Documentazione Tecnica Book Recommender

Indice dei Contenuti

Panoramica Architetturale

Book Recommender è un'**applicazione multi-modulo Maven** progettata per raccomandazioni di libri universitari utilizzando **Java 17** e **JavaFX 17.0.12**. Il sistema segue un'**architettura client-server distribuita** con RMI (Remote Method Invocation) per la comunicazione inter-processo.

Principi di Design Chiave

- **Architettura a tre livelli**: Layer Client, Server e Database
- **Pattern Model-View-Controller (MVC)** per componenti UI
- **Pattern Data Access Object (DAO)** per operazioni database
- **Layer Service** per astrazione logica di business
- **Dependency Injection** per accoppiamento debole
- **Strategia di Caching** a molteplici livelli (client e server)
- **Pattern Circuit Breaker** per tolleranza ai guasti

Stack Tecnologico

Backend

Tecnologia	Versione	Scopo
Java	17	Linguaggio principale
PostgreSQL	42.7.2	Database di produzione
H2	2.2.224	Database di testing

Tecnologia	Versione	Scopo
HikariCP	5.1.0	Connection pooling
RMI	Built-in	Comunicazione remota

Frontend

Tecnologia	Versione	Scopo
JavaFX	17.0.12	Framework GUI
FXML	-	UI dichiarativa
Ikonli	12.3.1	Libreria icone
FontAwesome5	8.9	Pacchetto icone

Librerie

Tecnologia	Versione	Scopo
Lombok	1.18.32	Generazione codice
BCrypt	0.10.2	Hashing password
JWT	0.12.5	Autenticazione token
OpenCSV	5.7.1	Parsing CSV
SLF4J	2.0.9	Facciata logging
Logback	1.4.14	Implementazione logging
Micrometer	1.12.0	Raccolta metriche
Resilience4j	2.2.0	Circuit breaker
JUnit	4.13.2	Testing unità
Mockito	5.11.0	Framework mocking

Struttura Modulare

Modulo Condiviso (`src/shared/`)

Contiene modelli comuni, utility e interfacce RMI condivise tra client e server.

Struttura Pacchetti:

```
it.uninsubria.shared/
├── model/              # Modelli di dominio
│   ├── Book.java
│   ├── User.java
│   ├── Library.java
│   ├── Review.java
│   └── Recommendation.java
```

```
├── exception/          # Classi di eccezioni
│   ├── AuthenticationException.java
│   ├── BusinessLogicException.java
│   ├── DatabaseException.java
│   ├── NetworkException.java
│   └── SecurityException.java
│   └── ValidationException.java
├── rmi/                # Interfacce remote
│   ├── UserService.java
│   ├── BookService.java
│   ├── LibraryService.java
│   ├── ReviewsService.java
│   └── SuggestionService.java
│
└── utils/              # Utility condivise
    ├── AppConfig.java
    ├── AppConstants.java
    └── LoggerUtil.java
```

Modulo Server (`src/serverBR/`)

Gestisce logica di business, operazioni database e implementazione server RMI.

Struttura Pacchetti:

```
it.uninsubria.server/
├── bootstrap/           # Inizializzazione applicazione
├── cache/             # Strategie di caching
│   ├── CacheManager.java
│   ├── DistributedCacheManager.java
│   ├── CacheWarmUpService.java
│   └── HybridCacheManager.java
├── dao/               # Data Access Objects
│   ├── UserDAO.java
│   ├── BookDAO.java
│   ├── LibraryDAO.java
│   ├── SuggestionDAO.java
│   ├── ReviewDAO.java
│   └── impl/
│       ├── UserDAOImpl.java
│       ├── BookDAOImpl.java
│       ├── LibraryDAOImpl.java
│       ├── SuggestionDAOImpl.java
│       └── ReviewDAOImpl.java
│
```

```
├── db/                # Gestione database
│   ├── DBInitializer.java
│   ├── ServerController.java
│   ├── DataSeeder.java
│   └── BookRecommenderServer.java
│
├── di/                # Dependency Injection
│   └── ServerDIContainer.java
│
├── exception/         # Eccezioni personalizzate
│   ├── DatabaseException.java
│   ├── AuthenticationException.java
│   ├── ValidationException.java
│   ├── BusinessLogicException.java
│   ├── NetworkException.java
│   └── SecurityException.java
│
├── monitoring/        # Monitoraggio sistema
│   ├── MonitoringService.java
│   └── CircuitBreakerService.java
│
├── rmi/               # Implementazioni servizi RMI
│   └── impl/
│       ├── UserServiceImpl.java
│       ├── BookServiceImpl.java
│       ├── LibraryServiceImpl.java
│       ├── ReviewServiceImpl.java
│       └── SuggestionServiceImpl.java
│
├── service/           # Layer logica di business
│   ├── LibraryServiceCore.java
│   ├── LibraryServiceCoreImpl.java
│   ├── BookServiceCore.java
│   ├── BookServiceCoreImpl.java
│   ├── ReviewsServiceCore.java
│   ├── ReviewsServiceCoreImpl.java
│   ├── SuggestionServiceCore.java
│   ├── SuggestionServiceCoreImpl.java
│   ├── PasswordService.java
│   ├── UserServiceCore.java
│   └── UserServiceCoreImpl.java
│
└── util/              # Utility server
    ├── InputValidator.java
    ├── ConnectionPoolManager.java
    ├── CodiceFiscaleValidator.java
```

```
    ├── JWTUtil.java
    ├── PasswordHashUtil.java
    └── SessionManager.java
```

Modulo Client (`src/clientBR/`)

Applicazione client basata su JavaFX con UI ricca e caching.

Struttura Pacchetti:

```
it.uninsubria.client/
├── cache/              # Caching lato client
│   └── ClientCacheManager.java
│
├── controller/         # Controller UI
│   ├── ControllerBase.java
│   ├── login/
│   │   ├── ControllerLogin.java
│   │   └── ControllerFormForgotPassword.java
│   ├── registration/
│   │   └── ControllerRegistration.java
│   │
│   └── homepage/
│       ├── home/
│       │   ├── ControllerDesign.java
│       │   └── viewHome.java
│       ├── libreria/
│       │   ├── LibraryListController.java
│       │   └── ValutazioneController.java
│       ├── categories/
│       │   ├── CategoriesController.java
│       │   └── CategoryDetailController.java
│       ├── help/
│       │   ├── FAQController.java
│       │   └── ContactFormController.java
│       └── settings/
│           ├── viewSettings.java
│           └── contentsettings/
│               ├── contentModifyProfile.java
│               └── contentChangePasswordSettings.java
│
├── di/                 # Dependency Injection client
│   └── DIContainer.java
│
├── rmi/                # Client RMI
│   ├── RMIClient.java
```

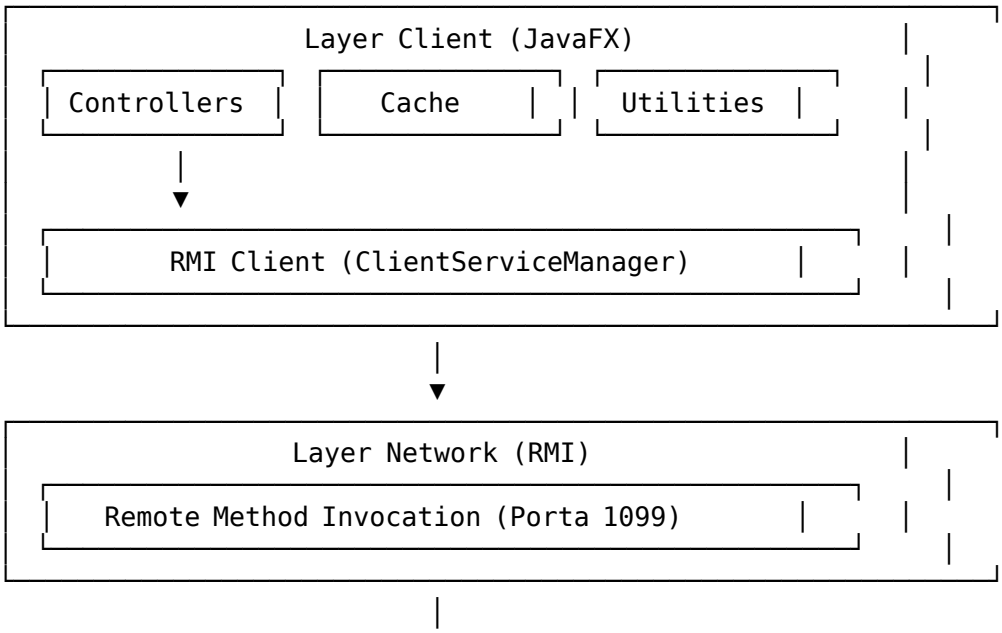
```
│   └── ClientServiceManager.java
│
├── utils/
│   ├── classesUI/     # Utility UI
│   │   ├── LanguageManager.java
│   │   ├── ThemeManager.java
│   │   ├── Navigator.java
│   │   ├── SessionManager.java
│   │   ├── NotificationManager.java
│   │   ├── BookServiceManager.java
│   │   └── ThreadPoolManager.java
│   └── classesLogic/ # Utility business
│       ├── ClassGetImages.java
│       └── VideoUtils.java
│
└── BookRecommenderApp.java # Punto di ingresso applicazione principale
```

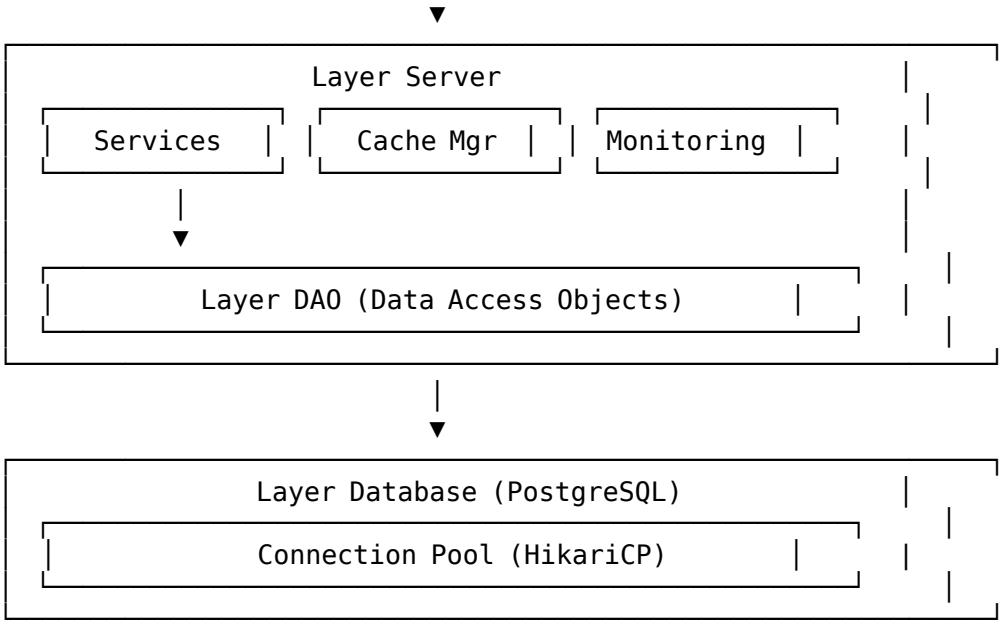
Modulo Launcher (`src/launcher/`)

Launcher GUI per gestione processi e avvio server.

Architettura di Sistema

Architettura a Tre Livelli





Flusso delle Richieste

1. **Azione Utente** → Controller gestisce evento UI
2. **Controller** → Chiama Service Manager (se disponibile in cache)
3. **Service Manager** → Client RMI invia richiesta
4. **Server RMI** → Riceve e instrada a Service Core
5. **Service Core** → Validazione logica di business
6. **DAO** → Operazioni database via HikariCP
7. **Risposta** → Traccia indietro attraverso i layer
8. **Aggiornamento Cache** → Cache client/server aggiornata
9. **Aggiornamento UI** → Thread JavaFX aggiorna interfaccia

Schema UML

UML

Schema Database

Diagramma Entità-Relazione

ER Diagram

Documentazione API

Servizi RMI Remoti

UserService

```
public interface UserService extends Remote {
    boolean registerUser(User user) throws RemoteException, Exception;
    String authenticateUser(String username, String password) throws RemoteExcep
    User getUserByUsernameOrEmail(String identifier) throws RemoteException, Exc
    boolean updateProfileInfo(User user) throws RemoteException, Exception;
    boolean updatePassword(String userID, String newPassword) throws RemoteExcep
    boolean validateSession(String token) throws RemoteException, Exception;
    void invalidateSession(String token) throws RemoteException, Exception;
}
```

BookService

```
public interface BookService extends Remote {
    List<Book> searchGlobally() throws RemoteException, Exception;
    Book getBookById(int bookId) throws RemoteException, Exception;
    List<Book> getBooksByCategory(String category) throws RemoteException, Excep
    List<Book> searchBooks(String query) throws RemoteException, Exception;
}
```

LibraryService

```
public interface LibraryService extends Remote {
    Library createLibrary(Library library) throws RemoteException, Exception;
    List<Library> getUserLibraries(int userId) throws RemoteException, Exception
    boolean addBookToLibrary(int libraryId, int bookId) throws RemoteException,
    boolean removeBookFromLibrary(int libraryId, int bookId) throws RemoteExcept
    List<Book> getLibraryBooks(int libraryId) throws RemoteException, Exception;
}
```

ReviewsService

```
public interface ReviewsService extends Remote {
    Review addReview(Review review) throws RemoteException, Exception;
    List<Review> getBookReviews(int bookId) throws RemoteException, Exception;
    Review getUserBookReview(int userId, int bookId) throws RemoteException, Exc
}
```

SuggestionsService

```
public interface SuggestionsService extends Remote {
    List<SuggestionEntry> getSuggestions(int userId, int libraryId) throws Remot
```

```
    List<Recommendation> getRecommendations(int userId) throws RemoteException,
}
```

Ottimizzazione Performance

Strategia di Caching

Caching Lato Server

- **Hybrid Cache Manager**: Combina caching in-memory e distribuito
- **Cache Warming**: Precarica libri popolari all'avvio
- **Configurazione TTL**: 5 minuti per utenti, 30 minuti per libri
- **Invalidazione Cache**: Automatica su modifiche dati

```
CacheManager.put(userId, key, result, DEFAULT_TTL_MS, category);
```

Caching Lato Client

- **Cache Immagini**: Cache basato su disco per copertine libri
- **Cache Catalogo Libri**: In-memory con persistenza disco
- **Cache Sessioni**: Token JWT e sessioni utente
- **Cache Risorse**: Risorse UI e template

Connection Pooling

Configurazione HikariCP: - **Dimensione Massima Pool**: 10 connessioni - **Minimo Idle**: 2 connessioni - **Timeout Connessione**: 30 secondi - **Timeout Idle**: 10 minuti - **Durata Massima**: 30 minuti

Operazioni Asincrone

Configurazione ThreadPoolManager: - **Pool Thread Critici**: 4 thread per task ad alta priorità - **Pool Thread Background**: 8 thread per task generali - **Pool Caricamento Immagini**: 6 thread per processamento immagini

```
ThreadPoolManager.executeCritical(() -> {
    loadingTask.run();
});
```

Ottimizzazione Database

- **Utilizzo Indici**: Chiavi esterne e colonne frequentemente interrogate
- **Ottimizzazione Query**: Prepared statements, operazioni batch
- **Riutilizzo Connessioni**: Pooling HikariCP

- **Lazy Loading**: Collezioni caricate on-demand

Implementazione Sicurezza

Autenticazione & Autorizzazione

Sicurezza Password

```
// Hashing BCrypt con 12 round
String hashedPassword = BCrypt.hashpw(plainPassword, BCrypt.gensalt(12));

// Verifica
boolean isValid = BCrypt.checkpw(plainPassword, hashedPassword);
```

Gestione Token JWT

```
// Generazione token
String token = JWTUtil.generateToken(userId, username, 30 * 60 * 1000); // 30 mi

// Validazione token
Claims claims = JWTUtil.validateToken(token);

// Invalidazione token
userService.invalidateSession(token);
```

Validazione Input

Validazione Lato Server

```
public static boolean checkEmail(String email) {
    if (email == null || email.isEmpty() || email.length() > 320) return false;
    String emailRegex = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$";
    return email.matches(emailRegex);
}
```

Requisiti Password

- Minimo 8 caratteri
- Almeno una lettera maiuscola
- Almeno una lettera minuscola
- Almeno una cifra
- Almeno un carattere speciale: `!@#$%^&*()_-+=<>?[]{}|;:,.`
- Nessun carattere non valido

Validazione Codice Fiscale

- Esattamente 16 caratteri

- Solo alfanumerici maiuscoli
- Seguono il pattern: **XXXXXXNNXNNXNNNX** di cui X = lettere e N = numeri

Gestione Sessioni

Gestione Timeout

- **Timeout Default**: 15 minuti di inattività
- **Timeout Remember Me**: 30 minuti
- **Auto-logout**: Scadenza sessione automatica

```
TimeoutManager.getInstance().resetTimer();
TimeoutManager.getInstance().setTimeoutMinutes(15);
```

Persistenza Sessioni

- Storage criptato su disco
- Gestione token sicura
- Pulizia automatica al logout

Crittografia

Crittografia Dati

```
// Crittografia AES-256 per dati sensibili
String encrypted = EncryptionService.encrypt(data, key);
String decrypted = EncryptionService.decrypt(encrypted, key);
```

Immagini Profilo

- Memorizzate come BLOB nel database
- Validazione tipo MIME
- Limiti dimensione applicati

Strategia di Testing

Testing Unitario

Framework: JUnit 4.13.2 + Mockito 5.11.0

Struttura Test:

```
src/*/test/java/it/uninsubria/
├── server/
│   ├── service/
│   │   ├── UserServiceCoreTest.java
│   │   └── BookServiceCoreTest.java
```

```
│   ├── cache/
│   │   └── CacheManagerTest.java
│   └── dao/
│       ├── TestUserDAO.java
│       └── TestBookDAO.java
├── client/
│   ├── utils/
│   │   ├── SessionManagerTest.java
│   │   └── LanguageManagerTest.java
│   └── cache/
│       └── ClientCacheManagerTest.java
└── shared/
    └── model/
        └── UserTest.java
```

Database Test: H2 in-memory per testing isolato

Testing di Integrazione

Scenari: - Comunicazione server-client RMI - Transazioni database - Consistenza cache - Workflow end-to-end

Smoke Test:

```java
@Test
public void testUserCoreDelegation() throws Exception {
    TestUserDAO dao = new TestUserDAO();
    UserServiceCore core = new UserServiceCoreImpl(dao);
    User u = User.builder().id("alice").name("Alice").build();
    assertTrue(core.registerUser(u));
}
```

Testing Performance

Load Testing: - Simulazione utenti concorrenti - Stress pool connessioni database - Metriche performance cache

Monitoraggio:

```
Metriche Micrometer:
- Latenza richieste
- Tassi hit cache
- Tempi query database
- Connessioni attive
```

Analisi codice

Analisi Metodi Fondamentali del Sistema BookRecommender

1. **Metodo Fondamentale del Client: start() in BookRecommenderApp.java**

Codice del Metodo:

```java
@Override
public void start(Stage primaryStage) {
    // Aggiungi shutdown hook per cleanup in caso di chiusura forzata
    addShutdownHook();

    // Aggiungi listener per intercettare chiusura finestra
    addWindowCloseListener(primaryStage);

    showSplashThenLaunchUI(primaryStage);
}
```

Spiegazione:

Il metodo `start()` rappresenta il punto di ingresso dell'applicazione JavaFX client. È responsabile dell'inizializzazione completa dell'interfaccia utente e della gestione del ciclo di vita dell'applicazione. Il metodo configura:

- **Shutdown hooks**: Per garantire il cleanup delle risorse anche in caso di chiusura forzata
- **Window close listeners**: Per gestire correttamente la chiusura della finestra principale
- **Splash screen e UI principale**: Coordina la sequenza di avvio con splash screen e caricamento progressivo

Complessità Algoritmica:

- **Tempo**: O(1) - operazioni sequenziali costanti
- **Spazio**: O(1) - nessun uso significativo di memoria aggiuntiva
- **Perché fondamentale**: È il metodo principale che orchestra tutto il flusso di avvio dell'applicazione client, gestendo sia il caricamento iniziale che la terminazione pulita.

2. Metodo Fondamentale del Server: bootstrapHeadless() in ServerApp.java

Codice del Metodo:

```
private static void bootstrapHeadless() {
    try {
        it.uninsubria.server.monitoring.MonitoringService.getInstance();
        logger.info("Monitoring service initialized");

        // Load database configuration from environment variables for security
        String host = System.getenv("DB_HOST");
        if (host == null || host.trim().isEmpty()) {
            host = "localhost"; // fallback for development
        }

        String dbName = System.getenv("DB_NAME");
        if (dbName == null || dbName.trim().isEmpty()) {
            dbName = "projectb"; // fallback for development
        }

        String user = System.getenv("DB_USER");
        if (user == null || user.trim().isEmpty()) {
            throw new RuntimeException("DB_USER environment variable is required
        }

        String password = System.getenv("DB_PASSWORD");
        if (password == null || password.trim().isEmpty()) {
            throw new RuntimeException("DB_PASSWORD environment variable is requ
        }

        it.uninsubria.server.db.DBInitializer.initialize(host, dbName, user, pas
        BookRecommenderServer.startServer();

        // Initialize cache warm-up after server is ready
        try {
            it.uninsubria.server.cache.CacheWarmUpService.warmUpCache(
                it.uninsubria.server.di.ServerDIContainer.getBookCore()
            );
        } catch (Exception e) {
            logger.warning("Cache warm-up failed to initialize: " + e.getMessage
        }

        logger.info("Headless server started with enhanced monitoring and cachin

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
```

```
            logger.info("Shutting down services...");
            it.uninsubria.server.cache.HybridCacheManager.shutdown();
            it.uninsubria.server.cache.CacheWarmUpService.shutdown();
            try {
                BookRecommenderServer.stopServer();
            } catch (Exception e) {
                logger.severe("Error stopping server: " + e.getMessage());
            }
            logger.info("Services shutdown completed");
        }));

        // Keep the server running
        while (true) {
            Thread.sleep(1000);
        }
    } catch (Exception e) {
        logger.severe("Server startup failed: " + e.getMessage());
        System.exit(1);
    }
}
```

Spiegazione:

Il metodo `bootstrapHeadless()` è il cuore dell'avvio del server in modalità headless (senza interfaccia grafica). Coordina l'inizializzazione completa del server RMI, inclusi:

- **Monitoring service**: Per il monitoraggio delle prestazioni
- **Configurazione database**: Caricamento sicuro delle credenziali da variabili d'ambiente
- **Inizializzazione DB**: Setup del database PostgreSQL/H2
- **Avvio server RMI**: Registrazione dei servizi remoti
- **Cache warm-up**: Precaricamento dei dati più utilizzati
- **Shutdown hooks**: Gestione pulita della terminazione

Complessità Algoritmica:

- **Tempo**: O(1) - operazioni di inizializzazione sequenziali, anche se alcune (come cache warm-up) potrebbero essere O(n) dove n è il numero di libri
- **Spazio**: O(1) - risorse allocate sono indipendenti dall'input
- **Perché fondamentale**: È il metodo che rende operativo l'intero backend del sistema, coordinando database, cache, e servizi RMI.

3. Metodo Fondamentale del Launcher: onRunServer() in LauncherApp.java

Codice del Metodo:

```
private void onRunServer() {
    if (currentEnvStatus == null || !currentEnvStatus.allPrerequisitesMet) {
        showAlert(LauncherLanguageManager.getString("launcher.alert.prerequisite
                  LauncherLanguageManager.getString("launcher.alert.prerequisit
        return;
    }

    if (serverRunning) {
        showAlert(LauncherLanguageManager.getString("launcher.alert.server.runni
                  LauncherLanguageManager.getString("launcher.alert.server.runn
        return;
    }

    appendToStatus(LauncherLanguageManager.getString("launcher.status.server.sta
    serverRunning = true;
    serverConnected = false; // Server not yet connected
    updateButtonsState(currentEnvStatus != null && currentEnvStatus.allPrerequis

    // Show progress indicator
    Platform.runLater(() -> {
        serverProgressIndicator.setVisible(true);
        serverProgressIndicator.setProgress(-1); // Indeterminate progress
    });

    // Run server startup in background senza aspettare che finisca
    CompletableFuture.runAsync(this::startServerProcessAsync)
        .thenRun(() -> {
            javafx.application.Platform.runLater(() -> {
                serverProgressIndicator.setVisible(false);
                serverProgressIndicator.setProgress(0); // Reset progress
                serverConnected = true; // Server is now connected to DB and RMI
                appendToStatus(LauncherLanguageManager.getString("launcher.statu
                appendToStatus(LauncherLanguageManager.getString("launcher.statu
                updateButtonsState(currentEnvStatus != null && currentEnvStatus.
            });
        })
        .exceptionally(throwable -> {
            javafx.application.Platform.runLater(() -> {
                serverProgressIndicator.setVisible(false);
                serverProgressIndicator.setProgress(0); // Reset progress
                appendToStatus(LauncherLanguageManager.getString("launcher.statu
```

```
                serverRunning = false; // Reset se fallisce
                serverConnected = false; // Server failed to connect - ensure cl
                updateButtonsState(currentEnvStatus != null && currentEnvStatus.
            });
            return null;
        });
}
```

Spiegazione:

Il metodo onRunServer() gestisce l'avvio del processo server dal launcher GUI. Include validazioni di sicurezza, gestione dello stato dell'interfaccia, e coordinamento asincrono del processo server. Le funzionalità chiave sono:

- **Validazione prerequisiti**: Controllo che Java, Maven, PostgreSQL siano disponibili
- **Gestione stato**: Prevenzione avvii multipli, aggiornamento UI
- **Esecuzione asincrona**: Avvio del server in background con gestione errori
- **Feedback utente**: Indicatori di progresso e messaggi di stato
- **Gestione errori**: Recupero graceful da fallimenti di avvio

Complessità Algoritmica:

- **Tempo**: O(1) - operazioni di controllo e setup costanti
- **Spazio**: O(1) - nessuna allocazione significativa di memoria
- **Perché fondamentale**: Coordina l'avvio del server dal launcher, gestendo tutti gli aspetti dell'interfaccia utente e della sicurezza.

4. Metodo Fondamentale del Shared: reloadConfiguration() in AppConfig.java

Codice del Metodo:

```
/**
 * Reloads configuration from external config.properties file
 * This allows dynamic configuration updates without restarting
 */
public static void reloadConfiguration() {
    Properties p = new Properties();
    boolean loadedFromFile = false;

    // Try project directory first
    try {
```

```
            java.io.File configFile = new java.io.File("config.properties");
            if (configFile.exists() && configFile.length() > 0) {
                try (java.io.FileInputStream in = new java.io.FileInputStream(config
                    p.load(in);
                    loadedFromFile = true;
                    // Only log successful loads to avoid spam - very rarely
                } catch (Exception e) {
                    // Log error but don't spam console
                    if (System.currentTimeMillis() % 10000 < 100) { // Log only once
                        logger.fine("Failed to load config from project directory: "
                    }
                }
            }
        } catch (Exception e) {
            // Log error but don't spam console
            if (System.currentTimeMillis() % 10000 < 100) { // Log only once every ~
                logger.fine("Error accessing config file: " + e.getMessage());
            }
        }

        // Apply reloaded properties only if we successfully loaded something
        if (loadedFromFile) {
            if (p.getProperty("language") != null) {
                String newLanguage = p.getProperty("language");
                if (!newLanguage.equals(language)) {
                    language = newLanguage;
                    // Only log actual language changes, not every reload
                    logger.info("Language changed to: " + language);
                }
            }
            // Reload other dynamic properties as needed
            if (p.getProperty("serverHost") != null) serverHost = p.getProperty("ser
            if (p.getProperty("poolSize") != null) poolSize = Integer.parseInt(p.get
        }
    }
```

Spiegazione:

Il metodo `reloadConfiguration()` permette la ricarica dinamica della configurazione dell'applicazione senza riavvio. È cruciale per sistemi distribuiti che necessitano di aggiornamenti di configurazione a runtime. Le caratteristiche principali:

- **Caricamento gerarchico**: Prima directory del progetto, poi classpath come fallback
- **Gestione errori**: Logging selettivo per evitare spam del console

- **Applicazioni selettive**: Solo proprietà dinamiche vengono ricaricate
- **Thread safety**: Essendo statico, deve gestire concorrenza appropriatamente

Complessità Algoritmica:

- **Tempo**: O(1) - operazioni I/O limitate e costanti
- **Spazio**: O(1) - strutture dati temporanee di dimensione fissa
- **Perché fondamentale**: Abilita la configurazione dinamica del sistema condiviso, essenziale per deployment flessibili.

5. Query Fondamentale: getLibraryBooks() in BookDAOImpl.java

Codice della Query:

```
@Override
public List<Book> getLibraryBooks(int libraryId) throws Exception {
    return executeBookListQuery("SELECT b.* FROM Books b JOIN Books_Libraries bl
}
```

Spiegazione:

Questa query SQL recupera tutti i libri associati a una specifica libreria attraverso un JOIN tra le tabelle `Books` e `Books_Libraries`. È fondamentale per la funzionalità di gestione delle librerie personali degli utenti.

Complessità Algoritmica della Query:

- **Tempo**: O(m + n) dove m è il numero di libri nella libreria e n è il numero totale di libri (a causa del JOIN)
- **Spazio**: O(m) dove m è il numero di libri restituiti
- **Ottimizzazioni**: Utilizza indici sulle chiavi esterne per performance ottimali
- **Perché fondamentale**: È la query principale per visualizzare i contenuti delle librerie utente, operazione centrale del sistema di raccomandazione.

Conclusioni

Questi metodi rappresentano i pilastri architetturali del sistema BookRecommender:

1. **Client**: `start()` orchestra l'avvio dell'interfaccia utente
2. **Server**: `bootstrapHeadless()` rende operativo il backend
3. **Launcher**: `onRunServer()` coordina l'avvio dei componenti
4. **Shared**: `reloadConfiguration()` gestisce la configurazione dinamica
5. **Database**: `getLibraryBooks()` recupera i dati delle librerie

Tutti questi metodi hanno complessità algoritmica efficiente (principalmente O(1) o O(n) lineare) e sono progettati per essere robusti, con gestione errori completa e logging appropriato.

Guida al Deployment

Prerequisiti

- **Java 17** o superiore
- **Maven 3.8+**
- **PostgreSQL 12+**
- **RAM Minima**: 4GB raccomandati

Processo di Build

```
# Clona repository
git clone <repository-url>
cd BookRecommenderApp

# Build tutti i moduli
mvn clean package

# Build senza test
mvn clean package -DskipTests

# Crea distribuzione
mvn clean install -DskipTests

# Avvia launcher
cd bin/
bash launch-universal.sh

# Avvia componente singolo
cd src/serverBR/
mvn javafx:run
```

Deployment Server

Setup Database

```
-- Crea database
CREATE DATABASE projectb;

-- Esegui script inizializzazione
\i src/serverBR/src/main/resources/init.sql
```

Variabili Ambiente

```
# Configurazione database
export DB_HOST=localhost
export DB_NAME=projectb
export DB_USER=your_username
export DB_PASSWORD=your_password

# Configurazione server
export SERVER_PORT=1099
export SERVER_HOST=0.0.0.0
```

Avvia Server (Headless)

```
java -jar target/serverBR-1.0-SNAPSHOT-jar-with-dependencies.jar --headless
```

Avvia Server (GUI)

```
mvn javafx:run -pl src/serverBR
```

Deployment Client

Configurazione

```
# Connessione server
export APP_SERVER_HOST=localhost
export APP_SERVER_PORT=1099
```

Avvia Client

```
mvn javafx:run -pl src/clientBR
```

Oppure esegui il JAR pacchettizzato:

```
java -jar target/clientBR-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Deployment Docker (Opzionale)

Docker Compose

```yaml
version: '3.8'
services:
  postgres:
    image: postgres:14
    environment:
      POSTGRES_DB: projectb
      POSTGRES_USER: bookuser
      POSTGRES_PASSWORD: bookpass
    ports:
      - "5432:5432"

  server:
    build: serverBR
    depends_on:
      - postgres
    environment:
      DB_HOST: postgres
      DB_NAME: projectb
      DB_USER: bookuser
      DB_PASSWORD: bookpass
    ports:
      - "1099:1099"

  client:
    build: clientBR
    depends_on:
      - server
    environment:
      APP_SERVER_HOST: server
```

Build ed Esegui

```
docker-compose up --build
```

Monitoraggio & Manutenzione

Health Check

- Disponibilità server RMI
- Stato pool connessioni database
- Utilizzo memoria cache
- Conteggio sessioni attive

File di Log

- Server: `logs/server.log`
- Client: `logs/client.log`
- Errori: `logs/error.log`

Strategia Backup

- Dump database giornalieri
- Backup configurazione settimanali
- Rotazione log mensili

Troubleshooting

Problemi Comuni

Connessione Rifiutata

```
Errore: java.rmi.ConnectException: Connection refused
Soluzione: Assicurati che server RMI sia in esecuzione sulla porta 1099
```

Timeout Connessione Database

```
Errore: HikariPool-1 - Connection timeout
Soluzione: Controlla servizio PostgreSQL, verifica credenziali, aumenta timeout
```

Cache Miss

```
Avviso: Cache miss rate > 50%
Soluzione: Aumenta dimensione cache, regola TTL, controlla allocazione memoria
```

Out of Memory

```
Errore: java.lang.OutOfMemoryError: Java heap space
Soluzione: Aumenta dimensione heap JVM: -Xmx2048m
```

Miglioramenti Futuri

1. **Architettura Microservizi**: Dividi server monolitico in servizi
2. **Message Queue**: Aggiungi RabbitMQ/Kafka per operazioni asincrone
3. **Cache Redis**: Sostituisci cache in-memory con Redis
4. **API Gateway**: API REST per client mobili
5. **GraphQL**: Linguaggio query flessibile per recupero dati
6. **Machine Learning**: Algoritmi raccomandazione migliorati
7. **Aggiornamenti Real-time**: WebSocket per notifiche live
8. **App Mobile**: Implementazione React Native o Flutter

Contatti & Supporto

Per problemi tecnici, domande o contributi:

- **Repository Progetto**: [https://github.com/andreasib03/ProjectB]
- **Email**: support@bookrecommender.it
- **Form all’interno dell’applicazione**