ATE - Diario.Solution versione 1.0

Indice

Presentazione

1.1 Descrizione generale

1.2 Tecnologie utilizzate

Backend

2.1 Modello teorico

2.2 Architettura nel Dettaglio

2.3 Database   
2.4 Classi modello

2.5 Controller

Implementazioni Future

3.1 Implementazioni

### 

PRESENTAZIONE

### **1.1 Descrizione generale**

"Diario" è un'API REST progettata per consentire agli utenti di gestire un diario personale. Ogni utente può registrarsi tramite un'apposita pagina, autenticarsi e accedere alla propria sezione, dove avrà la possibilità di creare, salvare e gestire liberamente i propri testi.

### **1.2 Tecnologie Utilizzate**

Per lo sviluppo del progetto "Diario" sono state utilizzate le seguenti tecnologie:

* **C#:** Linguaggio di programmazione utilizzato per lo sviluppo del backend.
* **ASP.NET Core**: Framework utilizzato per semplificare l'implementazione del backend e gestire le dipendenze del progetto.
* **Entity Framework Core (EF Core)**: ORM (Object-Relational Mapper) utilizzato per interagire con il database PostgreSQL, semplificando la gestione delle operazioni CRUD.
* **ServerSQL**: Database relazionale utilizzato per memorizzare la lista degli utenti registrati e gestire altre informazioni strutturate.
* **MongoDB**: Database NoSQL utilizzato per memorizzare i testi salvati dagli utenti. Questo approccio consente una gestione flessibile dei dati.

BACKEND

### **2.1 Modello Teorico**

Il progetto segue il modello teorico implementativo **Model-View-Controller (MVC)**, che separa la logica di business, la presentazione e la gestione delle richieste. In questo contesto:

* **Model**: Rappresentato dalle classi **Utente** e **Diario**, che sono responsabili della gestione dei dati. Queste classi definiscono la struttura dei dati che vengono utilizzati per interagire con il database.
* **View**: Vista che l'utente vede nel browser. Per questa versione del progetto non è presente e verrà implementata successivamente. La presentazione della view sarà realizzata utilizzando **Microsoft Blazor**, un framework web di Microsoft che consente di sviluppare interfacce utente dinamiche in C# direttamente nel browser.
* **Controller**: I **controller** gestiscono la comunicazione tra il **Model** e la **View**. In questo caso, sono presenti: **UtenteAdminController, UtenteUtenteController**, **DiarioUtenteController, DiarioAdminController**:

### **2.2 Architettura nel Dettaglio**

L'architettura del progetto è suddivisa in vari componenti, ognuno dei quali svolge un ruolo specifico nell'elaborazione e nella gestione dei dati.

### **Entity Layer**

Il **Entity Layer** è il componente che si occupa della rappresentazione degli oggetti del dominio attraverso le classi modello. Queste classi sono utilizzate per rappresentare i dati in entrata e in uscita dal sistema e sono mappate direttamente alle tabelle o collections dei rispettivi database (PostgreSQL e MongoDB). Nel progetto Diario abbiamo due entity

* **Utente**: Contiene le proprietà dell’utente
* **Diario:** Contiene le proprietà del diario

#### **DataLayer**

Il **DataLayer** è la componente che si occupa della gestione della connessione ai database e delle operazioni di accesso ai dati. All'interno di questa layer troviamo:

* **MongoDBContext**: Gestisce la connessione e le operazioni con il database MongoDB, dove vengono memorizzati i testi salvati dagli utenti.
* **SQLServerContext**: Gestisce la connessione e le operazioni con il database PostgreSQL, dove vengono memorizzati gli utenti e i loro dati di login.

#### **Service Layer**

Il **Service Layer** è il livello che contiene la logica di business e gestisce l'interazione con i dati tramite i metodi CRUD generici definiti nell'interfaccia **ICRUDService**. I principali servizi sono:

* **UtenteService**: Gestisce la logica per la gestione degli utenti, inclusi la registrazione, l'autenticazione (se implementata), e le operazioni CRUD per gli utenti.
* **DiarioService**: Gestisce la logica per la creazione, modifica, lettura e cancellazione dei diari degli utenti.

Entrambi i servizi si interfacciano con i rispettivi repository per eseguire operazioni di lettura e scrittura sui database.

#### **Business Layer**

Il **Business Layer** è il livello che implementa la logica complessa e specifica per la gestione dei dati. Questo livello contiene:

* **UtenteManager**: Si occupa della logica di gestione degli utenti, eseguendo operazioni di registrazione, aggiornamento e cancellazione degli utenti nel database.
* **DiarioManager**: Gestisce la logica per creare, leggere, aggiornare e eliminare i diari degli utenti.

I manager implementano i metodi CRUD definiti nel Service Layer e si interfacciano direttamente con i repository per l'accesso ai dati.

#### **ControllerLayer**

Il **ControllerLayer** è il livello che gestisce le chiamate HTTP in arrivo, indirizzandole ai metodi appropriati dei servizi e dei manager. I principali controller sono:

* **UtenteAdminController**: Gestisce le operazioni per l'amministratore, come la visualizzazione, l'aggiornamento e l'eliminazione degli utenti.
* **UtenteUtenteController**: Gestisce le operazioni che gli utenti possono eseguire sui propri dati, come la registrazione e l'aggiornamento del profilo.
* **DiarioUtenteController**: Gestisce le operazioni per gli utenti relative ai diari, inclusi la creazione, la lettura, l'aggiornamento e la cancellazione dei diari.
* **DiarioAdminController**: Gestisce le operazioni per l'amministratore relative ai diari, come la visualizzazione e la gestione dei diari di tutti gli utenti.

**2.3 Database**

Nel progetto "Diario", i database sono collegati al backend attraverso il DataLayer, come descritto nel paragrafo 2.1 Architettura Software. I DataLayer interagiscono con le classi modello per effettuare operazioni CRUD sui rispettivi database, **SQLServer** e **MongoDB**.

#### **SQLServer:**

Nel database **DiarioDBSQLSERVER** di SQLServer, la classe modello **Utente** è annotata con le convenzioni appropriate per fare riferimento alla tabella "utenti". La tabella è composta dai seguenti campi:

* **id**: è la primary key auto-incrementale.
* **username**: è un campo unico che identifica l'utente.
* **password**: campo di tipo **varchar** che memorizza la password dell'utente.
* **ruolo**: campo di tipo **varchar** che indica il ruolo dell'utente (es. admin, user).

#### **MongoDB:**

Per la gestione dei testi, il progetto utilizza il database **MongoDB**, con la **collection "testi"** contenuta nel documento **DiarioDb**. Ogni documento rappresenta un testo salvato dall'utente ed è composto dai seguenti campi:

* **\_id**: Campo di tipo **ObjectId** di MongoDB, che rappresenta un identificatore univoco per ogni testo creato dall'utente.
* **userId**: Campo di tipo **int** che fa riferimento all'ID dell'utente associato, il quale corrisponde all'ID dell'utente nel database **PostgreSQL**.
* **testo**: Campo di tipo **String** che contiene il contenuto del testo scritto dall'utente.
* **dataTesto**: Campo di tipo **LocalDate** che indica la data in cui il testo è stato scritto.

### **2.4 Classi Modello**

Le classi modello implementate in C# per il progetto sono:

#### **Diario**

La classe **Diario** è annotata con **[BsonCollection("testi")]** per collegarsi alla collection **"testi"** nel database **MongoDB**. Gli attributi della classe sono:

* **\_id**: ID univoco del documento (utilizza l'annotazione **[BsonId]** per collegarsi all'ObjectId di MongoDB).
* **userId**: ID dell'utente a cui appartiene il testo (è un campo **int** che fa riferimento all'ID dell'utente nel database PostgreSQL).
* **testo**: Il contenuto del diario (campo **string**).
* **dataTesto**: La data in cui il testo è stato scritto (campo **LocalDate**).

#### **Utente**

La classe **Utente** è annotata con **[Table("utenti")]** per collegarsi alla tabella **"utenti"** nel database **SQLServer**. Gli attributi della classe sono:

* **id**: ID univoco dell'utente (Primary Key auto-incrementale), che è annotato come **[Key]** per il riconoscimento da parte di Entity Framework.
* **utente**: Nome utente, che deve essere unico (**[Unique]**), ed è annotato con **[Column("username")]** per la gestione della colonna corrispondente.
* **password**: La password dell'utente (**[Column("password")]**).
* **role**: Il ruolo dell'utente (es. ROLE\_ADMIN o ROLE\_USER) annotato con **[Column("role")]**.

**2.5 Controller**

### **Controller: DiarioAdminController**

1. **GET** /DiarioApi/Diario/Admin/HomeDiari  
    Restituisce tutti i diari nel database (solo per Admin).
2. **GET** /DiarioApi/Diario/Admin/HomeDiari/{id}  
    Restituisce un diario per ID (solo per Admin).
3. **GET** /DiarioApi/Diario/Admin/HomeDiari/ListaUtente/{userId}  
    Restituisce tutti i diari di un utente specifico (solo per Admin).
4. **POST** /DiarioApi/Diario/Admin/HomeDiari  
    Crea un nuovo diario (solo per Admin).
5. **PUT** /DiarioApi/Diario/Admin/HomeDiari/{id}  
    Modifica un diario (solo per Admin).
6. **DELETE** /DiarioApi/Diario/Admin/HomeDiari/{id}  
    Elimina un diario (solo per Admin).

### **Controller: DiarioUtenteController**

1. **GET** /DiarioApi/Diario/Utente/HomeDiari/Testo/{id}  
    Restituisce un diario per ID (solo per Utente).
2. **GET** /DiarioApi/Diario/Utente/HomeDiari/{userId}  
    Restituisce tutti i diari di un utente specifico (solo per Utente).
3. **POST** /DiarioApi/Diario/Utente/HomeDiari  
    Crea un nuovo diario (solo per Utente).
4. **PUT** /DiarioApi/Diario/Utente/HomeDiari/{id}  
    Modifica un diario (solo per Utente).
5. **DELETE** /DiarioApi/Diario/Utente/HomeDiari/{id}  
    Elimina un diario (solo per Utente).

### **Controller: UtenteAdminController**

1. **GET** /api/UtenteAdmin/HomeUtenti  
    Restituisce tutti gli utenti (solo per Admin).
2. **GET** /api/UtenteAdmin/HomeUtenti/{id}  
    Restituisce un utente per ID (solo per Admin).
3. **POST** /api/UtenteAdmin/HomeUtenti  
    Crea un nuovo utente (solo per Admin).
4. **PUT** /api/UtenteAdmin/HomeUtenti/{id}  
    Modifica un utente (solo per Admin).
5. **DELETE** /api/UtenteAdmin/HomeUtenti/{id}  
    Elimina un utente (solo per Admin).

### **Controller: UtenteUtenteController**

1. **POST** /Registrazione  
    Registra un nuovo utente (accessibile senza autenticazione).
2. **GET** /api/UtenteUtente/{id}  
    Restituisce un utente per ID (solo per Utente).
3. **DELETE** /Utente/Eliminazione/{id}  
    Elimina un utente (solo per Utente).

IMPLEMENTAZIONI FUTURE

3.1 Implementazioni

1. **Front-End con Blazor**:

Il front-end dell'applicazione sarà sviluppato utilizzando **Blazor**, un framework che permette di creare interfacce utente interattive per il web utilizzando C# anziché JavaScript. Questo fornirà una soluzione più integrata con il back-end in .NET e migliorerà l'interazione tra la logica di business e la presentazione dell'applicazione.

1. **Aggiunta di Regole di Validazione**:  
    Saranno implementate regole di validazione sui moduli e sui dati inviati dai client. Questo garantirà che tutte le informazioni inserite siano corrette e sicure, migliorando l'affidabilità e la qualità dei dati. Le validazioni saranno applicate sia sul lato client (Blazor) che sul lato server per assicurarsi che non vengano mai processati dati errati.
2. **Gestione della Sicurezza delle Autenticazioni**:  
    Sarà implementato un sistema di **autenticazione sicura** per gestire l'accesso dell'utente all'applicazione. Verranno utilizzati protocolli sicuri come **JWT (JSON Web Tokens)** per la gestione delle sessioni utente, con crittografia adeguata per proteggere le credenziali e i dati sensibili. Verrà inoltre introdotto un sistema di autorizzazione, che limiterà l'accesso alle funzionalità in base ai ruoli (Admin, Utente), migliorando la sicurezza complessiva dell'applicazione.

Queste implementazioni contribuiranno a rendere l'applicazione più robusta, sicura e user-friendly, migliorando l'esperienza complessiva degli utenti e la manutenzione del progetto a lungo termine.