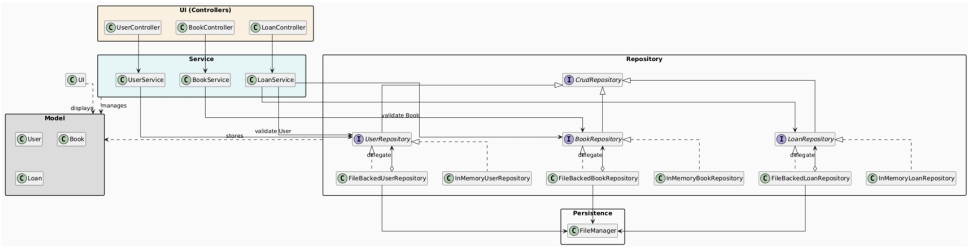
Diagrammi delle Classi e delle Sequenze

Diagrammi delle Classi

Diagramma delle Classi (Interazioni)



Obiettivo: il diagramma delle interazioni delle classi ha lo scopo di mostrare come i diversi componenti del sistema collaborano tra loro durante lo svolgimento delle principali operazioni applicative. Mette in evidenza il modo in cui i vari livelli architetturali (interfaccia utente, servizi, repository, persistenza e modello) comunicano tra loro e si coordinano, chiarendo la distribuzione delle responsabilità e le scelte progettuali adottate per gestire una buona organizzazione del sistema.

Coesione:

- UI (Controllers): gestisce l'interazione con il bibliotecario, raccoglie le richieste, effettua una validazione di base dei dati e li inoltra successivamente ai servizi applicativi.
- Service: coordina la logica applicativa e applica le regole di business, e rappresenta il punto di collegamento tra l'interfaccia utente e i repository
- Repository: Implementa le funzionalità CRUD sulle entità, ed in più implementa delle azioni specifiche e caratteristiche per là entità di riferimento
- Persistence: responsabile della memorizzazione dei dati su file e della loro lettura
- Model: rappresenta le entità del dominio (User, Book, Loan) e contiene i dati e il comportamento di base, senza dipendenze verso altri livelli.

I moduli rispettano un elevato livello di coesione, infatti tutte le classi di un modulo sono legate tra loro. Ad esempio, il modulo repository, contiene diverse interfacce che collaborano per implementare l'accesso ai dati.

Accoppiamento:
Attraverso l'utilizzo della *dependency injection*, abbiamo mantenuto basso il livello di accoppiamento. Come pattern generale, dove opportuno, le classi dipendono solo da interfacce, mentre la reale implementazione è iniettata a runtime da una classe ad-hoc (l'entry point MainApp).

Inoltre, nel caso del FileBackedRepository, abbiamo anche implementato un pattern decorator per far sì che la logica di gestione dei dati in memoria fosse separata dalla parte del salvataggio su file. Questo avviene grazie a un meccanismo di delega dove, da una parte si utilizza un repository In Memory per tutte le operazioni sui

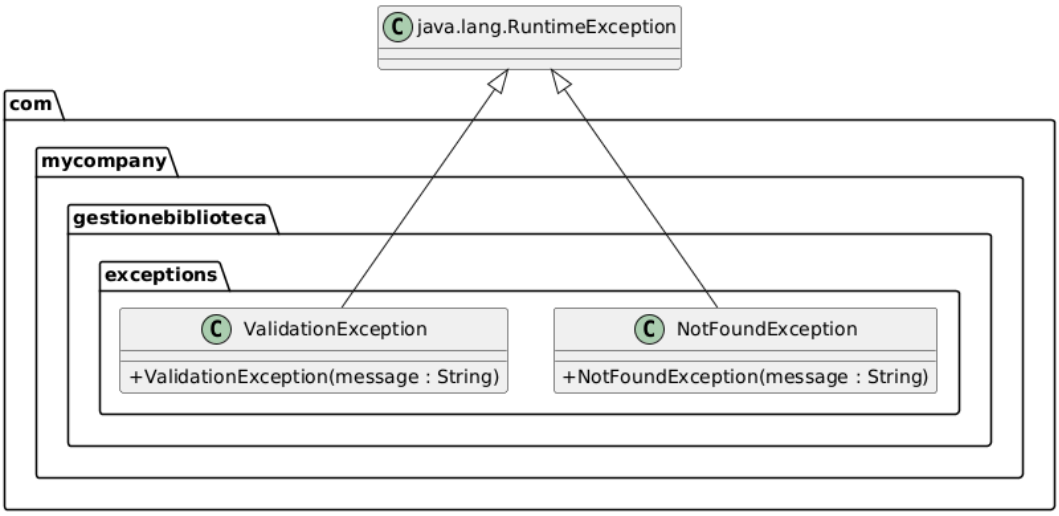
dati (appunto tramite la delega) e dall'altra, solo per le operazioni che effettivamente modificano un dato, viene persistita su File System la collezione di entità.

Principi di buona progettazione:

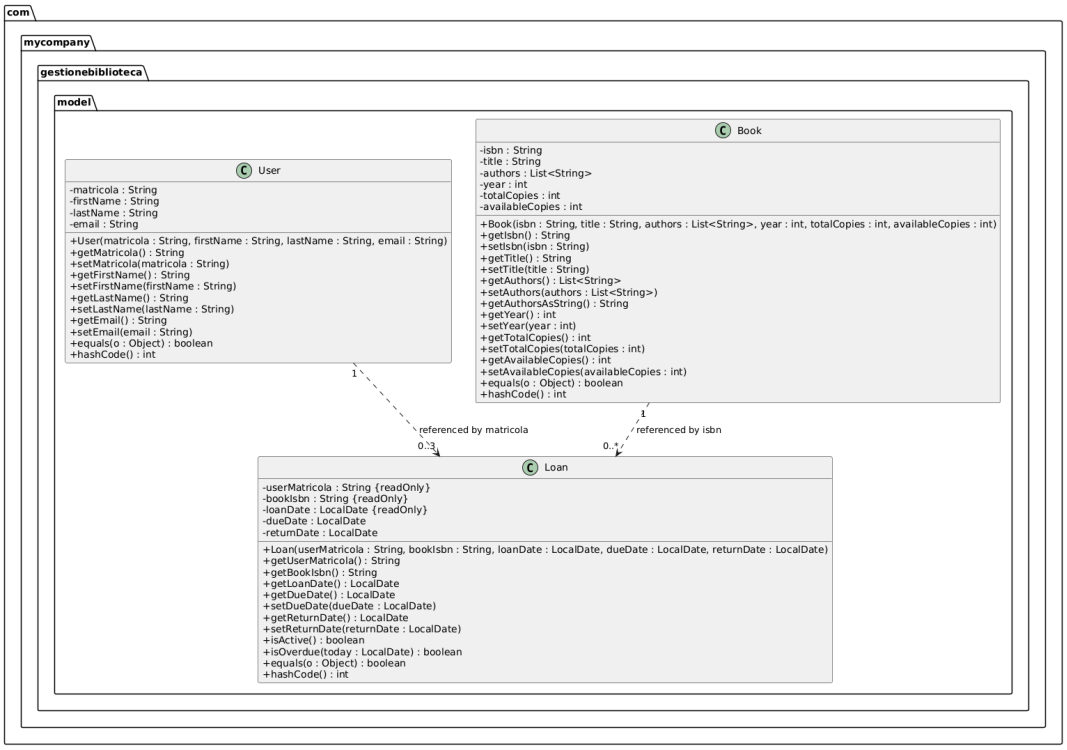
- ogni classe ha una responsabilità unica e ben definita
- i componenti di alto livello dipendono da astrazioni e non da implementazioni concrete
- la gestione dell'interfaccia, della logica applicativa e della persistenza sono separate
- il sistema può essere esteso (ad esempio se si vogliono aggiungere nuove strategie di persistenza) senza modificare il codice esistente

Diagramma delle Classi (Package)

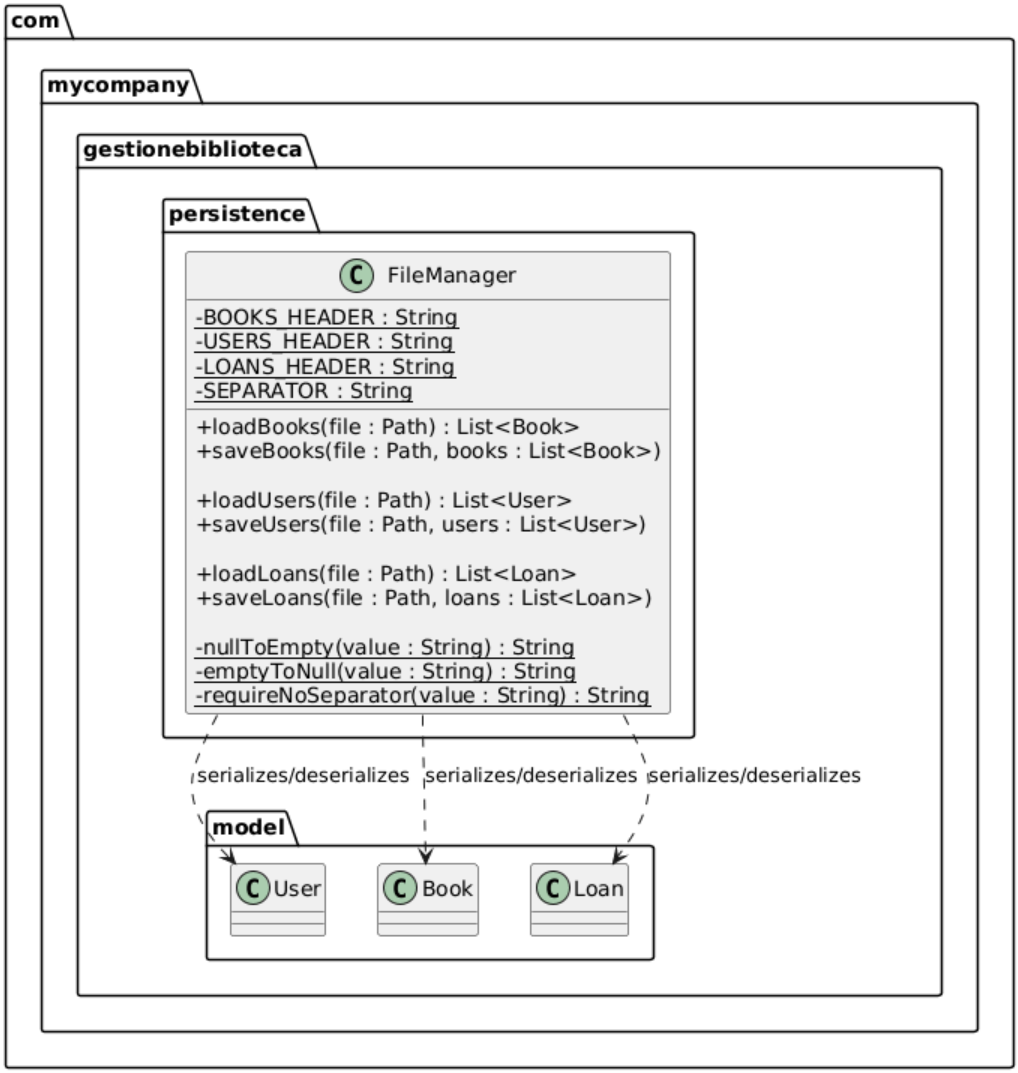
Package: exceptions



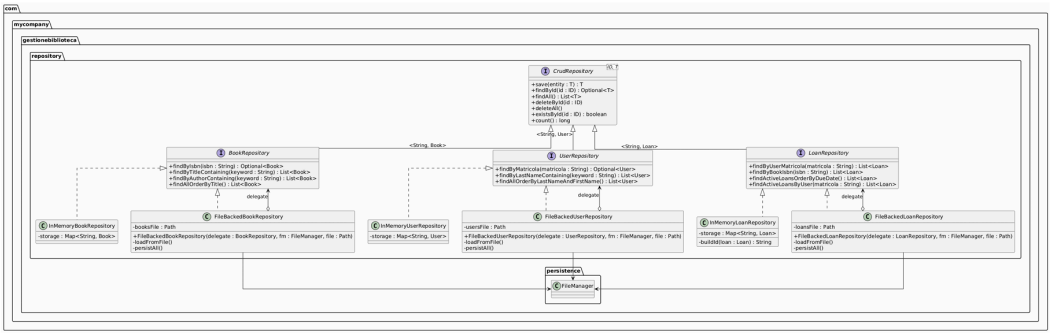
Package: model



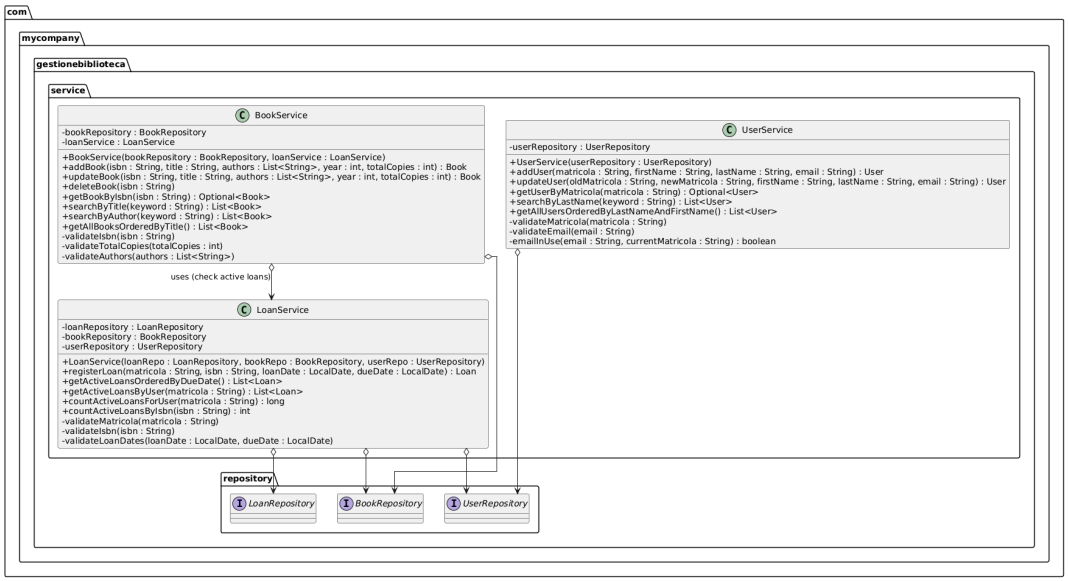
Package: persistence



Package: repository



Package: service



Package: ui (Controllers)

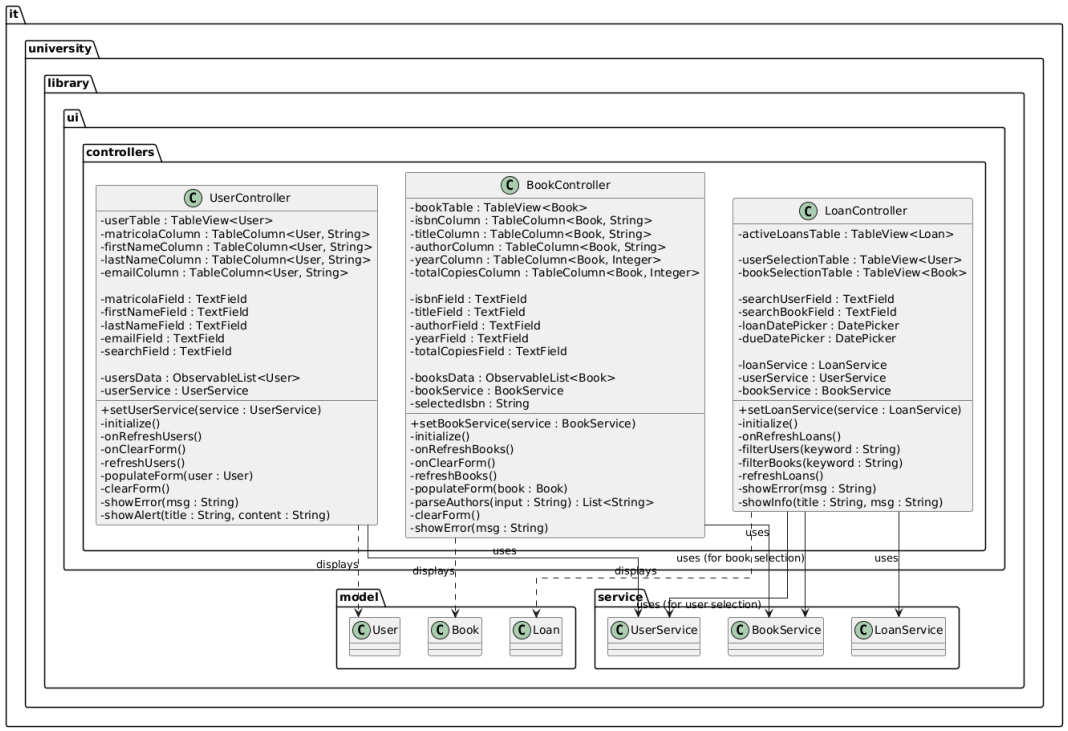
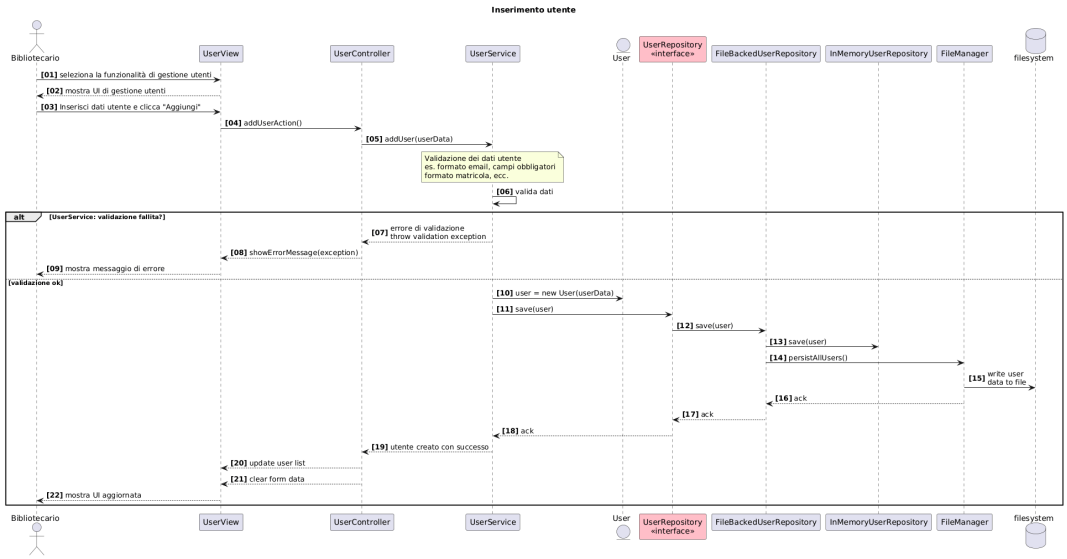


Diagramma delle Sequenze

Inserisci Utente



Obiettivo: il diagramma descrive cosa succede quando il bibliotecario inserisce un nuovo utente dalla UI e preme “Aggiungi”. Lo scenario coinvolge tutti i principali livelli architetturali del sistema e mostra come le responsabilità siano distribuite tra interfaccia, logica applicativa e persistenza.

Il flusso ha inizio quando il bibliotecario accede alla funzionalità di gestione utenti dalla UI. La UserView mostra l’interfaccia dedicata e consente l’inserimento dei dati dell’utente. Una volta completato l’inserimento, il bibliotecario preme il pulsante “Aggiungi”, generando un’azione che viene intercettata dalla vista stessa. La UserView non gestisce direttamente la logica dell’operazione, ma inoltra l’evento all’UserController, il quale ha il compito di ricevere le azioni provenienti dall’interfaccia e di delegare l’elaborazione al livello applicativo giusto. Così la vista rimane concentrata solo sugli aspetti di presentazione e interazione con l’utente.

L’UserController invoca poi il metodo dell’UserService, passando i dati inseriti. Il service rappresenta il cuore della logica applicativa e si occupa innanzitutto della validazione dei dati dell’utente. In questa fase si controllano aspetti come la presenza dei campi obbligatori, il formato della matricola o dell’indirizzo email. La scelta di mettere la validazione nel service permette di centralizzare le regole di business e di mantenerle indipendenti dall’interfaccia utente.

Il frammento alternativo evidenziato nel diagramma distingue due possibili esiti della validazione. Nel caso in cui fallisca, l’UserService segnala l’errore tramite un’eccezione di validazione. Quindi il controllo ritorna all’UserController che si occupa di comunicare l’errore alla UserView, la quale mostra un messaggio al bibliotecario. L’operazione viene interrotta e nessun dato viene salvato. Nel caso in cui la validazione ha esito positivo, l’UserService procede alla creazione dell’oggetto User, che rappresenta l’entità di dominio. Dopo, il service richiede il salvataggio dell’utente attraverso UserRepository, interagendo esclusivamente con l’interfaccia del repository. Questo consente al service di essere indipendente dai dettagli di persistenza.

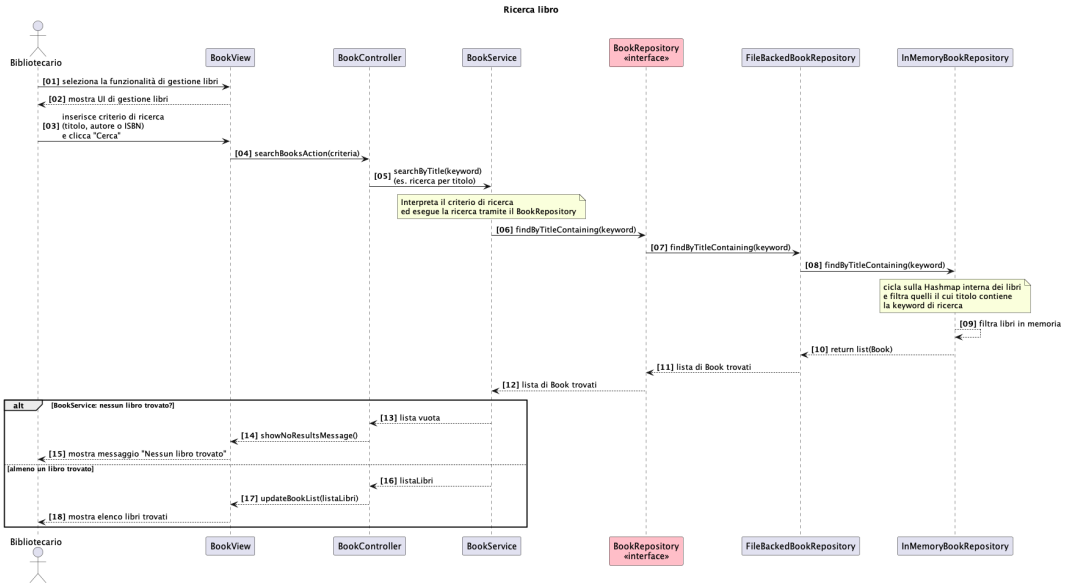
La chiamata di salvataggio viene gestita dal FileBackedUserRepository, che garantisce la persistenza dei dati su file. Questo repository utilizza un meccanismo di delega verso l’InMemoryUserRepository, che si occupa della gestione effettiva dei dati in memoria; il repository file-backed attiva la persistenza su file richiamando il FileManager.

Il FileManager è responsabile della scrittura dei dati sul FileSystem. Una volta completata l’operazione di scrittura, il controllo ritorna progressivamente ai livelli superiori.

A questo punto l’UserController riceve l’esito positivo e aggiorna la UserView, che provvede a mostrare la lista degli utenti aggiornata, a ripulire i campi del modulo di inserimento e a rendere visibile il risultato

dell'operazione. Il flusso si conclude con una UI aggiornata e coerente con lo stato del sistema.

Ricerca Libro



Obiettivo: l'azione di Ricerca Libro ha l'obiettivo di consentire al bibliotecario di individuare uno o più libri presenti nel sistema sulla base di un criterio di ricerca (come titolo, autore o ISBN). Abbiamo scelto l'operazione di ricerca perchè a differenza del caso precedente (Inserisci Utente) è un'operazione di sola lettura. Per questo motivo, quando la chiamata raggiunge il repository, l'implementazione FileBacked non coinvolge il FileManager per l'accesso ai dato su file, ma delega esclusivamente la lettura al repository in-memory, già popolato, sfruttando il meccanismo di delega.

Il flusso ha inizio quando il Bibliotecario seleziona dalla UI la funzionalità di gestione dei libri e inserisce un valore all'interno della barra di ricerca. Il sistema interpreta automaticamente l'input fornito dal bibliotecario e determina il tipo di ricerca più appropriato in base a ciò che è stato inserito.

La BookView intercetta l'evento di ricerca e inoltra l'input al BookController. Sarà il BookService a svolgere il ruolo di analisi del valore inserito, stabilendo quale strategia di ricerca adottare e quale metodo del repository invocare. Il BookService invoca poi il metodo di ricerca appropriato esposto dal BookRepository, interagendo esclusivamente con l'interfaccia del repository e rimanendo quindi indipendente dalla specifica implementazione di persistenza. Nel diagramma è mostrato il caso di una ricerca per titolo, effettuata tramite l'operazione findByTitleContaining.

La richiesta viene gestita dall'implementazione concreta FileBackedBookRepository, che adotta un meccanismo di delega verso un repository in memoria. Poiché l'operazione di ricerca non modifica i dati del sistema, il FileBackedBookRepository non attiva alcuna operazione di lettura o scrittura su file e non coinvolge il FileManager. La chiamata viene invece inoltrata all'InMemoryBookRepository, che contiene la collezione dei libri precedentemente caricata in memoria.

L'InMemoryBookRepository esegue quindi la ricerca scorrendo la struttura dati interna e filtrando i libri che soddisfano il criterio individuato dal service. Al termine dell'operazione, restituisce una lista di oggetti Book che rappresentano i risultati della ricerca. Tale lista viene propagata verso l'alto, ritornando prima al FileBackedBookRepository, poi al BookService e infine al BookController.

Il BookService valuta l'esito della ricerca attraverso un frammento alternativo. Nel caso in cui la lista dei risultati sia vuota, il service segnala l'assenza di libri corrispondenti al criterio di ricerca. Il BookController riceve questa informazione e aggiorna la BookView, che mostra al bibliotecario un messaggio informativo del tipo: *Nessun libro trovato*.

Nel caso in cui uno o più libri siano stati individuati, il BookController passa la lista dei risultati alla view. La BookView aggiorna l'interfaccia mostrando l'elenco dei libri trovati, consentendo al bibliotecario di visualizzare immediatamente l'esito della ricerca.

Il flusso si conclude con l'aggiornamento della UI, senza alterare lo stato del sistema. Questo scenario evidenzia come la separazione delle responsabilità tra i livelli architetturali e l'utilizzo del meccanismo di delega permettano di gestire in modo efficiente le operazioni di sola lettura, mantenendo il sistema modulare, estendibile e facilmente manutenibile.