungere un nuovo progresso di lettura.
Pre-Condizione	**contex:** PDLService::addPDL(PDL) **pre:** result = await this.loadPermissionsByToken(token,bookclub) if(result!='UNAUTHORIZED') return UnauthorizedException and PDL>0
Post-Condizione	**contex:** InviteService::PDL

Servizio per la gestione dei bookclubs

Nome Classe	BookclubService
Descrizione	Questa classe permette di gestire le operazioni relative ad un bookclub
Metodi	+findBookclubsInfo(user string)
Invariante	

Nome Metodo	+findBookclubsInfo(user string)
Descrizione	Questo metodo permette ad un partecipante di poter visualizzare tutti i bookclub di cui fa parte
Pre-Condizione	**contex:** BookclubService::findBookclubsInfo(Bookclubs) **pre:** result = await this.loadPermissionsByToken(token) if(result!='UNAUTHORIZED') return UnauthorizedException
Post-Condizione	**contex:** BookclubService::bookclubs[]

Gestione fondatore:

Servizio dell'obiettivo di lettura

Nome Classe	ODLService
Descrizione	Questa classe permette di gestire le operazioni relative all'obiettivo di lettura di un bookclub
Metodi	+checkStatus(bookclub number) +createODL(bookclub number, milestone number) +updateODL(bookclub number, milestone number) +checkODLStatus(bookclub number)
Invariante	

Nome Metodo	+checkStatus(bookclub number)
Descrizione	Questo metodo permette di conoscere, quando una nuova milestone deve esser inserita, se un obiettivo di lettura deve essere creato oppure aggiornato
Pre-Condizione	**contex:** ODLService::checkStatus(bookclub number) **pre:** result = await.this.loadPermissionsByTokenFounder(token) if(result=='AUTHORIZED') return checkStatus()
Post-Condizione	**contex:** ODLService::checkStatus()

Nome Metodo	+createODL(bookclub number, milestone number)
Descrizione	Questo metodo permette ad un fondatore di un bookclub di creare un obiettivo di lettura
Pre-Condizione	**contex:** ODLService::createODL(newODL) **pre:** result = await.this.loadPermissionsByTokenFounder(token) if(result=='AUTHORIZED') return createODL() and checkStatus(Id)=='CREATE'
Post-Condizione	**contex:** ODLService::ODL

Nome Metodo	+updateODL(bookclub number, milestone number)
Descrizione	Questo metodo permette ad un fondatore di un bookclub di aggiornare un obiettivo di lettura
Pre-Condizione	**contex:** ODLService::updateODL(bookclub number, milestone number) **pre:** result = await.this.loadPermissionsByTokenFounder(token) if(result=='AUTHORIZED') return updateODL() and checkStatus(Id)=='UPDATE'
Post-Condizione	**contex:** ODLService::ODL

Nome Metodo	+checkODLStatus(bookclub number)
Descrizione	Questo metodo viene utilizzato per conoscere quali membri di un bookclub hanno raggiunto l'obiettivo di lettura corrente
Pre-Condizione	**contex:** ODLService::checkODLStatus(bookclub number)
Post-Condizione	**contex:** ODLService::Bookclub_Membership[]

Servizio di gestione degli inviti

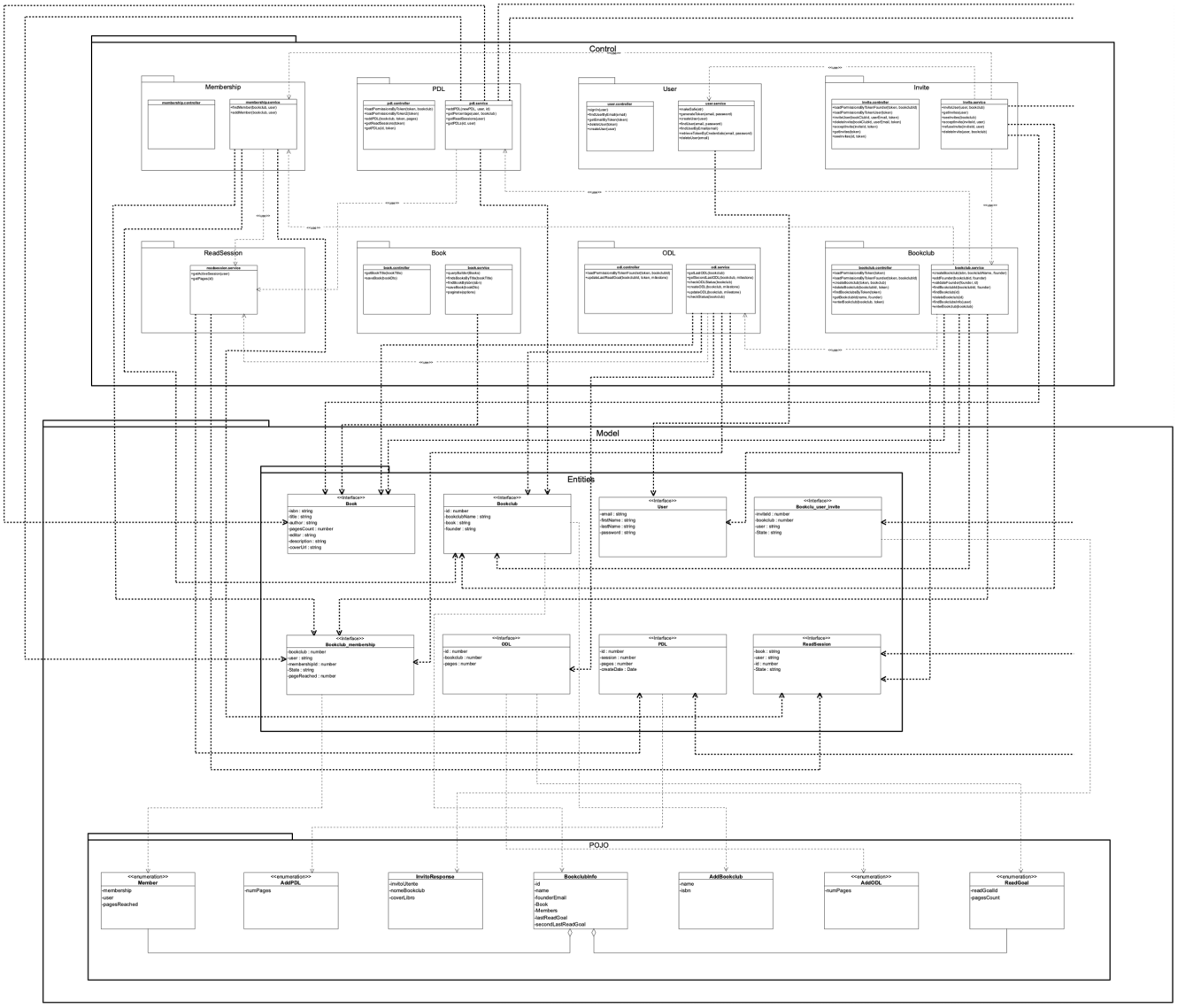
Nome Classe	InviteService
Descrizione	Questa classe permette di gestire le operazioni relative agli inviti di un bookclub da parte di un fondatore
Metodi	+inviteUser(user string, bookclub number) +seeInvites(id number) +deleteInvite(user string, bookclub number)
Invariante	

Nome Metodo	+inviteUser(user string, bookclub number)
Descrizione	Questo metodo permette ad un fondatore di un bookclub di invitare dei partecipanti
Pre-Condizione	**contex:** InviteService::inviteUser(newInvite) **pre:** result = await.this.loadPermissionsByTokenFounder(token) if(result==’AUTHORIZED’) return inviteUser()
Post-Condizione	**contex:** InviteService:: Invite

Nome Metodo	+seeInvites(id number)
Descrizione	Questo metodo permette ad un fondatore di un bookclub di visualizzare tutti gli utenti invitati
Pre-Condizione	**contex:** InviteService::seeInvites(id number) **pre:** result = await.this.loadPermissionsByTokenFounder(token) if(result=='AUTHORIZED') return seeInvites()
Post-Condizione	**contex:** InviteService:: Invite[]

Nome Metodo	+deleteInvite(user string, bookclub number)
Descrizione	Questo metodo permette ad un fondatore di un bookclub di eliminare un invito mandato ad un partecipante
Pre-Condizione	**contex:** InviteService::deleteInvite() **pre:** result = await this.loadPermissionsByTokenFounder(token) if(result=='AUTHORIZED') return deleteInvites()
Post-Condizione	**contex:** InviteService::{ "raw":[], "affected":1}

4. Class Diagram



5. Design pattern

Singleton pattern

Il singleton pattern, utilizzato per avere una singola istanza di una classe in un determinato momento, presente nella connessione al database ma anche nella suddivisione dei moduli, dei service e delle repository delle varie componenti, ha permesso la serializzazione di operazioni di modifica o di accesso a componenti che, in presenza di una parallelizzazione avrebbero portato a delle complicazioni. Trattandosi di un'applicazione multi-utente che però non registrerà un elevato carico di utenti, l'utilizzo del singleton pattern garantisce l'esecuzione di transazioni, necessarie per il corretto utilizzo dell'applicazione, senza inficiare troppo l'efficienza dell'applicazione stessa. Le operazioni interessate dall'utilizzo del singleton pattern vanno dalla creazione di book club, definizioni di obiettivi di lettura e progressi di lettura (operazioni centrali dell'applicazione) fino al salvataggio dei dati in modo permanente sul database

Facade pattern

Il facade pattern, implementato tramite delle api, permette di utilizzare delle interfacce che nascondono la struttura interna dei metodi definiti nel back-end all'apiClient nel front-end. Le routes con relativa tipologia di chiamata, rappresenteranno l'unico modo per chiamare i metodi definiti lato back-end da parte del front-end. La scelta di questo design pattern permette di suddividere i due "sottosistemi" e la possibilità di apportare modifiche a livello implementativo in uno di essi senza però dover necessariamente modificare anche l'altro, garantendo basso accoppiamento. Questa necessità è stata dettata dal fatto che, considerando quanto velocemente si evolvono le tecnologie per lo sviluppo di applicazioni, sarà spesso necessario andare a modificare sia il front-end che il back-end, ma in modo del tutto separato.

Bridge pattern

Il bridge pattern, implementato nella visualizzazione della bacheca, permette di dividere l'interfaccia dalla reale implementazione della componente. Le componenti che verranno richiamate, dovranno infatti eseguire operazioni differenti, utilizzando dunque l'ereditarietà per separare le responsabilità tra le componenti. La scelta di quale componente visualizzare verrà fatta in modo automatico, e ciò dipenderà dal numero di

book clubs a cui l'utente partecipa. L'utilizzo del design pattern si è rivelato necessario poiché, dal momento che sia dal punto di vista dell'interfaccia che dal punto di vista implementativo la bacheca contenente book clubs e i metodi ad essa associata potrebbero in un futuro cambiare, è stato ritenuto opportuno separare la bacheca vuota in cui metodi offerti sicuramente non cambieranno.

Proxy pattern

Il proxy pattern, implementato nella visualizzazione degli inviti ricevuti, permette, nel contesto dell'applicazione Book Club, di sostituire in fase iniziale del caricamento della pagina, la visualizzazione degli inviti ricevuti da un utente con degli skeleton. La pagina, che presenta la copertina del libro collegata al book club, il nome di quest'ultimo e lo stato dell'utente nei confronti di quello specifico book club, in presenza di molteplici inviti potrebbe risultare pesante da caricare in modo istantaneo. Sia dal punto di vista estetico dell'interfaccia utente che dal punto di vista funzionale, è stata fatta la scelta di usare questo design pattern per permettere all'utente di visualizzare degli skeleton e non una schermata completamente vuota. La schermata vuota, senza proxy pattern e con una lenta connessione, potrebbe infatti comunicare, in presenza di un utente inesperto, l'assenza di inviti nei confronti dell'utente stesso.

6. Glossario

Di seguito riportiamo la spiegazione di alcuni termini utilizzati nel documento con lo scopo di renderne più agevole la comprensione. Per ulteriori definizioni fare riferimento al paragrafo 1.5.

Sigla/Termine	Definizione
Singleton pattern	È un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.
Facade pattern	È un design pattern che indica un oggetto che permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro, nonché a blocchi di codice complessi.
Bridge pattern	È un design pattern che permette di separare l'interfaccia di una classe (che cosa si può fare con la classe) dalla sua implementazione (come lo fa). In tal modo si può usare l'ereditarietà per fare evolvere l'interfaccia o l'implementazione in modo separato.
Proxy pattern	È un design pattern che fornisce una classe che funziona come interfaccia per qualcos'altro. L'altro potrebbe essere qualunque cosa: una connessione di rete, un grosso oggetto in memoria, un file e altre risorse che sono costose o impossibili da duplicare.