Progetto “Diario” - ATE

Indice

Presentazione

1.1 Descrizione generale

1.2 Tecnologie Utilizzate

Backend

2.1 Architettura software

2.2 Database   
2.3 Classi modello

2.4 Controller static Admin e Utente

2.5 Controller Api Admin e Utente

2.6 Service

2.7 Repository

2.8 Sicurezza

Frontend

3.1 Html e css

3.2 Javascript

**Presentazione**

1.1 Descrizione generale

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

1.2 Tecnologie UtilizzatePer lo sviluppo del progetto "Diario" sono state utilizzate le seguenti tecnologie:

* *Java 21*: Linguaggio di programmazione utilizzato per lo sviluppo del backend.
* *Maven 4.0.0*: Build-tool utilizzato per la gestione e automazione del ciclo di vita del progetto.
* *Spring Boot 3.4.1:* Framework utilizzato per semplificare l'implementazione del backend e gestire le dipendenze del progetto. Le principali estensioni di Spring utilizzate sono:
* *Spring Security*: Per proteggere gli endpoint e implementare l'autenticazione degli utenti.
* *Spring Data JPA*: Per gestire l'interazione con il database PostgreSQL.
* *Spring Data MongoDB:* Per collegare il database MongoDB al backend.
* *Spring Validation*: Per applicare regole di validazione ai dati in ingresso.
* *Spring Web:* Fornisce il supporto per creare endpoint RESTful e gestire le richieste HTTP.
* *Spring Thymeleaf:* Utilizzato per gestire le pagine statiche e restituire file HTML in risposta alle richieste.
* *PostgreSQL*: Database utilizzato per memorizzare la lista degli utenti registrati.
* *MongoDB*: Database utilizzato per memorizzare i testi salvati dagli utenti.
* *Html*: Creazione della struttura e del contenuto delle pagine web, definendo gli elementi visibili all'utente.
* *Css*: Gestione della presentazione e dello stile delle pagine HTML per migliorare l'estetica e l'esperienza utente (User Experience).
* *Javascript:* Gestione della dinamica e dell'interattività delle pagine web, oltre a consentire la comunicazione con il back-end tramite richieste API.
* *Eclipse :* IDE utilizzato per il progetto

**Backend**

1.2 Architettura Software

*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 modello, in questo caso le classi *Utente* e *Diario*. Queste classi gestiscono i dati e la logica di business.
* *View:* Le pagine HTML che l'utente vede nel browser, che includono:

*Pagine statiche accessibili a tutti:*

* Home
* Login
* SignIn

*Pagine protette da sicurezza (accessibili in base ai permessi):*

* User
* Admin
* AreaTestiAdmin
* AreaUtenteAdmin
* *Controller*: Gestisce la comunicazione tra il Model e la View. In questo caso, abbiamo:
* *Controller statico*: Gestito con Spring Boot Thymeleaf, che serve le pagine HTML statiche al client.
* *Controller REST API*: Due controller, *ControllerApiAdmin* e *ControllerApiUtente*, che gestiscono le chiamate RESTful, comunicano con il Service e forniscono dati dinamici al frontend.

*Architettura nel Dettaglio*  
L'architettura del progetto è suddivisa in vari componenti:

* *Controller:*
* *Controller statico (gestito tramite Spring Boot Thymeleaf) :* serve le pagine HTML al client.
* *ControllerApiAdmin*: Gestisce le chiamate API per l'amministratore.
* *ControllerApiUtente*: Gestisce le chiamate API per gli utenti, consentendo loro di interagire con il diario e i propri dati.
* *Service*:  
  I Service contengono la logica per manipolare i dati del database. I principali sono:
* *AdminService*: Gestisce la logica dell'amministratore.
* *UtenteService*: Gestisce la logica dell'utente.

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

* *Repository:*  
  Le repository sono implementate tramite le interfacce *UtenteRepository* e *DiarioRepository*.

Utilizzano Spring Data JPA per PostgreSQL e Spring Data MongoDB per MongoDB, consentendo di eseguire operazioni CRUD sui rispettivi database.

2.2 Database

Nel progetto "Diario", i database sono collegati al backend attraverso le repository, come descritto nel paragrafo *2.1 Architettura Softwar*e. Le repository interagiscono con le classi modello per effettuare operazioni CRUD sui rispettivi database, PostgreSQL e MongoDB.

* *PostgreSQL:*  
  Nel database *diario\_db* di PostgreSQL, la classe modello Utente è annotata con le convenzioni di Spring Boot 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.
* *role*: 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 ObjectId di MongoDB fa riferimento all’id di un testo creato dall’utente.
* *userId*: Campo di tipo intero fa riferimento all'id dell'utente associato, che corrisponde all'id dell'utente nel database PostgreSQL.
* *testo*: Campo di tipo String che contiene il testo scritto dall'utente.
* *dataTesto*: Campo di tipo LocalDate che indica la data in cui è stato scritto il testo.

2.3 Classi Modello

Le classi modello implementate in Java per il progetto sono:

* *Diario*: La classe Diario è annotata con *@Document(collection = "testi")* per collegarsi alla collection *"testi"* nel database MongoDB. Gli attributi della classe sono:
* *id:* ID univoco del documento (ObjectId di MongoDB).
* *userId:* ID dell'utente a cui appartiene il testo.
* *testo:* Il contenuto del diario.
* *dataTesto:* La data in cui il testo è stato scritto.
* *Utente*: La classe Utente è annotata con *@Entity* e *@Table(name = "utenti")* per collegarsi alla tabella *"utenti"* nel database PostgreSQL. Gli attributi della classe sono:
* *id:* ID univoco dell'utente (Primary Key auto-incrementale).
* *utente:* Nome utente, che deve essere unico (unique)
* *password:* La password dell'utente.
* *role*: Il ruolo dell'utente. Possono essere due *ROLE\_ADMIN* e *ROLE\_USER*

Entrambe le classi sono state arricchite con Spring Validation per garantire l'integrità dei dati. I vincoli di validazione applicati sono:

* Verifica che il nome utente non sia vuoto. Si utilizza annotazione *@NotEmpty.*
* La password deve avere almeno 4 caratteri. Si utilizza annotazione *@Size.*

2.4 Controller static Admin e Utente

I controller statici gestiscono le pagine HTML che vengono restituite come risposta alle richieste HTTP. Tutte le richieste sono tipo GET. Le pagine statiche, come *home, login, signin, logout,* e altre riservate a *Utente e Admin,* sono contenute nella cartella *resources/templates* del progetto. *Thymeleaf* riconosce automaticamente questi file e li restituisce come risposta alle chiamate effettuate tramite i rispettivi endpoint.

Ecco la mappatura delle pagine statiche per ciascun ruolo:

* */home:* Restituisce la pagina *home* (accesso libero).
* */login:* Restituisce la pagina *login* (accesso libero).
* */signin:* Restituisce la pagina *signin* (accesso libero).
* */user:* Restituisce la pagina riservata all'utente (accesso solo per utenti autenticati).
* */admin:* Restituisce la pagina riservata all'amministratore (accesso solo per amministratori).
* */admin/areautenti:* Restituisce la pagina di gestione degli utenti per l'amministratore.
* */admin/areatesti:* Restituisce la pagina di gestione dei testi per l'amministratore.

*2.5 Controller Api Admin e Utente*

Gestiscono le *chiamate API* e fungono da intermediari tra le richieste *HTTP e i service*. Ogni controller espone vari endpoint, che consentono l'interazione con le risorse tramite richieste HTTP (GET, POST, PUT, DELETE). Tuttavia, i controller non contengono alcuna logica relativa alla gestione o alla validazione dei dati: si limitano a inoltrare le richieste ai rispettivi service, che si occupano di eseguire tutte le operazioni necessarie. In seguito vengono riportati i controller api coinvolti nel progetto e le rispettive chiamate con tipologia che effettuano:

* *ControllerApiAdmin* : Questo controller gestisce le operazioni relative agli utenti e ai testi per l'area amministrativa. Permette all'amministratore di visualizzare, modificare e eliminare utenti e testi.

*1.1) Per gli utenti (AreaUtentiAdmin)*:

* *GET /admin/api/utenti:* Restituisce la lista completa degli utenti.
* *GET /admin/api/utenti/{username}*: Restituisce un singolo utente in base al nome utente.
* *PUT/admin/api/utenti/{userId}:* Modifica la password di un utente specificato tramite *userId.*
* *DELETE /admin/api/utenti/{id}:* Elimina un utente specificato tramite *id.*

*1.2) Per i testi (AreaTestiAdmin):*

* *GET/admin/api/testi:* Restituisce tutti i testi (diari) presenti nel sistema.
* *GET/admin/api/testi/{userId}:* Restituisce la lista di testi di un utente specificato tramite *userId.*
* *POST/admin/api/testi/{userId}*: Crea un nuovo testo (diario) associato all'utente specificato tramite *userId.*
* *PUT /admin/api/testi/{id}:* Modifica un testo specifico identificato tramite id.
* *DELETE /admin/api/testi/{id}:* Elimina un testo specifico tramite id.
* *ControllerApiUtente :* Questo controller gestisce le operazioni relative ai testi e alla registrazione per gli utenti. Permette agli utenti di visualizzare, creare, modificare e eliminare i propri testi, e di registrarsi nel sistema.
* Per i testi (Utente):
* *GET /user/api/testi:* Restituisce la lista dei testi (diari) dell'utente autenticato.
* *GET/user/api/username* Restituisce l’user del’utente dopo l’auticazione
* *POST /user/api/testi:* Crea un nuovo testo (diario) per l'utente autenticato.
* *PUT/user/api/testi{id}:* Modifica un testo specificato tramite id (per l'utente autenticato).
* *DELETE/user/api/testi/{id}:* Elimina un testo specificato tramite id (per l'utente autenticato).

*2.2) Per la registrazione dell'utente:*

* *POST/signin/registrazione:* Permette la registrazione di un nuovo utente creando un nuovo account con username e password.

2.6 Service

I service descritti in questo paragrafo gestiscono le operazioni legate agli utenti e ai diari. In seguito vengono riportati e descritti i metodi implementati in ogni service. Ci sono due service:

* *AdminService :* si occupa delle operazioni amministrative quali la gestione degli utenti e dei loro diari (creazione, modifica, eliminazione e recupero).

Metodi GET:

* *dammiListaUtentiAdmin():* Restituisce una lista di tutti gli utenti registrati.
* *dammiUtentePerUsernameAdmin(username):* Restituisce un utente in base al suo username.
* *dammiTuttiTestiAdmin():* Restituisce tutti i testi (diari) presenti nel sistema.
* *dammiListaTestiAdminPerUtente(userId):* Restituisce tutti i testi scritti da un utente specifico, identificato dal suo userId.

Metodi POST:

* *creaTestoAdmin(diario, userId):* Crea un nuovo testo (diario) associato a un utente, impostando la data e il userId.

Metodi PUT:

* *modificaTestoAdminPerId(diario, id):* Modifica il testo di un diario esistente, identificato tramite id.
* modificaPasswordUtenteAdmin(utente,userId): Modifica la password di un utente (utilizzando la crittografia BCrypt), identificato dal suo *userId.*

Metodi DELETE:

* *eliminaUtenteAdminPerId(id):* Elimina un utente dal sistema identificato dal suo *id.*
* *eliminaTestoAdminPerId(id):* Elimina un diario specifico, identificato tramite il suo *id.*
* *UtenteService :* svolge le operazioni specifiche dell'utente, come la creazione e la gestione dei propri diari, la registrazione e la modifica delle proprie informazioni.

Metodi GET:

* *dammiListaTestiPerUtente():* Restituisce una lista di testi (diari) scritti dall'utente autenticato.
* *dammiTestoPerId(id):* Restituisce un diario specifico in base all'*id.*

Metodi POST:

* *creaTesto(diario):* Crea un nuovo diario associato all'utente autenticato e imposta la data di scrittura.
* *registraUtente(utente):* Registra un nuovo utente nel sistema, dopo aver verificato che non esista già un utente con lo stesso username e cifrando la sua password.
* Metodi PUT:
* *modificaTestoPerId(diario, id):* Modifica un diario esistente, identificato dal suo *id,* aggiornando il testo e la data.
* Metodi DELETE:
* *eliminaTestoPerId(id):* Elimina un diario esistente identificato dal suo *id.*

2.7 Repository

I repository utilizzati sono annotati con *@Repository* per indicare che si tratta di un componente che gestisce la persistenza dei dati. Le repository estendono le interfacce di persistenza specifiche per il tipo di database: *MongoRepository* per MongoDB e *CrudRepository* per PostgreSQL. I metodi definiti in queste interfacce sono responsabili di eseguire le operazioni CRUD (creazione, lettura, aggiornamento e cancellazione) dei dati.

* *Repository per MongoDB (IDiarioRepository) :* Gestisce le operazioni sui testi (diari) e include metodi personalizzati, come *findByUserId* per ottenere i diari di un utente specifico.
* *Repository per PostgreSQL (IUtenteRepository) :* Gestisce le operazioni sugli utenti e include un metodo per cercare un utente tramite il nome utente.

2.8 Sicurezza (autenticazione utente e autorizzazioni accesso a endpoint)

Per gestire l'autenticazione degli utenti e le autorizzazioni di accesso agli endpoint, è stato utilizzato *Spring Security.* In particolare, è stato implementato un sistema di autenticazione basato su username e password, con codifica delle password tramite *BCrypt.*

Per realizzare questo, è stato implementato un service personalizzato, chiamato *CustomUserDetailsService*, e una classe di configurazione annotata con *@Configuration* denominata *SecurityConfig.*

Inoltre, sono stati definiti ruoli specifici per gli utenti, come *"ROLE\_USER*" e *"ROLE\_ADMIN"*, per garantire un accesso controllato ai vari endpoint del sistema.

* *CustomUserDetailsService:* Questa classe gestisce il caricamento e l'autenticazione degli utenti, recuperando i dati degli utenti dal database tramite il repository *IUtenteRepository.* In particolare:
* Il metodo *loadUserByUsername* carica i dettagli dell'utente (username, password, e ruolo) e li restituisce per l'autenticazione.
* Il metodo *getAuthenticatedUserId* permette di ottenere l'ID dell'utente attualmente autenticato, sfruttando il contesto di sicurezza di Spring Security.
* SecurityConfig: Questa classe definisce la configurazione di sicurezza per l'applicazione, stabilendo le regole di accesso agli endpoint:
* Gli endpoint,  */home, /login, /signin* sono accessibili a tutti.
* Gli endpoint sotto */admin/\*\** sono accessibili solo agli utenti con il ruolo *"ADMIN"*, mentre quelli sotto */user/\*\** sono riservati agli utenti con il ruolo *"USER".*
* La configurazione include anche la gestione del login tramite una pagina personalizzata *(/login),* che indirizza gli utenti alla pagina appropriata in base al loro ruolo *admin o user).*
* Il /logout è accessibile solo agli utenti connessi qualora si prova ad entrare all’endpoint /logout senza accessi si viene rimandati alla pagina di login

**Frontend**

3.1 Html e css  
Queste sono le pagine HTML statiche con CSS per il frontend del progetto "Diario". Ogni pagina è associata a un file CSS che definisce lo stile visivo. Di seguito è descritta la struttura di ciascuna pagina e le azioni permesse all'utente

* *Pagina Admin:*
* Un'intestazione di benvenuto con il nome dell'admin.
* Due pulsanti: uno per "Gestire Utenti" e uno per "Gestire Testi".
* Azioni permesse all'utente: Solo l'admin può accedere a questa pagina e può gestire gli utenti e i testi tramite i pulsanti.
* *Pagina Gestione Testi (Admin):*
* Un'intestazione "Gestione Testi".
* Una barra di ricerca per cercare i testi per utente.
* Una lista di testi con opzioni per "Modificare" o "Eliminare" ciascun testo.
* Azioni permesse all'utente: Solo l'admin può gestire i testi degli utenti, con la possibilità di modificarli o eliminarli.
* *Pagina Gestione Utenti (Admin):*
* Un'intestazione "Gestione Utenti".
* Una barra di ricerca per cercare gli utenti tramite username o ID.
* Una lista di utenti con opzioni per "Modificare Password" o "Eliminare" ciascun utente.
* Azioni permesse all'utente: Solo l'admin può gestire gli utenti, modificando password o eliminandoli.
* *Home:*
* Un'intestazione con il titolo del progetto.
* Un'immagine centrale.
* Un'introduzione al progetto con un breve testo.
* Due pulsanti: uno per "Registrati" e uno per "Accedi".
* Azioni permesse all'utente: Gli utenti possono registrarsi o accedere tramite i pulsanti, se non hanno già un account.
* *Pagina Login:*
* Un'intestazione "Login".
* Due campi di input: uno per l'username e uno per la password.
* Un pulsante per inviare il login.
* Un link per la registrazione (nel caso l'utente non abbia un account).
* Azioni permesse all'utente: Gli utenti con un account possono effettuare il login.
* *Pagina Signin:*
* Un'intestazione "Registrazione".
* Due campi di input: uno per l'username e uno per la password.
* Un pulsante per inviare la registrazione.
* Un link per il login (nel caso l'utente abbia già un account).
* Azioni permesse all'utente: Gli utenti che non hanno un account possono registrarsi tramite il modulo.
* *Pagina Utente:*
* Un'intestazione con il nome dell'utente.
* Una tabella che visualizza i testi dell'utente, con opzioni per "Modificare" o "Eliminare" ciascun testo.
* Un pulsante per "Creare un nuovo testo" che mostra un'area di input per scrivere.
* Azioni permesse all'utente: L'utente può visualizzare, modificare ed eliminare i propri testi, oltre a creare nuovi testi nel proprio diario.

3.2 Javascript

Per gestire la dinamica delle pagine statiche e per collegare il front-end al back-end, è stato utilizzato JavaScript insieme alle Fetch API. Le Fetch API permettono di fare richieste HTTP asincrone dal browser verso il server senza dover ricaricare la pagina, facilitando l'interazione con il backend. Questo approccio è stato utilizzato per inviare e ricevere dati tra il client e il server, gestendo operazioni come il caricamento, la creazione, la modifica e l'eliminazione di dati in modo fluido e reattivo. In seguito viene riportata dal descrizione dell’utilizzo del js pe pagina statica:

*1. Admin*

Eventi Listener:

* *Gestisci Utenti:* Reindirizza l'admin alla pagina */admin/areautenti* per la gestione degli utenti.
* *Gestisci Testi:* Reindirizza l'admin alla pagina */admin/areatesti* per la gestione dei testi.

*2. Area testi admin*

Il JavaScript tramite chiamate fetch si occupa di:

* Caricamento della lista dei testi: GET/admin/api/testi.
* Ricerca dei testi per ID utente: GET/admin/api/testi/{id}.
* Modifica di un testo: PUT/admin/api/testi/{id}.
* Eliminazione di un testo: DELETE/admin/api/testi/{id}.

Eventi Listener:

* *Bottone "Cerca":* Effettua una ricerca dei testi tramite ID.
* *Bottone "Modifica Testo"*: Apre una finestra per modificare il testo selezionato e salva le modifiche.
* *Bottone "Elimina":* Chiede conferma ed elimina il testo selezionato.

*3. Area utenti admin*

Il JavaScript tramite chiamate fetch si occupa di:

* *Caricamento utenti:* *GET/admin/api/utenti:* Popola dinamicamente la lista utenti.
* *Ricerca utente:* *GET/admin/api/utenti/{username}:* Mostra il risultato filtrato.
* *Modifica password:* *PUT/admin/api/utenti/{id}:* Aggiorna la password di un utente tramite prompt.
* *Eliminazione utente:* *DELETE/admin/api/utenti/{id}:* Rimuove un utente con conferma e ricarica la lista.

*Eventi Listener:*

* *Cerca:* Verifica input, chiama l’API e aggiorna la lista.
* *Modifica Password:* Prompt per nuova password e invio richiesta PUT.
* *Elimina:* Conferma eliminazione, invio richiesta DELETE, ricarica la lista.

*4. Home*

Eventi Listener:

* *Registrati:* Il bottone reindirizza l'utente alla pagina di registrazione *(/signin)*, utilizzando il comando *location.href='/signin'.*
* *Accedi:* Il bottone reindirizza l'utente alla pagina di login *(/login),* utilizzando il comando *location.href='/login'.*

*5. Login*

Eventi Listener:

* *Accedi:*
* Invia i dati inseriti nel modulo (username e password) al server con un'azione POST tramite il metodo del modulo stesso (action="/login").
* Se i dati sono corretti, l'utente viene autenticato e reindirizzato alla pagina successiva, altrimenti riceverà un errore di login.
* *Non hai un account?:*
* Viene mostrato un link che rimanda alla pagina di registrazione per i nuovi utenti.

*6. Signin*

Il JavaScript tramite chiamate fetch si occupa di:

* *Registrazione utente: POST/signin/registrazione:* Invia i dati del modulo (username, password) al backend per creare un nuovo utente.

Eventi Listener:

* Submit del modulo:
* Previene il comportamento predefinito del form.
* Recupera i valori di input (username, password).
* Invia i dati con fetch e gestisce: se *successo:* Mostra un messaggio di conferma (es. "Registrazione riuscita") altrimenti se *errore*: Mostra un messaggio d’errore.

*7. Pagina Utente*

Il JavaScript tramite chiamate fetch si occupa di:

* *GET:* Recupera la lista dei testi da */user/api/testi* e li mostra nella tabella.
* *POST*: Invia un nuovo testo a */user/api/testi* quando l'utente lo crea.
* *PUT*: Aggiorna un testo specifico inviando una richiesta a */user/api/testi/{id}.*
* *DELETE:* Elimina un testo specifico tramite una chiamata a */user/api/testi/{id}.*

Eventi Listener:

* *Aggiungi Testo:*
* Mostra l'area di testo nascosta per scrivere un nuovo testo.
* Quando confermi, invia i dati con una chiamata POST.
* Ricarica la pagina per aggiornare la lista.
* *Modifica Testo:*
* Mostra un prompt con il testo attuale.
* Invia i dati aggiornati con una chiamata PUT.
* Ricarica la pagina per aggiornare la lista.
* *Elimina Testo:*
* Mostra un dialogo di conferma.
* Se confermato, invia una richiesta DELETE.
* Ricarica la pagina per aggiornare la lista.