RELAZIONE

Sistema di Gestione Biblioteca Digitale

Corso: ICT 2023/2025

Edoardo Spinelli, Rodari Filippo, Luca Cadei, Nicholas Martorelli

1. INTRODUZIONE

Questa relazione descrive il progetto di sviluppo del sistema backend per la gestione di una biblioteca digitale.

Il sistema sviluppato permette la gestione completa del catalogo, includendo informazioni relative ai libri, agli autori, agli utenti registrati e alle operazioni di prestito.

Particolare attenzione è stata posta all'implementazione di un sistema di autenticazione e autorizzazione robusto, che garantisce l'accesso controllato alle diverse funzionalità in base al ruolo dell'utente.

2. ARCHITETTURA DEL SISTEMA

2.1 Pattern Architetturale

L'architettura del sistema segue il pattern Model-View-Controller (MVC), che garantisce una chiara separazione delle responsabilità e facilita la manutenibilità del codice.

Nel contesto di un'applicazione API-first come questa, i modelli rappresentano le entità del dominio (User, Author, Book, Loan) e gestiscono la logica di accesso ai dati.

I controller contengono la logica di business e orchestrano le operazioni tra i diversi componenti del sistema.

Le view, in questo caso, sono rappresentate dall'interfaccia web presente nella cartella FE/library, che consuma le API esposte dal backend.

2.2 Middleware Stack

Il flusso di elaborazione delle richieste HTTP attraversa una serie di middleware che svolgono funzioni specifiche.

Ogni richiesta in ingresso viene prima processata dai middleware globali, poi da quelli di routing e infine raggiunge il controller appropriato.

Il middleware ContainerBindingsMiddleware si occupa di iniettare le dipendenze nel container IoC dell'applicazione.

Successivamente, il ForceJsonResponseMiddleware garantisce che tutte le risposte siano nel formato JSON, importante per un'API REST.

Il middleware CORS gestisce le policy di Cross-Origin Resource Sharing, permettendo al frontend di comunicare con il backend anche se serviti da domini diversi.

Dopo questi middleware globali, intervengono quelli specifici del router: il BodyParserMiddleware si occupa di parsare il corpo delle richieste (JSON, form-data, etc.), mentre InitializeAuthMiddleware inizializza il sistema di autenticazione controllando la presenza e validità dei token.

Infine, i middleware custom (auth, admin, userOrAdmin) implementano la logica di autorizzazione specifica dell'applicazione prima che la richiesta raggiunga il controller.

2.3 Sistema di Autenticazione

Il sistema di autenticazione implementato è di tipo token-based, seguendo lo standard Bearer Token.

Questa scelta è stata dettata dalla natura stateless delle API REST e dalla necessità di garantire scalabilità orizzontale del sistema.

Quando un utente effettua il login con credenziali valide, il sistema genera un token univoco che viene memorizzato nel database nella tabella auth_access_tokens.

Il client deve includere questo token nell'header Authorization di ogni richiesta successiva.

L'hashing delle password avviene tramite l'algoritmo Scrypt. Scrypt è stato scelto in quanto particolarmente resistente agli attacchi brute-force grazie alla sua natura memory-hard.

Il sistema implementa due ruoli distinti: admin e user.

Gli utenti con ruolo admin hanno accesso completo a tutte le funzionalità del sistema, inclusa la gestione del catalogo.

Gli utenti con ruolo user possono effettuare operazioni di lettura sul catalogo e gestire i propri prestiti, ma non possono modificare libri o autori.

3. MODELLO DEI DATI

3.1 Schema del Database

Il database è strutturato in quattro tabelle principali che rappresentano le entità fondamentali del dominio.

La tabella users memorizza le informazioni degli utenti registrati, includendo email, password (hashata), nome completo e ruolo.

Ogni utente è identificato univocamente da un ID numerico auto-incrementale.

La tabella authors contiene i dati relativi agli autori dei libri.

Oltre all'identificativo univoco e al nome dell'autore, è presente un campo bio opzionale che può contenere informazioni biografiche.

La tabella books rappresenta il catalogo della biblioteca e include titolo, codice ISBN (con constraint di unicità), riferimento all'autore, anno di pubblicazione e un flag booleano che indica la disponibilità del libro per il prestito.

Infine, la tabella loans gestisce le operazioni di prestito.

Ogni record rappresenta un prestito e contiene il riferimento all'utente che ha richiesto il prestito, il libro prestato, la data di prestito, la data di restituzione effettiva e un flag booleano che indica se il libro è stato restituito.

3.2 Relazioni tra Entità

Le relazioni tra le entità sono state progettate seguendo le best practice della normalizzazione database.

Ogni libro appartiene a un singolo autore attraverso una relazione many-to-one implementata tramite la foreign key author_id nella tabella books.

Questa scelta modellistica riflette la realtà della maggior parte delle pubblicazioni, dove un libro ha un autore principale identificato.

Al contrario, un autore può aver scritto molti libri, realizzando così una relazione one-to-many dal punto di vista dell'autore.

Un utente può effettuare molteplici prestiti nel tempo, stabilendo una relazione one-to-many tra users e loans.

Analogamente, un libro può essere oggetto di molteplici prestiti in momenti diversi, creando una relazione one-to-many anche tra books e loans.

Queste relazioni sono implementate tramite foreign key con constraint di integrità referenziale e policy di cascade delete dove appropriato.

Ad esempio, l'eliminazione di un utente comporta automaticamente l'eliminazione di tutti i suoi prestiti, mantenendo la consistenza dei dati.

3.3 Vincoli di Integrità

Il database implementa diversi constraint per garantire l'integrità dei dati.

Il codice ISBN è definito come univoco nella tabella books, prevenendo l'inserimento accidentale di duplicati.

Le foreign key sono definite con constraint ON DELETE CASCADE dove appropriato: eliminando un autore vengono eliminati tutti i suoi libri, eliminando un utente vengono eliminati tutti i suoi prestiti, ed eliminando un libro vengono eliminati tutti i record di prestito associati.

Sono inoltre presenti constraint NOT NULL sui campi essenziali come email, password, nome dell'autore, titolo del libro e date di prestito.

Il campo role nella tabella users è implementato come ENUM con valori 'user' e 'admin', garantendo che non possano essere inseriti valori non validi.

Questi constraint a livello database rappresentano una prima linea di difesa contro dati inconsistenti, complementare alla validazione applicativa.

4. SCELTE PROGETTUALI

4.1 Autenticazione Token-Based

La scelta di implementare un sistema di autenticazione token-based è stata motivata da diverse considerazioni architetturali.

Primo, le API REST sono per natura stateless, e l'uso di sessioni server-side avrebbe violato questo principio fondamentale.

I token Bearer permettono di mantenere lo stato di autenticazione sul client, trasmettendo ad ogni richiesta le credenziali necessarie.

Secondo, questa architettura facilita la scalabilità orizzontale.

Non essendo necessario mantenere sessioni in memoria sul server, è possibile distribuire il carico su più istanze dell'applicazione senza dover implementare meccanismi di condivisione dello stato di sessione.

Terzo, i token sono facilmente utilizzabili da diversi tipi di client: applicazioni web, mobile native, desktop, o altri servizi backend che devono interagire con le nostre API.

L'implementazione utilizza il modulo @adonisjs/auth con il provider DbAccessTokensProvider, che memorizza i token nel database.

Ogni token ha una durata illimitata per semplicità, ma in un ambiente di produzione sarebbe opportuno implementare una scadenza temporale e un meccanismo di refresh token.

Il token viene generato automaticamente al momento del login utilizzando un generatore crittograficamente sicuro, garantendo l'impossibilità pratica di predire o falsificare token validi.

4.2 Sistema di Autorizzazione Basato su Middleware

L'autorizzazione è implementata attraverso un sistema di middleware modulari che possono essere composti per creare policy di accesso complesse.

Il middleware auth rappresenta il livello base di protezione: verifica semplicemente che l'utente sia autenticato controllando la presenza e validità del token.

Se il token non è presente o non è valido, la richiesta viene rifiutata con un errore 401 Unauthorized.

Il middleware admin implementa un controllo più restrittivo, verificando che l'utente autenticato abbia il ruolo 'admin'.

Questo middleware è applicato a tutte le operazioni di scrittura (CREATE, UPDATE, DELETE) sulle entità books e authors, garantendo che solo gli amministratori possano modificare il catalogo.

Il middleware userOrAdmin implementa una logica più sofisticata necessaria per la gestione degli utenti.

Quando un utente tenta di modificare un profilo (PUT /users/:id), il middleware verifica che l'ID nell'URL corrisponda all'ID dell'utente autenticato oppure che l'utente sia un admin.

In questo modo, ogni utente può modificare solo il proprio profilo, mentre gli admin mantengono il controllo completo.

Questa soluzione elegante evita la duplicazione di logica nei controller e rende le policy di autorizzazione esplicite e facilmente manutenibili.

4.3 Validazione degli Input con VineJS

La validazione degli input utente è un aspetto critico per la sicurezza e l'integrità dei dati.

VineJS è stato scelto come libreria di validazione per diverse ragioni.

Primo, è la soluzione raccomandata e mantenuta dal team di AdonisJS, garantendo integrazione ottimale con il framework.

Secondo, offre un'API fluente e dichiarativa che rende i validator facilmente leggibili e manutenibili.

Terzo, fornisce excellent integration con TypeScript, permettendo di inferire automaticamente i tipi dei dati validati.

Per ogni operazione di scrittura è stato creato un validator dedicato.

Ad esempio, il createBookValidator definisce le regole per la creazione di un nuovo libro: il titolo deve essere una stringa non vuota di massimo 255 caratteri, l'ISBN deve rispettare il pattern numerico con eventuali trattini, l'author_id deve essere un numero positivo.

Il validator si occupa anche di sanitizzare i dati, rimuovendo spazi bianchi superflui (trim) e convertendo i tipi dove necessario.

Un aspetto importante è la presenza di validator separati per le operazioni di creazione e aggiornamento.

Il createBookValidator richiede tutti i campi obbligatori, mentre l'updateBookValidator rende tutti i campi opzionali, permettendo aggiornamenti parziali delle risorse.

Questo approccio segue le best practice REST e migliora l'usabilità dell'API.

4.4 Factory Pattern per la Generazione di Dati di Test

Il pattern Factory è stato implementato per facilitare la generazione di dati di test realistici e coerenti.

Le factory sono definite utilizzando @faker-js/faker, una libreria che genera dati casuali ma verosimili in molte categorie (nomi, date, testi, numeri, etc.).

AuthorFactory genera istanze di autori con nomi completi realistici e biografie costituite da più paragrafi di testo lorem ipsum.

BookFactory genera libri con titoli di tre parole, codici ISBN numerici di 13 cifre, anni di pubblicazione casuali negli ultimi 50 anni e stati di disponibilità casuali.

Importante notare che BookFactory definisce anche la relazione con AuthorFactory, permettendo di generare automaticamente l'autore associato ad ogni libro.

Questa implementazione permette di popolare rapidamente il database con dati di test per lo sviluppo e il debugging.

Ad esempio, con una singola chiamata è possibile creare cinque autori, ciascuno con tre libri associati.

Le factory sono particolarmente utili anche per i test automatizzati, garantendo che ogni test parta da uno stato iniziale consistente e prevedibile.

4.5 Seeder Idempotente

La strategia di seeding del database è stata progettata per essere idempotente, ovvero può essere eseguita molteplici volte senza creare duplicati o causare errori.

Questo è particolarmente importante in scenari di sviluppo dove si potrebbe voler resettare e ripopolare il database frequentemente.

L'idempotenza è ottenuta utilizzando il metodo updateOrCreate invece del semplice create.

Questo metodo cerca prima un record esistente in base a una chiave univoca (ad esempio, l'ISBN per i libri o il nome per gli autori).

Se il record esiste, viene aggiornato con i nuovi dati; se non esiste, viene creato.

In questo modo, eseguire il seeder più volte aggiorna semplicemente i dati esistenti invece di tentare di creare duplicati e fallire a causa dei constraint di unicità.

Un aspetto tecnico importante è la gestione degli ID nelle relazioni.

Poiché gli ID generati dal database potrebbero cambiare tra diverse esecuzioni del seeder, viene mantenuta una mappa in memoria che associa gli ID logici (usati nel file di seed) agli ID effettivi del database.

Questo permette di mantenere corrette le relazioni tra autori e libri anche quando il seeder viene eseguito su un database già popolato.

4.6 Comando CLI per la Gestione degli Admin

Il comando create:admin è stato implementato per semplificare la creazione di utenti amministratori.

In un'applicazione reale, il primo utente admin deve essere creato prima che l'applicazione sia funzionalmente utilizzabile, creando un problema di bootstrap.

Permettere la registrazione diretta come admin tramite l'API sarebbe un grave problema di sicurezza.

Il comando risolve questo dilemma fornendo un'interfaccia a riga di comando sicura per la creazione di admin.

Essendo eseguito direttamente sul server, richiede accesso fisico o SSH al sistema, rappresentando un livello di sicurezza appropriato per un'operazione privilegiata.

Il comando utilizza l'interfaccia prompts di AdonisJS per richiedere interattivamente email, password e nome completo, con la password mascherata per sicurezza.

Prima di creare l'admin, il comando verifica che non esista già un utente con la stessa email, prevenendo errori di duplicazione.

La password viene hashata utilizzando lo stesso meccanismo dell'endpoint di registrazione, garantendo consistenza.

Una volta creato l'admin, viene visualizzato un messaggio di conferma con l'email dell'utente creato.

4.7 Gestione del Ciclo di Vita dei Prestiti

La logica di gestione dei prestiti implementa un workflow che riflette le operazioni reali di una biblioteca.

Quando viene creato un nuovo prestito, il sistema verifica prima che il libro richiesto sia disponibile interrogando il flag available nella tabella books.

Se il libro è già in prestito ad un altro utente, la richiesta viene rifiutata con un errore esplicativo.

Se il libro è disponibile, viene creato il record di prestito nella tabella loans con i dati forniti dall'utente (book_id e loan_date) e l'ID dell'utente autenticato recuperato dal token.

Contemporaneamente, il flag available del libro viene impostato a false, impedendo che venga prestato a più utenti contemporaneamente.

Tutte queste operazioni avvengono implicitamente in una transazione database, garantendo consistenza anche in caso di errori.

La restituzione del libro avviene tramite un'operazione di update sul prestito, impostando il flag returned a true e la data di restituzione.

A questo punto il flag available del libro viene nuovamente impostato a true, rendendo il libro disponibile per nuovi prestiti.

È importante notare che questa operazione può essere eseguita solo dall'utente che ha effettuato il prestito o da un amministratore, garantendo che gli utenti non possano modificare i prestiti altrui.

5. IMPLEMENTAZIONE DELLE API REST

5.1 Principi di Design

Le API implementate seguono i principi architetturali REST (Representational State Transfer) e le convenzioni comuni nell'industria.

Ogni risorsa è identificata da un URL univoco e le operazioni disponibili sono mappate sui metodi HTTP standard: GET per la lettura, POST per la creazione, PUT per l'aggiornamento completo, PATCH per l'aggiornamento parziale, DELETE per l'eliminazione.

Gli URL seguono una struttura gerarchica e semantica.

Le collezioni sono rappresentate al plurale (ad esempio /books per la lista di tutti i libri), mentre le risorse individuali sono identificate aggiungendo l'ID (ad esempio /books/5 per il libro con ID 5).

Le sotto-risorse seguono la stessa logica: /authors/3/books potrebbe rappresentare tutti i libri di un autore specifico, anche se questa funzionalità non è stata implementata nel progetto corrente.

Le risposte utilizzano i codici di stato HTTP appropriati per comunicare l'esito dell'operazione.

200 OK indica successo per operazioni di lettura e aggiornamento, 201 Created per creazioni riuscite, 204 No Content per eliminazioni riuscite.

Gli errori utilizzano i codici 4xx per problemi client-side (400 Bad Request per validazione fallita, 401 Unauthorized per autenticazione mancante, 403 Forbidden per autorizzazione insufficiente, 404 Not Found per risorse inesistenti) e 5xx per problemi server-side.

5.2 Endpoints Pubblici

Gli endpoint pubblici sono accessibili senza autenticazione e permettono le operazioni fondamentali per l'utilizzo dell'applicazione.

L'endpoint POST /register permette la registrazione di nuovi utenti fornendo email, password e nome completo.

La validazione garantisce che l'email sia in formato valido e che la password abbia almeno 8 caratteri.

Se la registrazione ha successo, viene restituito l'oggetto utente creato insieme ad un token di autenticazione, permettendo l'accesso immediato senza dover effettuare un login separato.

L'endpoint POST /login autentica un utente esistente verificando email e password.

Se le credenziali sono corrette, viene generato e restituito un nuovo token di autenticazione che il client dovrà includere nelle richieste successive.

La risposta include anche l'oggetto utente con le sue proprietà (incluso il ruolo), permettendo al client di configurare l'interfaccia appropriatamente.

Gli endpoint GET /books e GET /authors permettono la consultazione pubblica del catalogo.

Supportano parametri di query string per la paginazione (page, per_page) e la ricerca (search).

La ricerca è case-insensitive e cerca nel titolo e nell'ISBN per i libri, nel nome e nella biografia per gli autori.

Le risposte sono paginate per default a 25 elementi per pagina, con metadata che indicano la pagina corrente e il numero totale di pagine.

5.3 Endpoints Protetti

Gli endpoint protetti richiedono un token di autenticazione valido nell'header Authorization nel formato "Bearer {token}".

Se il token è mancante o invalido, tutte le richieste vengono rifiutate con un errore 401.

Gli endpoint di gestione dei prestiti sono accessibili a tutti gli utenti autenticati.

GET /loans restituisce la lista dei prestiti dell'utente corrente (o tutti i prestiti se l'utente è admin).

POST /loans crea un nuovo prestito verificando la disponibilità del libro.

PUT /loans/:id permette di marcare un prestito come restituito.

DELETE /loans/:id, riservato agli admin, permette di eliminare definitivamente un prestito dal sistema.

Gli endpoint di gestione del catalogo (POST/PUT/DELETE su /books e /authors) sono protetti dal middleware admin, garantendo che solo gli amministratori possano modificare il catalogo.

Questi endpoint accettano i dati validati e eseguono le operazioni corrispondenti, restituendo l'oggetto aggiornato o un codice 204 per le eliminazioni.

Gli endpoint di gestione utenti permettono agli utenti di visualizzare e modificare i propri dati.

GET /users/:id e PUT /users/:id utilizzano il middleware userOrAdmin, permettendo a ciascun utente di accedere solo al proprio profilo, mentre gli admin possono accedere a tutti i profili.

Questo design bilancia privacy e necessità amministrative.

5.4 Formato delle Risposte

Tutte le risposte sono in formato JSON con l'header Content-Type impostato a "application/json".

Le risposte di successo per operazioni di lettura e creazione includono i dati richiesti.

Per le collezioni, i dati sono wrappati in un oggetto che include anche metadata di paginazione:

json

```
{
  "meta": {
    "currentPage": 1,
    "lastPage": 5,
    "perPage": 25,
    "total": 120
  },
  "data": [...]
}
```

Le risposte di errore seguono una struttura consistente con un campo error che descrive il problema in linguaggio naturale.

Gli errori di validazione includono dettagli specifici su quali campi hanno fallito la validazione e perché. Questa consistenza nel formato delle risposte facilita la gestione degli errori sul client.

6. SICUREZZA

6.1 Sicurezza delle Password

La sicurezza delle password è implementata seguendo le raccomandazioni OWASP. Le password non vengono mai memorizzate in chiaro ma solo in forma di hash crittografico.

L'algoritmo utilizzato è Scrypt, scelto per la sua resistenza agli attacchi brute-force tramite hardware specializzato.

Scrypt è infatti un algoritmo memory-hard, che richiede una quantità significativa di memoria oltre al tempo di calcolo, rendendo economicamente non conveniente l'uso di hardware dedicato per attacchi su larga scala.

I parametri di configurazione di Scrypt sono stati impostati per bilanciare sicurezza e performance.

Il cost factor di 16384 garantisce che ogni operazione di hashing richieda circa 100-200ms su hardware moderno, abbastanza per essere impercettibile per un utente legittimo ma sufficiente a rendere impraticabile un attacco brute-force.

Il salting è gestito automaticamente dall'implementazione, con un salt univoco generato per ogni password che impedisce l'uso di rainbow tables precompilate.

La validazione delle password richiede un minimo di 8 caratteri.

In un'applicazione di produzione, sarebbe opportuno implementare requisiti più stringenti (lunghezza minima maggiore, presenza di caratteri speciali, numeri, maiuscole/minuscole), ma per semplicità in questo progetto didattico i requisiti sono stati mantenuti minimi.

6.2 Protezione da SQL Injection

La protezione da SQL Injection è garantita dall'utilizzo consistente dell'ORM Lucid per tutte le operazioni sul database.

Lucid utilizza internamente query parametrizzate, dove i valori forniti dall'utente vengono passati separatamente dal testo della query e correttamente escaped dal driver del database.

Questo rende impossibile l'iniezione di codice SQL arbitrario attraverso gli input utente.

Anche nelle rare situazioni dove sono state utilizzate query raw per operazioni complesse, i parametri sono stati sempre passati attraverso i binding dell'ORM invece di concatenazione di stringhe.

Questo approccio viene applicato automaticamente da Lucid, ma è importante che gli sviluppatori ne siano consapevoli e evitino di bypassare queste protezioni con query costruite manualmente.

6.3 Gestione delle Cross-Origin Requests

Il Cross-Origin Resource Sharing (CORS) è configurato per permettere al frontend di comunicare con il backend anche quando serviti da origini diverse.

In modalità sviluppo, CORS è configurato per accettare richieste da qualsiasi origine, semplificando il workflow di sviluppo.

In un ambiente di produzione, questa configurazione dovrebbe essere ristretta per accettare richieste solo da domini specifici e fidati.

La configurazione CORS include anche il supporto per credenziali (credentials: true), permettendo l'invio di token di autenticazione.

I metodi HTTP permessi includono tutti quelli utilizzati dall'applicazione (GET, POST, PUT, DELETE), mentre le opzioni preflight sono gestite automaticamente dal middleware.

6.4 Protezione da XSS

La protezione da Cross-Site Scripting è facilitata dall'architettura dell'applicazione.

Essendo un'API pura che restituisce solo JSON senza rendering di HTML sul server, molti vettori di attacco XSS sono automaticamente mitigati.

Il frontend implementa le proprie protezioni gestendo correttamente l'escape dei dati prima di inserirli nel DOM.

Inoltre, la validazione degli input rimuove o escape automaticamente caratteri potenzialmente pericolosi.

I campi di tipo string vengono sanitizzati con trim() e altri metodi di pulizia forniti da VineJS.

Questo approccio defense-in-depth garantisce che anche se un layer di protezione fallisce, altri layer sono pronti a prevenire l'exploit.

6.5 Rate Limiting e Protezione da Abusi

Sebbene non implementato in questa versione del progetto, un sistema di rate limiting dovrebbe essere aggiunto in produzione per prevenire abusi delle API.

Il rate limiting limiterebbe il numero di richieste che un singolo client può effettuare in un determinato intervallo di tempo, prevenendo attacchi denial-of-service e tentativi di brute-force sugli endpoint di autenticazione.

AdonisJS fornisce supporto per rate limiting attraverso moduli dedicati che possono essere facilmente integrati.

Una configurazione tipica potrebbe limitare gli endpoint di autenticazione a 5 tentativi per IP ogni 15 minuti, mentre gli altri endpoint potrebbero permettere 100 richieste per utente autenticato al minuto.