Immagine che contiene schizzo, disegno, arte, bianco e nero

Descrizione generata automaticamente

Scuola di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato del corso di:

SOFTWARE ARCHITECTURES AND METHODOLOGIES – 9 CFU

2023-2024

MyPortfolio:

Implementazione di una piattaforma Backend + Frontend con architettura API RESTful

Andrea Neri - 7060638

Sommario

[Introduzione 5](#_Toc175847674)

[Obiettivi e intenti dell'elaborato 5](#_Toc175847675)

[Analisi dei requisiti 6](#_Toc175847676)

[Vincoli 6](#_Toc175847677)

[Requisiti non funzionali 6](#_Toc175847678)

[Requisiti funzionali 6](#_Toc175847679)

[Analisi di progettazione 8](#_Toc175847680)

[Gestione e organizzazione delle immagini 8](#_Toc175847681)

[Accesso e permessi utente 8](#_Toc175847682)

[Visualizzazione e Copyright delle Immagini 8](#_Toc175847683)

[Sezione Shop (estensione del progetto) 9](#_Toc175847684)

[Considerazioni sulla scalabilità e manutenibilità 9](#_Toc175847685)

[Domain Model 9](#_Toc175847686)

[Analisi diagrammi di progetto 10](#_Toc175847687)

[UML Diagram 11](#_Toc175847688)

[Deployment Diagram 12](#_Toc175847689)

[Frontend 13](#_Toc175847690)

[Backend 13](#_Toc175847691)

[Sequence Diagram 14](#_Toc175847692)

[Use Case Diagram 16](#_Toc175847693)

[Descrizione dei package 17](#_Toc175847694)

[Implementazione lato JAVA 18](#_Toc175847695)

[it.myportfolio.configuration.myportfolioApplication.java 19](#_Toc175847696)

[Annotations 22](#_Toc175847697)

[it.myportfolio.model 23](#_Toc175847698)

[Work 23](#_Toc175847699)

[ImageProject 24](#_Toc175847700)

[ShopableImage 25](#_Toc175847701)

[User 26](#_Toc175847702)

[ERole 27](#_Toc175847703)

[Role 28](#_Toc175847704)

[Cart 28](#_Toc175847705)

[SalesOrder 29](#_Toc175847706)

[it.myportfolio.dto 30](#_Toc175847707)

[UserPersonalDetailsDTO vs User 31](#_Toc175847708)

[SignupRequest 32](#_Toc175847709)

[DetailsSalesOrderDTO 33](#_Toc175847710)

[SimpleShopableImageDTO 34](#_Toc175847711)

[it.myportfolio.mapper 34](#_Toc175847712)

[it.myportfolio.service 35](#_Toc175847713)

[it.myportfolio.repository 39](#_Toc175847714)

[Spring Data JPA 39](#_Toc175847715)

[JpaRepository 40](#_Toc175847716)

[Metodi CRUD 41](#_Toc175847717)

[Query Methods 41](#_Toc175847718)

[ImageRepository 42](#_Toc175847719)

[WorkRepository 42](#_Toc175847720)

[UserRepository 43](#_Toc175847721)

[it.myportfolio.controller 43](#_Toc175847722)

[Gestione Copyright delle immagini 45](#_Toc175847723)

[API security: gestione autenticazione e autorizzazione nelle API 46](#_Toc175847724)

[WebSecurityConfig 50](#_Toc175847725)

[UserDetailsServiceImpl 53](#_Toc175847726)

[UserDetailsImpl 53](#_Toc175847727)

[AuthTokenFilter 54](#_Toc175847728)

[AuthEntryPointJwt 55](#_Toc175847729)

[AuthController 56](#_Toc175847730)

[JwtUtils 58](#_Toc175847731)

[API security: gestione sicurezza immagini(x) 60](#_Toc175847732)

[Descrizione componenti di utility 60](#_Toc175847733)

[Implementazione lato DB 61](#_Toc175847734)

[Implementazione front-end 64](#_Toc175847735)

[Sezione shop e blockchain 64](#_Toc175847736)

[API 65](#_Toc175847737)

[Analisi API - tabella di copertura 66](#_Toc175847738)

# Introduzione

L'idea del progetto nasce da una richiesta di un amico fotografo che aveva il problema di condividere in modo selettivo e con autenticazione alcune cartelle del suo NAS, contenenti i sample di progetti fotografici con collaboratori, clienti e futuri clienti.

Il primo tentativo ipotizzato è stato quello di sfruttare una delle funzionalità del NAS, ovvero l’accesso ai file tramite una app messa a disposizione dal produttore. Inoltre, il NAS permette la creazione di utenze, con la possibilità di definire l’accesso alle share folder in modo granulare. Seppure comoda come soluzione, il dover condividere sulla rete pubblica l’intera library (se pur passando dai server/app fornite dal produttore) del NAS, è stata scartata.

Il secondo tentativo, prima dell’idea di questo progetto ed ancora in uso, era l’invio tramite piattaforme come WeTransfer (e simili) dei sample dei clienti e dei futuri clienti; questa modalità risulta molto scomoda, lenta e temporanea, in quanto il link per il download ha una scadenza.

Le richieste fondamentali erano la possibilità di far visualizzare immagini protette da watermark (copyright) ad utenti registrati ed ai quali erano state fornite le autorizzazioni di visualizzazione.

La catalogazione delle foto sarà gestita tramite raccolte (definite Work) ognuna delle quali rappresenterà un singolo progetto fotografico (es. shooting fotografico per l’azienda ACME corp.) o un insieme di lavoro accumunati da una stessa caratteristica (es. fotografia paesaggistica), questo permetterà una gestione fine degli accessi.

# Obiettivi e intenti dell'elaborato

* Fornire un sito web fruibile sia da PC che smartphone, che oltre ad avere una homepage, una sezione di presentazione e una di contatto, avesse un’area dedicata a cui accedere solo tramite login (username e password)
* Fornire una piattaforma per la condivisione selettiva e autenticata di shooting fotografici raggruppati in Work (questo concetto verrà esposto nelle prossime sezioni).
* Avesse una visualizzazione in base al ruolo assegnato all’utente che si autentica
* L’utenza standard, una volta registrato e loggato, potrà accedere in visualizzazione alla/e cartella/e in base alle autorizzazioni fornite dall’amministratore.
* La visualizzazione della foto sarà tramite una galleria fotografica (una per ogni cartella o area di lavoro al quale quell’utente ha accesso) sfruttando le thumbnail in modo da rendere più veloce il caricamento della pagina, che al click verranno mostrate a tutto schermo.

Volendo stilare un elenco delle funzionalità/sezioni minime da sviluppare e rendere disponibili lato front-end per l’utilizzatore finale, queste potrebbero essere riassunte così:

* Homepage,
* Pagina di presentazione
* Pagina contatti
* Login/Registrazione
* Pannello di amministrazione per poter caricare nuove foto e cancellare/modificare quelle già presenti
* Pannello di amministratore per poter gestire i permessi di accesso sulle cartelle
* Sezione galleria in cui verranno mostrate le foto presenti in una singola cartella
* Realizzazione delle gallerie fotografiche tramite thumbnail in modo da rendere il caricamento delle pagine più veloci
* Sia le thumbnail che le immagini a dimensioni reali dovranno presentare un watermark per proteggere gli scatti da eventuali furti.

Come estensione al progetto di base (che rimarrà solo per scopi didattici) è stato pensato di aggiungere una sezione Shop. In questa sezione l’amministratore potrà inserire un certo numero di foto che vorrà vendere (in copia unica, essendo opere artistiche). Gli utenti una volta loggati potranno inserire nel carrello le foto che vorranno e procedere all’acquisto.

L’acquisto verrà registrato all’interno di blockchain per certificare l’acquisto. Quest’ultima caratteristica è stata realizzata tramite Rest API fornire da una libreria esterna Open Source che registra le transazioni su un fork della blockchain Ethereum.

# Analisi dei requisiti

## Vincoli

1. Le foto sono già presenti in un NAS su rete LAN privata. Le immagini non devono essere caricate sul Database come Blob, ma verranno registrati solo gli URL.
2. Le foto dovranno essere necessariamente caricate in formati fotografici standard (PNG, JPEG, JPG, BMP). Non potranno essere caricati i cosiddetti "scatto grezzo", formati fotografici non renderizzati come NEF, CR2, CR3, ARW e RAF.

## Requisiti non funzionali

1. Architettura del Sistema
   1. L'architettura del sistema deve essere basata sul modello MVC.
   2. Il back-end deve essere sviluppato utilizzando JakartaEE con il framework Spring.
   3. Il front-end deve essere sviluppato utilizzando Vue JS e Bootstrap.
   4. L'architettura deve garantire la portabilità del sistema.
2. Gestione del Copyright
   1. Il sistema deve garantire il rispetto del copyright sulle opere

## Requisiti funzionali

1. Tipologie utenze e azioni permesse
   1. L'applicazione deve prevedere 2 tipolgie di utenza: Admin e User
   2. L'utenza con Role User deve avere la possibilità di visualizzare i work a cui gli è stato fornito l'accesso
   3. L'utenza con Role Admin deve aver la possibilità di visualizzare tutti i work disponibili
   4. L'utenza con Role Admin deve aver la possibilità di accedere ai pannelli di amministrazione sia della sezione Galleria che Shop
   5. L'utenza con Role User deve accedere e fare acquisti nella sezione Shop
2. Gestione pannello di controllo dell'amministratore
   1. L'amministratore deve poter accedere a un pannello di controllo.
   2. Dal pannello di controllo, l'amministratore deve poter caricare risorse (immagini).
   3. Gestione Permessi di Lettura
   4. Dal pannello di controllo, l'amministratore deve poter gestire i permessi di lettura per i singoli "work".
3. Gestione immagini
   1. Le risorse caricate devono poter essere raggruppate per "work" (es. shooting fotografico per l’azienda ACME corp. o un insieme di lavori accomunati da una stessa caratteristica, come fotografia paesaggistica).
   2. L'applicativo deve supportare il caricamento, la catalogazione e il recupero di risorse immagine nei formati PNG, JPEG, JPG e BMP.
   3. Il sistema deve generare automaticamente le miniature (thumbnail) per ogni immagine caricata per ottimizzare i tempi di caricamento delle pagine.
   4. La creazione delle miniature deve avvenire al click del pulsante "Genera Thumbnail" e le miniature devono essere salvate nella sottodirectory 'thumbnail' nella stessa cartella delle immagini originali.
   5. Il sistema deve garantire l’applicazione di un watermark on demand dal lato back-end, nel momento in cui viene richiesta la visualizzazione di un’immagine a risoluzione standard
   6. Il sistema deve permettere di applicare in modo statico un watermark sulle thumbnail
4. Gestione fine dei permessi di autorizzazione e accesso
   1. Gli utenti devono registrarsi alla piattaforma per accedere alle risorse.
   2. Ogni utente deve avere autorizzazioni mirate su cartelle specifiche.
   3. Alcune immagini utilizzate nella homepage devono avere visibilità pubblica senza necessità di login.
   4. L'accesso alle foto private deve avvenire solo dopo login dell'utente.
   5. L'utente deve poter visualizzare solo le foto per cui ha i diritti di accesso.
5. Componenti Java riusabili
   1. Il sistema deve includere componenti Java riusabili per la generazione automatica delle miniature delle immagini.
   2. Il sistema deve consentire l'aggiunta di un watermark personalizzato alle risorse immagine su richiesta (on demand).
6. Sezione Shop e gestione acquisti
   1. L'applicativo deve prevedere una sezione "Shop" dove saranno presenti alcune immagini selezionate dall'amministratore, simili a NFT.
   2. Gli utenti devono poter acquistare le immagini presenti nella sezione "Shop".
   3. Gli acquisti devono essere registrati su un sistema blockchain utilizzando le API fornite da un provider di tecnologia blockchain.

# Analisi di progettazione

L'obiettivo principale è la creazione di un sistema efficiente e sicuro per la gestione, visualizzazione e vendita delle immagini, sfruttando le tecnologie moderne e garantendo la protezione dei diritti d'autore.

## Gestione e organizzazione delle immagini

Poiché le foto sono già presenti sul NAS, si è deciso di non caricarle su un database come BLOB (Binary Large Object). Invece, verranno registrati solo gli URL delle immagini all’interno di un campo testuale nel database. Questo approccio offre diversi vantaggi:

* Efficienza del database: poiché le query dovranno recupere solo stringhe testuali, sia ha una riduzione del carico di lavoro sul database, migliorando le prestazioni complessive del sistema.
* Minor occupazione di spazio: registrando sul database solo gli URL delle foto si è evitato di avere un Database grande in termini di occupazione di memoria
* Accesso rapido: Gli URL consentono un accesso diretto e rapido ai file immagine, migliorando l'esperienza utente.
* Tempistiche di Facilità di Backup ridotte.

Per garantire la sicurezza e la corretta visualizzazione delle immagini, sarà implementata una ACL (Access Control List) ad hoc sul NAS. Questa lista di controllo degli accessi permetterà di definire l’indirizzo IP sorgente del server dove sarà installato l’applicativo.

Per una gestione più pratica, le singole immagini verranno caricate e organizzate raggruppandole per "work". Un "work" può essere definito come un progetto fotografico (es. shooting fotografico per l’azienda ACME corp.) o un insieme di lavori accomunati da una stessa caratteristica (es. fotografia paesaggistica). Questa organizzazione permetterà una più facile gestione delle risorse e un accesso più intuitivo per gli utenti, ottenendo:

* Facilità di navigazione: gli utenti a trovare facilmente le risorse di cui hanno bisogno.
* Gestione Efficace: gli amministratori potranno gestire in modo efficace e organizzato le immagini caricate
* Gestione fine dei permessi di visualizzazione

## Accesso e permessi utente

Gli utenti dovranno registrarsi alla piattaforma per accedere alle risorse. Una volta completata la registrazione ed effettuato il login, l'amministratore (utente con ruolo Admin) potrà concedere i permessi di visualizzazione dei "work" agli utenti. Sarà predisposta una sezione contatti all’interno dell’applicazione, dove gli utenti potranno fare richieste di visualizzazione, fornendo una descrizione della tipologia di immagini che desiderano vedere.

## Visualizzazione e Copyright delle Immagini

Gli utenti potranno visualizzare solo le immagini per cui hanno i diritti di accesso tramite una gallery.   
La pagina verrà carica utilizzando i thumbnail generati staticamente, salvati nella sub-directory e con il watermark già presente.  
Per le immagini a dimensione reale si è deciso di non applicare il watermark in modo statico, ma per garantire il rispetto del copyright, si è deciso di generare le immagini con watermark lato back-end. Questo approccio eviterà di dover salvare sul NAS sia le foto originali che quelle con watermark, riducendo l’occupazione di spazio e semplificando la gestione delle risorse. Le immagini con watermark saranno generate dinamicamente quando richieste, garantendo così la protezione dei diritti d'autore senza compromettere la qualità del servizio.

## Sezione Shop (estensione del progetto)

Come estensione del progetto, che non sarà richiesta nell’applicativo che andrà in produzione inizialmente, si realizzerà una sezione shop. In questa sezione, l’amministratore potrà caricare immagini da vendere come opere uniche. Gli utenti potranno acquistare queste immagini, e ogni acquisto sarà registrato su un sistema blockchain (nelle prossime sezioni analizzeremo questo aspetto). Questo processo sfrutterà API messe a disposizione da un provider di tecnologia blockchain, garantendo la tracciabilità e la sicurezza delle transazioni.

## Considerazioni sulla scalabilità e manutenibilità

Il sistema è progettato per essere scalabile e mantenibile. Utilizzando un'architettura basata su JakartaEE con il framework Spring per il back-end e Vue JS con Bootstrap per il front-end, si garantisce una separazione chiara delle responsabilità e una facile manutenibilità.

**Scalabilità orizzontale:** Il sistema può essere scalato orizzontalmente aggiungendo più server man mano che la base di utenti cresce. Questa capacità garantisce che le prestazioni del sistema rimangano ottimali anche con un numero crescente di utenti e carichi di lavoro.

**Modularità:** La modularità del sistema permette di aggiungere nuove funzionalità senza compromettere quelle esistenti. I componenti del sistema sono progettati per essere indipendenti, facilitando l'integrazione di nuove caratteristiche e miglioramenti in modo sicuro ed efficiente.

**Aggiornamenti facili:** L'architettura moderna facilita l'aggiornamento del sistema con nuove tecnologie e miglioramenti. Le tecnologie adottate, come Spring e Vue JS, sono ampiamente supportate e aggiornate, garantendo che il sistema possa evolvere e incorporare rapidamente nuove funzionalità e patch di sicurezza.

**Applicazione Responsiva:** Utilizzando Bootstrap per il front-end, l'applicazione è completamente responsive. Questo significa che l'interfaccia utente si adatta automaticamente a diverse dimensioni di schermo e dispositivi, garantendo un'esperienza utente ottimale sia su desktop che su dispositivi mobili.

**Creazione di App native:** Utilizzando le API, è possibile creare app native per Android e iPhone senza dover utilizzare un browser.

# Domain Model

Dopo avere definito tutte quelle che sono le richieste e i requisiti dell’utilizzatore finale, un ulteriore passo da considerare è l’analisi del *domain model* (modello di dominio), che fornisce a tutti coloro che devono lavorare su un sistema una base comune di concetti su cui ragionare e una terminologia condivisa rigorosa e specifica.

In questa analisi di definiscono modelli concettuali (UML, Use Case, Sequence …) che forniranno una guida per la realizzazione del codice.

A partire quindi da vincoli, requisiti non funzionali e requisiti funzionali sono state estrapolate le seguenti entità:

* User: per rappresentare un utente all’interno del sistema.
* Work: rappresenta un lavoro fotografico.
* ImageProject: rappresenta un'immagine associata a un progetto.
* ShopableImage: rappresenta un'immagine disponibile per l'acquisto all’interno dello shop.
* Cart: rappresenta il carello all’interno della sezione shop.
* SalesOrder: rappresenta un ordine di acquisto.
* Role: rappresenta il ruolo di un utente (attualmente i ruoli presenti sono Admin e User).

e le seguenti relazioni tra entità:

* Ad un User possono essere associati uno o più Role
* Un User ha una relazione di visibilità con Work (associazione 0..N).
* Ogni Work contiene ImageProject (aggregazione 1..N).
* Ogni User ha un Cart (associazione 1..1).
* Cart contiene ShopableImage (associazione 1..N).
* SalesOrder contiene ShopableImage (associazione 1..N).
* Un User può aver associato un SalesOrder (associazione 0..N).

# Analisi diagrammi di progetto

A partire dall’analisi fatta per il domain model sono stati realizzati i 4 diagrammi seguenti.

# UML Diagram

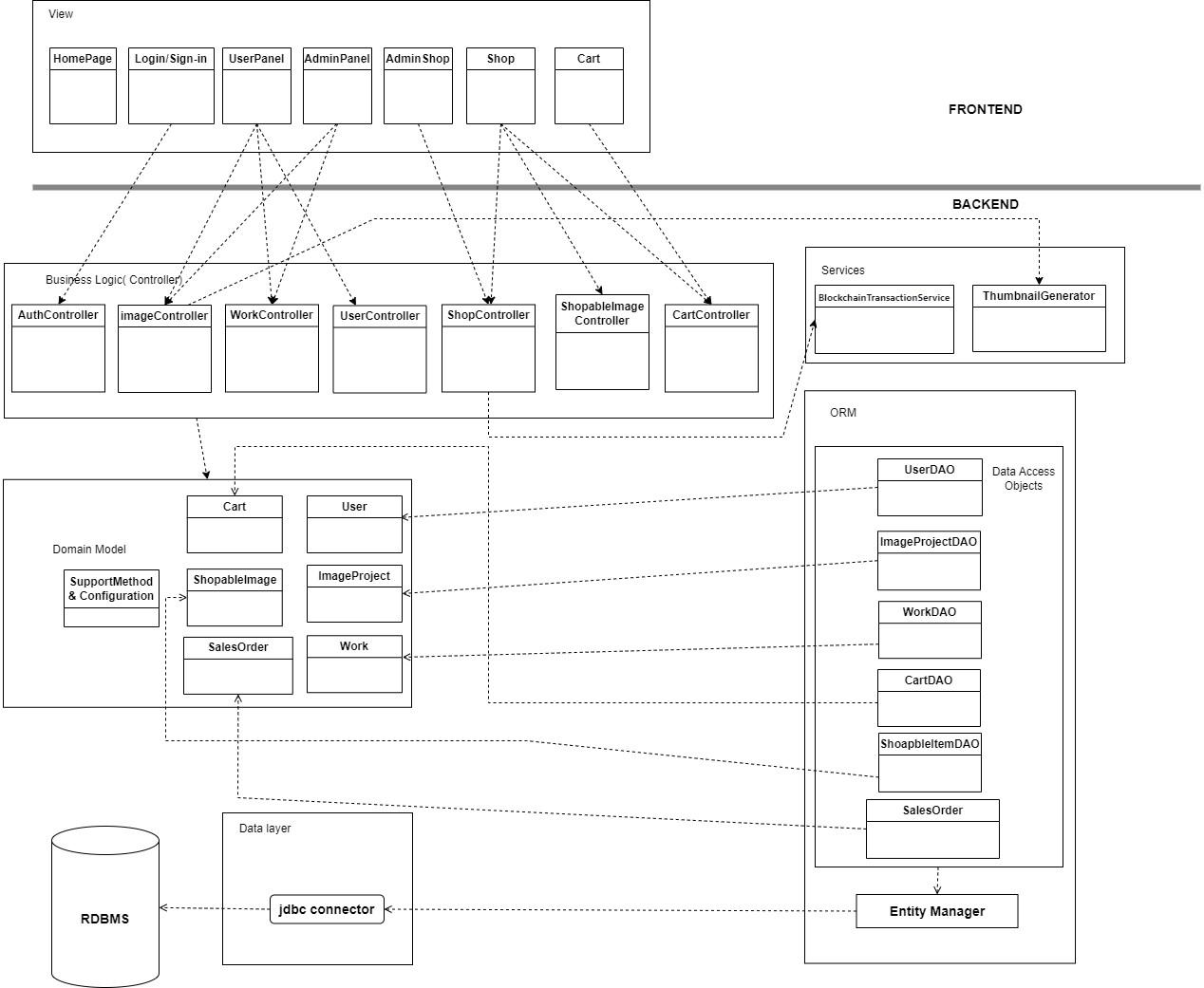
Immagine che contiene schermata, diagramma, design

Descrizione generata automaticamente

Descrizione entità:

* **Role**: rappresenta i ruoli che uno User può avere, grazie a questo attributo è possibile gestire le autorizzazioni sulle chiamate API in modo selettivo. Attualmente i ruoli configurati sono 2: Admin e User.
* **User:** modello generico dell’utente. Ha una relazione N-N con la classe Role, poiché un utente può avere entrambi i ruoli. Ha una relazione N-N con la classe Work, risolta a livello database dalla tabella intermedia visibleWork. Grazie a questo attributo è possibile gestire i permessi di visualizzazione dei Work.  
  Ogni utente ha associato un oggetto Cart, che rappresenta il carrello temporaneo (prima di procedere con l’acquisto vero e proprio) nello shop.
* **Work:** modello che permette di raggruppare le ImageProject appartenenti allo stesso shooting o accomunate da una stessa caratteristica
* **ImageProject:** Rappresenta i singoli scatti. Tramite gli attributi URL e thumbnailURL sarà possibile “scaricarle dal NAS” e visualizzarle sul Client
* **Cart:** permette la memorizzazione temporanea delle opere che un utente vorrebbe comprare
* **ShopableImage:** rappresenta le “opere” vendibili. Questa classe è ottenuta come estensione della classe ImageProject. Vengono aggiunti i parametri isSold e price.
* **SalesOrder:** classe che permette la registrazione delle varie vendite, memorizzando chi e quando sono state acquistate determinate opere.  
  le informazioni che memorizza questa classe dovranno essere successivamente salvate in una Blockchain.

## Deployment Diagram



Grazie al deployment diagram possiamo vedere i componenti principali dell'architettura del sistema.

Possiamo notare la suddivisione netta tra i componenti di front-end e quelli di back-end, le relazioni tra i vari componenti e come si interfacciano tra di loro.

### Frontend

Il front-end è composto da diverse pagine html (viste), che grazie al framework VueJS, interagiscono con i vari controller presenti nel back-end per fornire funzionalità all'utente finale.  
L’interazione avviene tramite chiamate API RESTFul con tre livelli di autenticazione in base al Role dell’utente (nessun ruolo, User, Admin).

* **HomePage:** la pagina principale dell'applicazione, da dove è possibile recuperare le varie informazioni del fotografo ed iniziare l’interazione con l’applicazione.
* **Login/Sign-in**: pagina per l'autenticazione e la registrazione degli utenti.
* **UserPanel**: gli utenti “loggati” da questa pagina potranno visualizzare le gallerie fotografiche per le quali hanno i diritti di visualizzazione.
* **AdminPanel**: pannello di controllo per gli amministratori. Questa è la sezione principale per gli amministratori del sistema: da qui potranno creare/modificare/cancellare le raccolte fotografiche (Work), potranno caricare/modificare/cancellare immagini e potranno fornire o revocare i permessi di visualizzazione agli utenti.
* **Admin Shop**: pannello di controllo per gli amministratori per la gestione dello Shop.
* **Shop**: Sezione per la visualizzazione e l'acquisto di immagini da parte dell’utenza.
* **Cart:** Sezione per la gestione del carrello degli acquisti e la visualizzazione dei propri acquisti.

### Backend

Il back-end è composto dai Controller che espongono API, dai servizi (classi di ausilio), dal Domain Model, dall’ ORM, dal JDBC per l’interazione con il Database e il DB MySQL per la memorizzazione dei dati.

**Business Logic (Controller)**

* **AuthController:** Gestisce l'autenticazione e la registrazione degli utenti.
* **ImageController:** Gestisce le operazioni relative alle immagini.
* **WorkController:** Gestisce le operazioni relative ai lavori (work).
* **UserController:** Gestisce le operazioni relative agli utenti.
* **ShopController:** Gestisce le operazioni relative al negozio online.
* **ShopableImageController:** Gestisce le operazioni relative alle immagini acquistabili.
* **CartController:** Gestisce le operazioni relative al carrello.

**Services**

* **BlockchainTransactionService:** Gestisce le transazioni sulla blockchain per la vendita delle immagini.
* **ThumbnailGenerator:** Genera le miniature delle immagini presenti sul NAS.

Nel **Domain Model** sono realizzate le Entities, che permettono di rappresentare la semantica degli oggetti di dominio e i **relativi Data Access Objects (DAO)** per la gestione della persistenza delle entità nel database.

**SupportMethod & Configuration** contiene i metodi di supporto e configurazioni dell’applicazione

Il livello dei dati o **Data Layer** ha il compito di interfacciarsi con il database relazionale (RDBMS) tramite un connettore JDBC per eseguire operazioni di lettura e scrittura (operazioni CRUD).

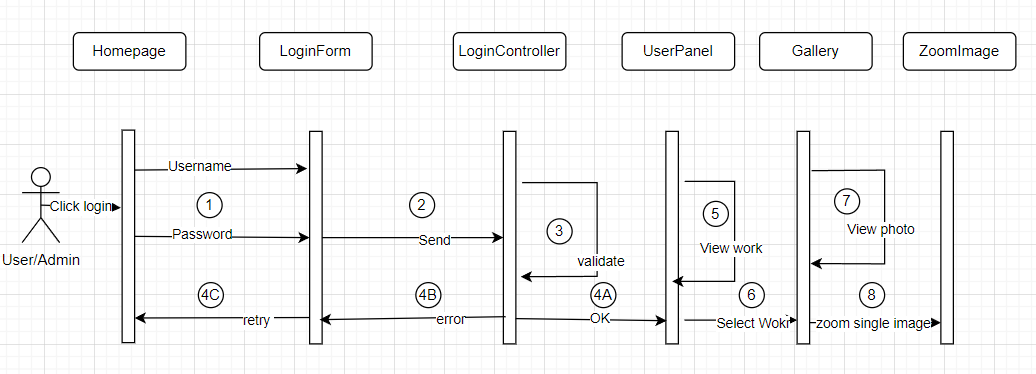
Grazie a questa suddivisione, si ottengono due vantaggi molto importanti:

* **Separazione delle responsabilità**: l'architettura del sistema garantisce una chiara separazione delle responsabilità tra le varie componenti, facilitando la manutenibilità e la scalabilità del sistema.
* **Modularità:** la modularità del sistema permette di aggiungere nuove funzionalità e componenti senza compromettere quelle esistenti, grazie alla suddivisione in controllori, servizi e DAO.

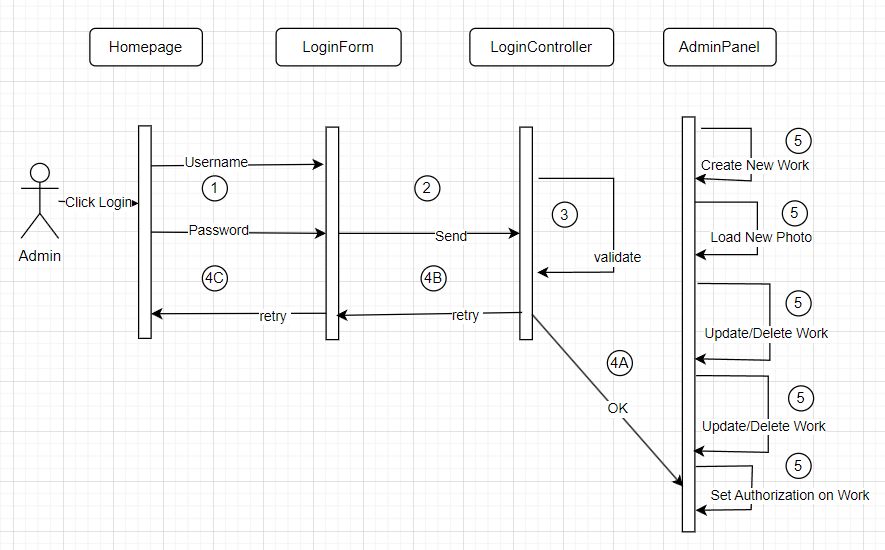
## Sequence Diagram

Un utente (Admin o User) dopo essersi registrato ed aver eseguito il login, potrà accedere alla propria galleria di immagini.

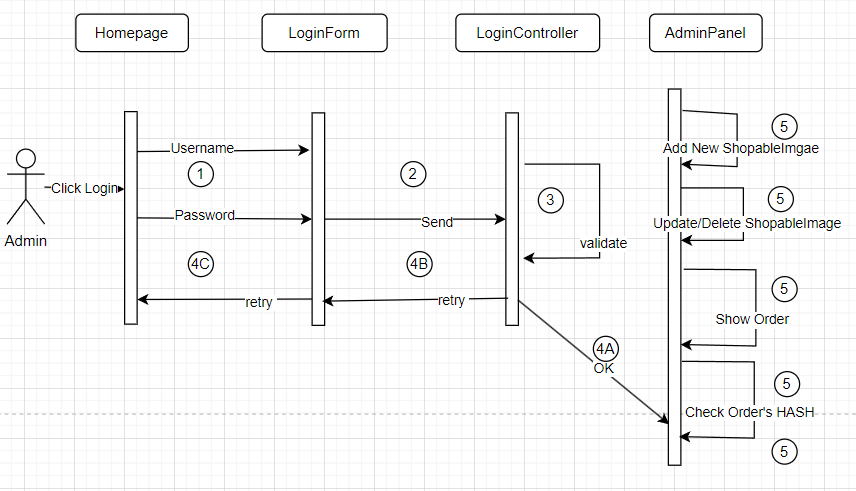
Da qui potrà accedere in visualizzazione ai Work e alle relative immagini contenute in essi, per i quali avrà ottenuto l’autorizzazione della visualizzazione.



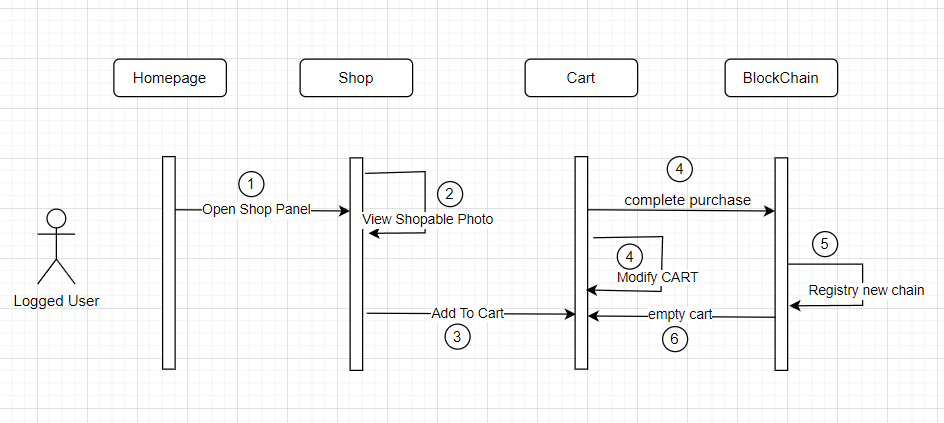
L’admin dal proprio pannello di gestione potrà creare di nuovi Work, avrà la completa gestione di aggiornamento e cancellazione dei Work, potrà caricare immagini specificando a quale Work appartengono. Avrà la completa gestione di aggiornamento e cancellazione delle singole immagini e potrà settare le autorizzazioni di visualizzazione in modo granulare.



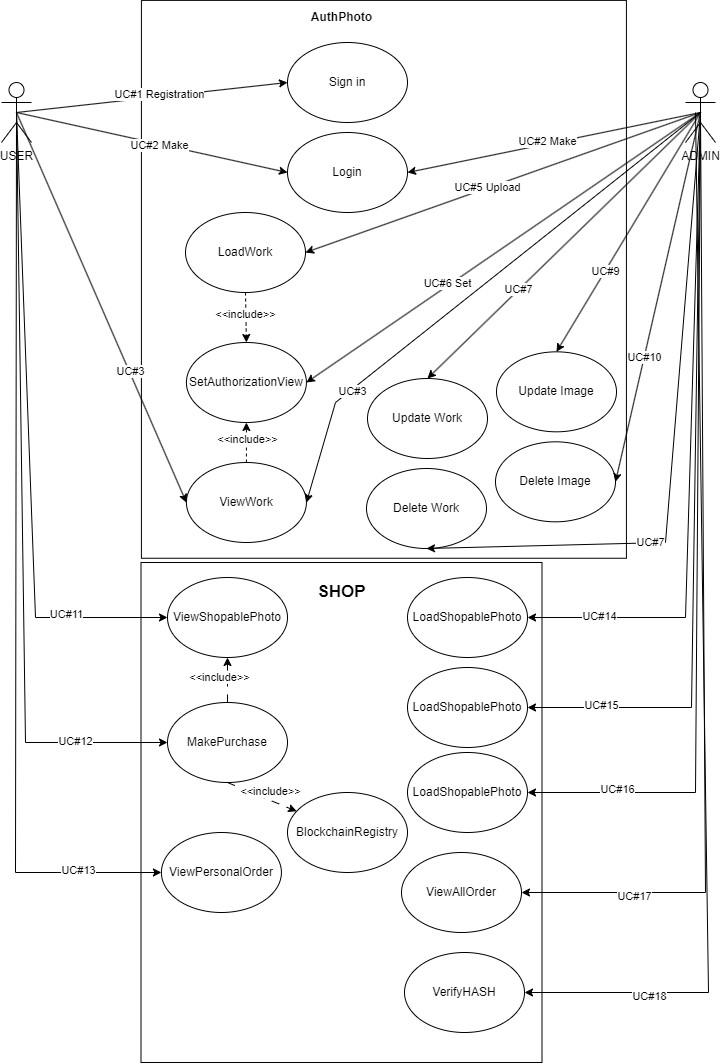
L’admin dalla View AdminShop potrà caricare nuove ShopableImage, cancellare/aggiornare quelle già in vendita, potrà visualizzare tutti gli ordini fatti e verificare la veridicità dell’Hash delle transizioni di vendita.



L'utente potrà visualizzare le ShopableImage con relative caratteristiche e costo. Potrà acquistarle e l'acquisto verrà registrato in una blockchain



## Use Case Diagram



* UC#1 Registration: L'utente si registra al sistema.
* UC#2 Make Login: L'utente effettua il login nel sistema.
* UC#3 LoadWork: L'utente carica un lavoro fotografico.
* UC#4 SetAuthorizationView: L'utente richiede l'autorizzazione per visualizzare un lavoro.
* UC#5 Upload: L'amministratore carica una nuova immagine.
* UC#6 Update Work: L'amministratore aggiorna un lavoro esistente.
* UC#7 Delete Work: L'amministratore elimina un lavoro.
* UC#8 Update Image: L'amministratore aggiorna un'immagine esistente.
* UC#9 Delete Image: L'amministratore elimina un'immagine.
* UC#10 Set: L'amministratore concede l'autorizzazione per visualizzare un lavoro.
* UC#11 ViewWork: L'utente visualizza un lavoro
* UC#12 ViewShopablePhoto: L'utente visualizza le foto disponibili per l'acquisto.
* UC#13 MakePurchase: L'utente effettua un acquisto.
* UC#14 ViewPersonalOrder: L'utente visualizza i propri ordini.
* UC#15 LoadShopablePhoto: L'amministratore carica le foto disponibili per la vendita.
* UC#16 LoadShopablePhoto: L'amministratore carica le foto disponibili per la vendita.
* UC#17 ViewAllOrder: L'amministratore visualizza tutti gli ordini.
* UC#18 VerifyHASH: L'amministratore verifica l'hash di una transazione nella blockchain.

Nella sezione *“Analisi API - tabella di copertura”* verranno analizzati ognuno di questi Use Case, andando vedere quale API o la sequenza di API da chiamare per verificarlo.

# Descrizione dei package

Prima di analizzare nel dettaglio l’implementazione del back-end con il framework Spring, in questa sezione si analizza la divisione logica dei package.

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente

* **configuration:** contiene la classe utilizzata per eseguire il bootstrap e avviare l'applicazione Spring. Inoltre, contiene il Bean CommandLineAppStartupRunner per l’inizializzazione di uno User e di un Admin.
* **controller:** contiene i controller che si occupano dell’esposizione e nella gestione delle risorse attraverso le operazioni HTTP.  Si occupano del mapping delle risorse: gestiscono la conversione dei dati in formato JSON durante la trasmissione delle risposte e l'interpretazione (JSON -> Oggetti del Modello dei dati) dei dati ricevuti nelle richieste
* **dto:** contiene i DTO utilizzati per trasferire dati tra il livello di persistenza dei dati (database) e il livello di esposizione delle API (controller).
* **mapper:** Contiene una classe, che sfruttando i generics, esegue il mapping tra model e DTO e viceversa. Per alcuni parametri che non vengono mappati, la mappatura viene fatta manualmente all’interno dei controller.
* **model:** contiene le classi che modellano che costituiscono il modello di dominio dell'applicazione, la rappresentazione dei concetti chiave e delle entità del dominio del problema.
* **repository:** contiene le classi che sono responsabili della gestione della persistenza dei dati, sviluppate secondo il framework Spring Data JPA Le principali responsabilità e funzionalità di questo livello sono: interfacciarsi con il Database, gestire le operazioni CRUD, mappare gli oggetti Java sulle tabelle del database, gestiscono la conversione dei dati tra i tipi di dati Java e i tipi di dati del database e la gestione della persistenza
* **security:** contiene la Classe WebSecurityConfig che abilita l'autenticazione e l'autorizzazione all’interno dell’applicazione. Inoltre, come configurare il sistema di sicurezza per l'applicazione web. Questa classe estende o implementa le classi di configurazione fornite da Spring Security per personalizzare il comportamento del sistema di sicurezza.
  + **jwt:** metodi per la creazione, verifica e il controllo del token JWT
  + **payload:** DTO per la gestione della login/signin e l’aggiornamento della password di un utente
* **service:**contiene tutte le classi che si trovano tra il layer dei controller e il layer di accesso ai dati (repository/DAO).
* **utility:** Contiene due classi di servizio. ThumbnailGenerator che si occupa della generazione delle miniature e dell’aggiunta dei watemark. La classe BlockchainTransactionService utilizzata per la registrazione di nuove transizioni o della verifica di vecchie transizioni all’interno della blockchain

# Implementazione lato JAVA

L’implementazione del progetto è stata realizzata basandosi sull’architettura MVC basata su JakartaEE. In particolare, si è deciso di utilizzare con il framework Spring per la realizzazione del back-end.

Del “mondo” Spring si è deciso di utilizzare Spring Boot: è un framework open-source per creare applicazioni web basate su Spring, semplificando la configurazione e l'avvio delle applicazioni Spring. La versione utilizzata è la **3.3.2.**

Si è fatto uso del tool <https://start.spring.io> che ci permette di eliminare gran parte della configurazione manuale ed evitare conflitti tra le varie versioni delle librerie esterne.

La community di Spring mette a disposizione anche l’IDE Spring Tool Suite basato sui Eclipse, per questo progetto è stata utilizzata la versione 4.21.0.

Nel dettaglio le tecnologie/librerie utilizzate sono:

* Java OpenJDK 17.0.9
* Maven 3.9.6
* spring-boot-starter-parent 3.24
* spring-boot-starter-data-jdbc
* spring-boot-starter-jpa
* spring-boot-starter-web
* spring-boot-starter-jaxb-runtime (glassfish web server)
* spring-boot-starter-mariadb-java-client
* spring-boot-starter-security
* jjwt-api 0.11.5
* jjwt-impl 0.11.5
* jjwt-jackson 0.11.5
* javax.servlet-api 4.0.1
* Apache Tomcat 10.1.19

Per i test delle API è stato utilizzato PostMan 11.3.2

L’analisi dell’implementazione percorrerà tutti i package, in modo da analizzare l’architettura generale, ma soffermerà l’attenzione i dettagli implementativi ritenuti più interessanti.

## it.myportfolio.configuration.myportfolioApplication.java

Questa, anche se presenta poche righe di codice, è una delle classi principali del progetto. Si occupa di fare il discovery dei vari componenti del boostrap dell’applicazione.  
Nel dettaglio:

* @SpringBootApplication: è un’annotazione “scorciatoia” che combina tre annotazioni molto importanti:
  + @Configuration: indica che la classe è una sorgente di definizioni di bean per il contesto dell'applicazione.
  + @EnableAutoConfiguration: abilita la configurazione automatica di Spring Boot. Tenta di configurare automaticamente i bean necessari per l'applicazione in base alle dipendenze trovate nel classpath.
  + @ComponentScan: abilita la scansione dei componenti, permettendo a Spring di trovare e registrare automaticamente i bean all'interno del pacchetto corrente (e/o dei suoi sottopacchetti).
* @ComponentScan(basePackages = {"it.myportfolio.\*"}): questa annotazione specifica i pacchetti da scansionare per trovare i componenti Spring (come @Component, @Service, @Repository). In questo caso, si specifica al framework Spring di scansionare tutti i sottopacchetti del pacchetto it.myportfolio.
* @EnableJpaRepositories(basePackages = "it.myportfolio.repository"): questa annotazione abilita l’uso dei repository JPA e il relativo framework di querying Data JPA. In questo caso le interfacce repository sono presenti all'interno del pacchetto it.myportfolio.repository.
* @EntityScan("it.myportfolio.model"): questa annotazione specifica i pacchetti da scansionare per trovare le entità JPA. In questo caso le entità JPA sono all'interno del pacchetto it.myportfolio.model.

Nel codice dello Snippet 1 possiamo vedere la funzione main(), che richiama il metodo “padre” di tutta l’applicazione: **SpringApplication.run(MyApplication.class, args)**

La chiamata di quel metodo esegue il bootstrap dell'applicazione Spring Boot. Vediamo nel dettaglio cosa succede:

1. **Crea un'istanza di SpringApplication**:

* SpringApplication è una classe di convenienza che esegue automaticamente il bootstrap di un'applicazione Spring da un metodo main statico.

1. **Configura l'applicazione**:

* Determina l'ambiente di esecuzione, se specificato (ad es. development, production).
* Configura il contesto dell'applicazione (ad es. ApplicationContext).
* Abilita il logging, se richiesto.

1. **Scansione dei componenti**: scansiona i pacchetti per trovare i componenti, le configurazioni e i servizi annotati con @Component, @EnableJpaRepositories, @EntityScan
2. **Auto-configurazione**:

* Configura automaticamente molti aspetti dell'applicazione in base alle dipendenze trovate nel classpath.
* Download e linking automatico di tutto le dipendenze richieste all’interno del pom.xml
* Connessione al database, sfruttando i parametri (credenziali, connettore, dialetto) presenti all’interno del file application.properties

1. **Avvia l'ApplicationContext**: è il container principale di Spring che gestisce i bean e le loro dipendenze.
2. **Avvia il server web integrato**: avvia automaticamente il server web Tomcat incorporato per servire l'applicazione.



Snippet 1: Codice completo classe MyportfolioApplication

All’interno della classe è stato utilizzato un CommandLineAppStartupRunner (classe che implementa l'interfaccia CommandLineRunner) che viene utilizzato per eseguire del codice subito dopo il completamento del bootstrap dell'applicazione, ma prima che l'applicazione inizi ad accettare richieste. In particolare è stato utilizzato per inizializzare un’utenza con ruolo admin, un’utenza con ruolo user e il Work “Shop” che raccoglierà le immagini acquistabili all’interno dello shop.

## Annotations

La prossima sezione descriverà le **entità di dominio** realizzate per modellare il dominio applicativo, ma prima di vederle nel dettaglio, in questa sezione si analizzeranno tutte le annotations utilizzate per definire il comportamento delle entità (modelli), delle tabelle, delle relazioni tra tabelle, e di altri aspetti legati alla gestione della.

* **@Entity:** è utilizzata per indicare che una classe Java rappresenta un'entità persistente, cioè un oggetto che sarà mappato a una tabella nel database relazionale.  
  Una volta annotata una classe con @Entity, ogni istanza di questa classe può essere gestita da JPA/Hibernate come una riga in una tabella del database.
* **@Table(name = "x")** è utilizzata per specificare i dettagli della tabella del database a cui l'entità è mappata. In questo caso è utlizzata per definire il nome della tabella nel database a cui è mappata l'entità.
* **@Table(name = "user", uniqueConstraints = { @UniqueConstraint(columnNames = "username"), @UniqueConstraint(columnNames = "email") }):** oltre a definire il nome della tabella, la @Table consente di specificare dei vincoli unici sulle colonne.  
  In particolare, tramite *uniqueConstraints*, le colonne username e email della tabella user devono essere uniche.
* **@Id:** indica che il campo annotato è la **chiave primaria** dell'entità. Questo campo sarà utilizzato come identificatore univoco per ogni istanza dell'entità.
* **@GeneratedValue(strategy = GenerationType.IDENTITY**): viene utilizzata insieme a @Id per indicare che il valore della chiave primaria deve essere generato automaticamente.  
  In tutti i modelli ad esclusione dei Model ImageProject e ShopableImage è stato utilizzato il parametro *GenerationType.IDENTITY*, grazie al quale la chiave primaria viene generata dal database attraverso una colonna auto-incrementante.
* **@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS):** viene utilizzata per specificare la strategia di ereditarietà tra entità in Hibernate. Ogni classe concreta della gerarchia ereditaria viene mappata su una propria tabella nel database. Ogni tabella avrà tutte le colonne della classe padre e delle classi figlie.
* **@GeneratedValue(strategy = GenerationType.TABLE):** la chiave primaria viene generata usando una tabella specifica nel database che tiene traccia dei valori delle chiavi primarie per tutte le entità.
* **@Column(name = "is\_sold", columnDefinition = "boolean default false"):** specifica che il campo isSold sarà mappato su una colonna chiamata is\_sold nel database e che la colonna sarà di tipo booleano, con false come valore predefinito.
* **@Column(nullable = false):** specifica che la colonna associata non può contenere valori null e garantisce che il campo abbia sempre un valore prima di essere inserito nel database.
* **@OneToMany(cascade = CascadeType.ALL)**indica una relazione uno-a-molti tra due entità. Il campo CascaseType specifica che tutte le operazioni di persistenza (persist, merge, remove, refresh, detach) effettuate sulla entità padre saranno propagate anche alle entità figlie.
* **@JoinColumn(name = "work\_id"):** utilizzata per specificare la colonna di join nel database per una relazione tra due entità. Questa annotazione definisce il nome della colonna che fungerà da chiave esterna nella tabella dell'entità figlia.
* **@ManyToOne:** definisce una relazione molti-a-uno tra due entità; viene applicata sul lato "molti" della relazione, cioè sull'entità che contiene la chiave esterna che punta all'entità "uno".
* **@ManyToMany(fetch = FetchType.LAZY):** definisce una relazione molti-a-molti tra due entità. Tramite il campo *FetchType* si specifica che l'entità correlata deve essere caricata pigramente (solo quando viene effettivamente richiesta). Questo migliora le prestazioni evitando il caricamento di dati non necessari.
* **@JoinTable(name = "visible\_work", joinColumns = @JoinColumn(name = "work\_id"), inverseJoinColumns = @JoinColumn(name = "user\_id**")): usata per definire la tabella di join in una relazione molti-a-molti.
  + **name**: definisce il nome della tabella di join.
  + **joinColumns**: specifica la chiave esterna nella tabella di join che si riferisce all'entità corrente.
  + **inverseJoinColumns**: specifica la chiave esterna nella tabella di join che si riferisce all'entità opposta.
* **@ManyToMany(mappedBy = "shopableImages"):** indica che questa è la parte inversa di una relazione molti-a-molti e che la gestione della relazione è delegata all'altra entità

## it.myportfolio.model

Come visto in precedenza il package model contiene le classi che modellano il dominio applicativo. I model o **entità di dominio (Domain Entities)** rappresentano oggetti del dominio dell'applicazione e sono mappati su tabelle di un database relazionale tramite Hibernate.

### Work

Il modello Work rappresenta un progetto fotografico o un insieme di fotografie che condividono una caratteristica comune, come il soggetto (es. paesaggi) o il contesto (es. commissione per un'azienda). In pratica, il Work è un contenitore che raggruppa le fotografie scattate per un cliente specifico o con un obiettivo tematico comune, come la fotografia paesaggistica, commerciale o di eventi.

In sostanza, il modello Work serve a organizzare e gestire le fotografie in base a progetti o tematiche, facilitando la catalogazione e la presentazione del lavoro svolto in un contesto professionale o artistico.

Le risorse immagini caricate devono poter essere raggruppate per Work ed esistono se e solo se sono collegate ad un Work.

Ogni Work ha come campi:

* **Id:** chiave univoca auto incrementale
* **title:** rappresenta il titolo del lavoro fotografico
* **company:** rappresenta l’azienda/cliente per il quale il fotografo ha lavoro per realizzare lo shooting. Company può essere utilizzato anche per descrivere un raggruppamento di immagini con caratteristiche precise, ma non appartenenti ad un’azienda (es. fotografia paesaggistica)
* **completationDate:** data in cui è stato completato lo shooting
* **image:** lista di tutte le immagini scattate durante il lavoro fotografico
* **user:** lista di utenti che hanno accesso in visualizzazione a quel lavoro. A livello di database la relazione N-N tra i Work e gli User viene risolta con la tabella intermedia “visible\_work”



### ImageProject

Il modello ImageProject è la struttura che rappresenta una singola immagine all'interno di un progetto fotografico. Esso memorizza informazioni fondamentali per l'identificazione e l'accesso a ogni immagine, inclusi un'etichetta descrittiva, l'URL completo dell'immagine e un URL per una versione miniaturizzata.

I campi di ImageProject sono:

* **Id**: chiave univoca auto incrementale
* **Label**: la label è una breve descrizione o nome assegnato all'immagine, che può essere utilizzato per identificare o descrivere brevemente il contenuto dell'immagine stessa.
* **URL**: Il campo URL memorizza l'indirizzo (absolute path) completo dell'immagine salvata sul NAS.
* **Thumbnail URL**: memorizza l'indirizzo (absolute path) completo dell'immagine in formato miniatura salvata sul NAS. Questa miniatura viene utilizzata per una visualizzazione più rapida nella galleria di immagini senza dover caricare l'immagine completa.



### ShopableImage

Il modello ShopableImage è un'estensione (sfruttando l’ereditarietà) del modello ImageProject, specificamente progettato per gestire la vendita di immagini come opere fotografiche uniche. Questo model consente di trattare ogni immagine come un pezzo da collezione, venduto in singola unità.

Una volta venduta, l'immagine non è più disponibile per altri acquirenti, riflettendo il suo carattere esclusivo e da collezione. La relazione con Cart consente ai clienti di aggiungere l'immagine ai loro carrelli per poi eventualmente procedere con l’acquista, ma la natura unica dell'opera impone che una volta acquistata, essa non possa più essere riacquistata.

I campi di questo model sono:

* **Ereditati da ImageProject**: eredita tutte le proprietà di base dal modello ImageProject, come l'ID, l'etichetta (label), l'URL dell'immagine, e l'URL della miniatura (thumbnailURL).
* **isSold**: Questo campo booleano (isSold) indica se l'immagine è già stata venduta. Dato che ShopableImage rappresenta opere fotografiche in numero unico, una volta che il campo isSold è impostato su true, l'immagine non sarà più disponibile per la vendita.
* **price**: price è un campo obbligatorio che rappresenta il prezzo di vendita dell'immagine.
* **cart**: Questa è una relazione ManyToMany con il modello Cart, che rappresenta i carelli, all’interno dello shop, dei clienti in cui l'immagine è stata aggiunta. Tuttavia, poiché l'immagine è unica, una volta venduta, l'immagine non può essere acquista: al momento dell’acquisto viene verificato se le immagini contenute nel carrello sono state già comprate, in caso affermativo restituisce un errore contenente quali immagini non possono essere acquistate.



### User

Il modello User è una rappresentazione completa di un utente all'interno dell'applicazione, che include sia informazioni personali come nome, cognome, username, email e password, sia aspetti legati alla sicurezza e alle autorizzazioni, come lo stato dell'account (enable) e i ruoli (roles). Inoltre, gestisce le relazioni con i progetti che l'utente può visualizzare (visibleWorks), offrendo una visione chiara e organizzata dei dati utente e delle sue interazioni all'interno del sistema.

I campi di questo model sono:

* **Id**: chiave univoca auto incrementale
* **name**, **surname:** i campi che memorizzano rispettivamente il nome e il cognome dell'utente. Questi campi servono per identificare l'utente in modo più umano, oltre che per scopi di visualizzazione.
* **username**: è un campo univoco che identifica l'utente all'interno del sistema. Viene utilizzato per l'autenticazione e deve essere unico per ogni utente, come indicato dal vincolo di unicità (@UniqueConstraint).
* **email**: è un altro campo univoco utilizzato per contattare l'utente. La sua unicità è garantita da un vincolo di unicità (@UniqueConstraint).
* **password**: è il campo che memorizza la password dell'utente in forma crittografata lato DB.
* **enable**: è un campo booleano che indica se l'account dell'utente è attivo (true) o disabilitato (false). Questo campo è utile per gestire lo stato dell'account, ad esempio in caso di sospensione temporanea.
* **VisibleWorks**: è una relazione ManyToMany con il modello Work. Rappresenta i progetti (o insiemi di immagini) che l'utente ha il permesso di visualizzare. Questa relazione è definita come LAZY, il che significa che i Work associati non vengono caricati automaticamente insieme all'utente, ma solo quando richiesto, per migliorare le prestazioni.
* **Roles**: roles è una relazione ManyToMany con il modello Role, che rappresenta i ruoli assegnati all'utente. Questi ruoli definiscono i permessi e le autorizzazioni dell'utente all'interno del sistema, come l'accesso a determinate funzionalità o risorse. La relazione è gestita tramite una tabella di join (user\_roles) che collega gli utenti con i ruoli.



### ERole

ERole è un'enumerazione (enum) che definisce i ruoli disponibili all'interno del sistema.

* **ROLE\_USER**: Rappresenta un ruolo standard per gli utenti dell'applicazione. Gli utenti con questo ruolo hanno accesso alle gallerie per i quali hanno diritto di visualizzazione e allo shop.
* **ROLE\_ADMIN**: Rappresenta un ruolo amministrativo con privilegi elevati. Gli utenti con questo ruolo hanno accesso a tutte le funzionalità dell'applicazione: gestione e amministrazione Work e immagini, la gestione degli utenti.



### Role

Il modello Role rappresenta un'entità nel database che memorizza le informazioni dei due ruoli definiti all'interno dell'applicazione. Questi ruoli sono utilizzati per controllare e gestire le autorizzazioni e i permessi degli utenti.



### Cart

Il modello Cart gestisce i carrelli della spesa per gli utenti, consentendo loro di selezionare e salvare immagini acquistabili che intendono comprare. Ogni carrello è associato a un singolo utente e può contenere più immagini ShopableImage.

I campi di questo model sono:

* **Id**: chiave univoca auto incrementale
* **User**: rappresenta l'utente associato a questo carrello. La relazione è definita come @OneToOne, indicando che ogni carrello è associato a un solo utente e viceversa.
* **ShopableImages**: è una relazione ManyToMany che rappresenta le immagini acquistabili aggiunte al carrello. Utilizza la tabella di join *“cart\_shopable\_image”* per gestire l'associazione tra i carelli degli utenti e le immagini.



### SalesOrder

**Descrizione del modello SalesOrder**

Il modello SalesOrder è essenziale per gestire gli ordini di vendita all'interno del sistema, includendo non solo le informazioni di base sull'acquisto, come la data e l'utente associato, ma anche un'integrazione avanzata con la tecnologia blockchain. Il campo hash, generato tramite uno script che registra l'ordine su un fork di Ethereum, fornisce un ulteriore livello di sicurezza e trasparenza, assicurando che ogni ordine sia verificabile e immutabile, proteggendo così gli interessi sia dell'acquirente che del venditore (vedremo più avanti una visione più dettagliata della registrazione all’interno della blockchain).

I campi di questo model sono:

* **Id**: chiave univoca auto incrementale
* **Timestamp**: campo di tipo Date che memorizza la data e l'ora in cui l'ordine è stato effettuato.
* **PurchasedImage**: relazione OneToMany con il modello ShopableImage. Questo campo rappresenta la lista delle immagini acquistate come parte dell'ordine.
* **User**: rappresenta l'utente che ha effettuato l'ordine. La relazione è definita come ManyToOne, indicando che più ordini possono essere associati a un singolo utente, ma ogni ordine è legato a un solo utente.
* **Hash**: hash è un campo di tipo String che memorizza un valore di hash unico associato all'ordine e generato tramite le chiamate API fornite dalla libreria Infura (vedremo più avanti una visione più dettagliata della registrazione all’interno della blockchain).



## it.myportfolio.dto

Un **DTO (Data Transfer Object)** è un oggetto usato per trasportare dati tra il livello di servizio (in questo caso le API esposte al pubblico) e il livello di persistenza (un database).

I DTO sono usati principalmente per:

1. **Evitare di esporre oggetti di dominio direttamente:** esporre direttamente questi oggetti nelle API può portare a problemi di sicurezza e manutenibilità. I DTO offrono un livello di astrazione che consente di filtrare e controllare quali dati vengono effettivamente esposti all'esterno.
2. **Ridurre il sovraccarico delle comunicazioni:** i DTO consentono di inviare solo i dati necessari, riducendo la quantità di informazioni che viaggiano sulla rete.
3. **Separazione dei concetti:** I DTO separano la logica di business dalle informazioni di presentazione, migliorando la manutenibilità del codice. Le entità del dominio possono contenere logica di business e associazioni complesse, mentre i DTO contengono solo dati necessari per una specifica operazione o visualizzazione.
4. **Sicurezza dei dati:** esporre le entità del dominio direttamente può essere pericoloso. Ad esempio, un'entità potrebbe contenere campi sensibili come password, ID interni, o altre informazioni che non dovrebbero essere inviate al client.
5. **Validazione dei dati:** Quando un DTO viene utilizzato per ricevere dati da un client (come in una richiesta HTTP POST o PUT), è possibile applicare le annotazioni di validazione di Spring (@NotNull, @Size, ecc.) direttamente sui campi del DTO. Questo permette di convalidare i dati prima di convertirli in entità di dominio, garantendo che i dati immessi siano conformi alle regole di business.
6. **Aggiornamento delle:** iDTO possono aiutare a gestire diverse versioni dell'API. Ad esempio, se una nuova versione di un'API richiede nuovi campi o una struttura di dati diversa, è possibile creare nuovi DTO specifici per la nuova versione, mantenendo la compatibilità con le versioni precedenti.

Nella sezione precedente sono stati descritti nel dettaglio tutti i model presenti nel progetto, questa parte si concentrerà sugli aspetti più rilevanti e sui punti di maggiore interesse relativi ai DTO, si metteranno in evidenza solo le caratteristiche e le implementazioni che meritano particolare attenzione.

### UserPersonalDetailsDTO vs User

In questa sezione di mette a confronto il Model User ed il DTO UserPersonalDetailsDTO per mettere in risalto le caratteristiche di entrambi e capirne le differenze e i differenti usi.

**1. Differenze di Scopo e Ruolo**

* **Model (User)**: È la rappresentazione diretta della tua entità nel database. Include tutti i campi che corrispondono alle colonne della tabella del database, contiene anche le relazioni con altre entità (ad esempio, roles e visibleWorks). Questo oggetto viene utilizzato nelle operazioni di persistenza e recupero dei dati.
* **DTO (UserPersonalDetailsDTO)**: ll DTO contiene solo i campi necessari per un particolare scopo o contesto, riducendo così la quantità di dati trasferiti e migliorando l'efficienza. L’esempio presentato non riporta i campi come username, enable, password e visibleWorks, includendo solo i dati rilevanti per l'uso specifico (come name, surname, email, role). Questo DTO ad esempio è utilizzato come ritorno dell’API GET "/api/user" per restituire l’anagrafica dell’utente per un eventuale aggiornamento dei campi e permette di nascondere dettagli sensibili o irrilevanti del modello, come la password o il campo username. Questo riduce il rischio di esposizione accidentale di informazioni sensibili.



### SignupRequest

Un altro esempio di utilizzo di un DTO è durante il processo di registrazione di un utente.  
Il DTO SignupRequest è progettato per gestire i dati necessari durante il processo di registrazione di un nuovo utente nell'applicazione. Questo oggetto è utilizzato per raccogliere e trasferire in modo sicuro le informazioni dal client (frontend) al backend, dove verrà gestita la creazione dell'utente e la persistenza sul database.

Il DTO permette di limitare l'esposizione dei campi sensibili e di controllare quali dati vengono accettati dall'esterno. Ad esempio, nel processo di registrazione, il DTO accetta solo i campi strettamente necessari, evitando che informazioni non desiderate o potenzialmente pericolose vengano iniettate.



### DetailsSalesOrderDTO

Vediamo adesso un altro uso dei DTO, ovvero quello di dare in output valori che non sono presenti all’interno del DB, ma vengono calcolati/generati a runtime al momento della richiesta.

Il DetailsSalesOrderDTO è un DTO che viene utilizzato per fornire agli utenti Admin una rappresentazione dettagliata di un ordine di vendita. Questo DTO viene utilizzato per trasferire al client tutte le informazioni rilevanti su un ordine di vendita, rendendo disponibili non solo i dati presenti nel modello SalesOrder, ma anche alcune informazioni calcolate dinamicamente.

Un aspetto significativo del DetailsSalesOrderDTO è che i campi totalPrice e piece non sono memorizzati direttamente nel modello SalesOrder. Invece, questi valori vengono calcolati dinamicamente ogni volta che il DTO viene richiesto.

Vediamo i campi più interessanti:

* **purchaseImage**: Una lista di oggetti ShopableImageDTO che rappresentano le immagini associate agli articoli acquistati.
* **totalPrice**: Questo campo rappresenta il prezzo totale dell'ordine, ossia la somma dei prezzi di tutti gli articoli acquistati. È importante notare che totalPrice non è un campo presente nativamente nel modello SalesOrder, ma viene calcolato a runtime al momento della richiesta.
* **piece**: Indica il numero totale di pezzi acquistati nell'ordine. Anche questo campo non è presente nel modello SalesOrder e viene calcolato a runtime.

Grazie a questo oggetto si evitano di memorizzare dati calcolati e superflui che possono essere derivati nel database, riducendo la ridondanza e semplifica la struttura del modello.  
I dati vengono generati solo quando effettivamente necessari, riducendo così il rischio di inconsistenze.



### SimpleShopableImageDTO

Per ultimo vediamo l’esempio di semplice, ma intuitivo dei DTO. Ovvero la semplificazione del modello di Dominio, in un oggetto più semplice da utilizzare per passare informazioni nelle due direzioni backend -> frontend e frontend-> backend.



## it.myportfolio.mapper

All’interno di questo package è presente la classe Mapper, che sfruttando i generics, esegue il mapping tra model e DTO e viceversa. Per alcuni parametri che non vengono mappati, la mappatura viene fatta manualmente all’interno dei controller.

Il metodo *toEntity* utilizza BeanUtils (libreria Spring Framework) e la riflessione per copiare le proprietà con lo stesso nome e tipo dal DTO all'oggetto Entity. Praticamente, se DTO e Entity hanno campi comuni (ad esempio id, Url ecc), i valori verranno copiati dall'oggetto DTO all'oggetto entity.

Nel dettaglio quello che viene fatto:

* Tramire **entityClass.getConstructor()** viene recuperato il costruttore predefinito (senza parametri) della classe entityClass.
* Il metodo **constructor.newInstance()** crea una nuova istanza di Entity utilizzando il costruttore recuperato.
* Tramite il metodo statico copyProperties della classe BeanUtils si copiano i valori delle proprietà del bean sorgente specificato nel bean di destinazione.



Snippet 2: metodo per fare il mapping DTO -> Entity

Il metodo *toDTO* è il complemento del metodo *toEntity*. Serve a convertire un oggetto di tipo Entity (un'entità del dominio) in un oggetto di tipo DTO (Data Transfer Object).



Snippet 3: metodo per fare il mapping Entity -> DTO

## it.myportfolio.service

Dopo aver visto i modelli di dominio, gli elementi che ci permettono di passare informazioni tra client e server e gli oggetti che si occupano di fare il mapping, vediamo adesso il layer Service.

Il Service layer è uno dei layer architettonici fondamentali, principalmente responsabile dell'implementazione della Business Logic di un'applicazione. Si trova tra il controller e il data access layer (repository o DAO).

Grazie all’aggiunta di questo layer otteniamo i seguenti vantaggi:

* **Inversion of Control (IoC)**: Spring gestisce l'istanziazione e l'iniezione dei service (tramite Dependency Injection). Non dobbiamo creare manualmente gli oggetti dei service, ma possono essere iniettati tramite l’annotazione come @Autowired.
* **Riutilizzabilità**: essendo separati dal livello di presentazione e accesso ai dati, possono essere facilmente riutilizzati in altre parti dell'applicazione o in altri progetti.
* **Loose Coupling**: utilizzando interfacce per classi di servizio e iniettandole tramite l'iniezione di dipendenza di Spring, si promuove un loose coupling all’interno dell’applicazione, che la rende più manutenibile e testabile.
* **Separation of concerns**: avere un livello di servizio separato assicura una chiara separazione delle preoccupazioni. Il livello controller/web si concentra sulla gestione delle richieste e delle risposte HTTP, mentre il livello di servizio si concentra sulla Business Logic. Questa separazione rende la base di codice organizzata e più facile da gestire.

**I**l livello di servizio collabora con il livello di accesso ai dati (repository o DAO), che vedremo nella prossima sezione, per recuperare, manipolare e archiviare i dati.

Annotando una classe con @Service, la si rende un bean gestito dal container di Spring. Questo significa che Spring si occuperà di creare e gestire l'istanza di questa classe, oltre a gestire le sue dipendenze tramite dependency injection. Grazie a questa annotazione è possibile iniettare il service in altre parti dell'applicazione, come controller o altri service, senza dover creare manualmente istanze della classe.

A titolo di esempio analizziamo il *WorkService* ed il metodo *addToCart* del *CartService.*

Il service WorkService presentato implementa una serie di operazioni CRUD (Create, Read, Update, Delete) per gestire l'entità Work. Questa classe utilizza il repository WorkRepository per interagire con il database e contiene la logica necessaria per eseguire operazioni di base sui dati.

**Create (C) - Aggiunta di un nuovo elemento**: l'operazione di creazione viene gestita dal metodo addWork. Questo metodo riceve un oggetto Work, lo salva nel database tramite *workRepository.save(work)* e restituisce l'oggetto salvato, che rappresenta l'elemento appena creato con tutte le informazioni aggiornate (incluso, l'ID generato automaticamente).

**Read (R) - Lettura di uno o più elementi**: Ci sono vari metodi che implementano la funzionalità di lettura:

* **getWorkById(Long id)**: Restituisce un oggetto Work avvolto in un Optional basato sull'ID fornito, se esiste.
* **getAllWork()**: Restituisce una lista di tutti gli oggetti Work presenti nel database.
* **getWorkDTOByIdAndUser(Long workId, Long userId)**: Questo metodo è una lettura condizionale che restituisce un oggetto Work specifico in base all'ID del lavoro e all'ID dell'utente, utilizzando un metodo personalizzato definito nel repository.

**Update (U) - Aggiornamento di un elemento esistente**: Il metodo updateWork gestisce l'aggiornamento di un oggetto Work esistente. Prende come input l'ID dell'oggetto da aggiornare e un oggetto Work contenente i nuovi dati. Se l'elemento esiste, ne aggiorna i campi (titolo, azienda, data di completamento) e lo salva nuovamente nel database.

**Delete (D) - Eliminazione di un elemento**: Il metodo deleteWorkById gestisce l'eliminazione di un oggetto Work dal database. Riceve come parametro l'ID dell'oggetto da eliminare e utilizza il repository per rimuovere l'elemento corrispondente.



Nell’esempio precedente abbiamo visto l’uso di metodi tutti esposti dai DAO senza l’uso di codice ausiliario. Vediamo adesso il metodo *addToCart* Questo metodo permette di aggiungere un oggetto ShopableImage al carrello di un utente.

1. **Recupero del carrello dell'utente**:
   * Il metodo cerca il carrello associato all'utente (user) utilizzando il metodo cartRepository.getCartByUser(user).
   * Se il carrello non esiste, viene creato un nuovo carrello per l'utente, associato e salvato nel database.
2. **Recupero dell'oggetto ShopableImage**:
   * Viene recuperato l'oggetto ShopableImage corrispondente all'ID fornito (shopableImageId) dal repository shopableImageRepository.
3. **Aggiornamento del carrello**:
   * Viene nuovamente recuperato il carrello dell'utente e si verifica se esiste già una lista di immagini (List<ShopableImage> images) associata al carrello.
   * Se la lista è null, viene creata una nuova lista, l'immagine viene aggiunta alla lista, e la lista viene associata al carrello.
   * Se la lista esiste, l'immagine viene semplicemente aggiunta alla lista e il carrello viene aggiornato.
4. **Aggiornamento del lato ShopableImage**:
   * Viene gestita la relazione bidirezionale tra Cart e ShopableImage.
   * Si controlla se l'immagine ha già una lista di carrelli associati (List<Cart> SHCarts). Se la lista è null, ne viene creata una nuova e il carrello corrente viene aggiunto. Se la lista esiste, il carrello viene semplicemente aggiunto alla lista.
5. **Salvataggio finale**:
   * L'immagine e il carrello aggiornati vengono salvati nel database tramite i rispettivi repository, mantenendo così la consistenza delle relazioni tra le entità.
   * Il metodo restituisce l'oggetto Cart aggiornato.



## it.myportfolio.repository

In questa sezione vedremo i repository, che sono componenti fondamentali utilizzati per interagire con il database. Si basano sul concetto di **DAO (Data Access Object)**, che fornisce un'interfaccia per la persistenza dei dati e per le operazioni di accesso ai dati (come CRUD: Create, Read, Update, Delete).

### Spring Data JPA

L’ambiente Sping mette a disposizione il modulo Spring Data che permette di semplificare lo stato di persistenza rimuovendo completamente l’implementazione dei DAO dall’applicazione e per semplificare l'interazione con i database utilizzando Java Persistence API (JPA). Per fare ciò, l’interfaccia DAO deve estendere JpaRepository e Spring Data creerà automaticamente un’implementazione dotata dei metodi CRUD più rilevanti per l’accesso ai dati.

L'obiettivo principale di Spring Data JPA è ridurre la quantità di codice boilerplate necessario per interagire con il database. Spring Boot consente di definire repository utilizzando interfacce, senza dover implementare manualmente i metodi di accesso ai dati, detti CRUD (Create, Read, Update, Delete) e query personalizzate.

Fornisce strumenti per:

* Creare facilmente repository di dati per gestire entità JPA.
* Effettuare operazioni CRUD e di query in modo semplice.
* Automatizzare l'implementazione di metodi di accesso ai dati tramite l'uso di interfacce e naming convention.
* Supportare query personalizzate sia con JPQL (Java Persistence Query Language) che con SQL nativo.

La configurazione di Spring Data JPA è molto semplice, attraverso il file *application.properties* è possibile configurare le proprietà per la connessione al database:



### JpaRepository

La classe *org.springframework.data.jpa.repository.JpaRepository* è una delle interfaccie di Spring Data JPA che fornisce un'implementazione generica del pattern **Repository**. Questa interfaccia estende altre interfacce più semplici come CrudRepository e PagingAndSortingRepository. Grazie a questa interfaccia, è possibile effettuare operazioni di persistenza senza scrivere codice SQL o implementare manualmente le query.

JpaRepository è un'interfaccia generica con due parametri:

****

Dove:

* **T**: Rappresenta il tipo dell'entità (model) che vogliamo gestire.
* **ID**: Rappresenta il tipo dell'identificatore primario dell'entità (ad esempio, Long).

Questa interfaccia dispone di alcuni metodi CRUD più importanti disponibili (che vedremo nella prossima sezione), ma offre anche la possibilità di scrivere query in linguaggio JPQL e l’uso del modulo Query Method: consente di definire metodi di query nel repository semplicemente dichiarando il nome del metodo secondo una convenzione specifica, senza dover scrivere implementazione.

**TIPS:** Per impostazione predefinita, i metodi ereditati da CrudRepository ereditano la configurazione transazionale da SimpleJpaRepository, in particolare:

* I **metodi di lettura** (come findById, findAll, etc.) sono considerati read-only. Spring aggiunge automaticamente l'annotazione @Transactional(readOnly = true) a questi metodi.
* **I metodi di scrittura** (come save, delete, deleteById, etc.) sono transazionali: le operazioni di scrittura vengono eseguite all'interno di una transazione. Se qualcosa va storto durante l'esecuzione del metodo, la transazione viene annullata (rollback) e nessuna modifica viene applicata al database.

### Metodi CRUD

**Spring Data JPA** fornisce già una serie di metodi CRUD (Create, Read, Update, Delete) che coprono la maggior parte delle necessità comuni di manipolazione attraverso le sue interfacce, come **CrudRepository**, **PagingAndSortingRepository**, e **JpaRepository**. Questi metodi semplificano le operazioni di base per la gestione delle entità nel database, senza la necessità di scrivere codice aggiuntivo.

Esempi di metodi disponibili sono:

* deleteById
* existsById
* findById
* save
* findAll
* count
* exists

### Query Methods

Grazie a questa funzionalità è possibile definire metodi che eseguiranno query SQL sul DB semplicemente dichiarando il nome del metodo secondo una convenzione specifica. Spring Data JPA genera automaticamente una query basata sui campi dell'entità per eseguire operazioni di ricerca senza la necessità di scrivere query SQL o JPQL manualmente.

Spring Data JPA supporta molte keyword, le più comuni sono:

* **findBy**: Seleziona entità in base a una condizione.
* **countBy**: Conta il numero di entità che soddisfano una condizione.
* **deleteBy**: Elimina entità che soddisfano una condizione.
* **existsBy**: Verifica se esistono entità che soddisfano una condizione.

Puoi anche combinare queste keyword con operatori logici e confronti:

* **And**, **Or**: Operatori logici per concatenare condizioni.
* **GreaterThan**, **LessThan**: Confronti numerici.
* **Between**: Filtra un campo tra due valori.
* **Like**: Cerca valori che corrispondono a un pattern.
* **In**: Cerca valori all'interno di una lista.
* **OrderBy**: Ordina i risultati per uno o più campi.

I Query Methods sono ideali per query semplici e comuni, in alternativa è possibile utilizzare l'annotazione @Query per scrivere query JPQL o SQL personalizzate.

### ImageRepository

Questo il più semplice repository del progetto. Come possiamo notare non sono stati dichiarati ulteriori metodi, in quanto quelli già inclusi erano sufficienti per il model ImageProject.



### WorkRepository

In questo repository possiamo notare un esempio di query scritta in JPQL per il recupero di un Work dato un Id che hanno un legame di join con un ID di un User, anch’esso passato come parametro. In altre parole stiamo verificando che la richiesta di visualizzazione di un Work da parte di un utente sia legittima.



### UserRepository

In questo repository possiamo vedere un metodo creato tramite annotazione @Query per il recupero di tutti i Work visualizzabili da un certo User.

Sotto possiamo vedere 4 Query method per:

* Recupero di tutti i campi di un User fornito lo username
* Verifica della presenza di un User con un certo username passato come parametro
* Verifica della presenza di un User con una certa mail passata come parametro
* Recupero di tutti i campi di un User passando il suo username e verificando nello stesso tempo che quell’utenza abbia il campo Enable a true.



## it.myportfolio.controller

Ultimiamo la sezione dei vari componenti implementati in questo progetto, con i componenti che si occupano dell’interazione Client-Server: i controller.

Il ruolo del Controller è quello di gestire le richieste HTTP (GET, POST, PUT, DELETE, PATCH), elaborarle e restituire una risposta appropriata.

In Spring Boot per “eleggere” una classe java a Controller (con architettura API RESTful) basta annotarla con @RestController: un'annotazione specializzata di @Controller che combina anche l'annotazione @ResponseBody, in modo da far restituire direttamente i dati nel corpo della risposta (solitamente in formato JSON o XML), invece di restituire una vista.



Vediamo tutte le varie annotazioni e i comportamenti che possiamo far assumere ad un controller

* *@CrossOrigin*: utilizzata per abilitare il CORS (Cross-Origin Resource Sharing) per tutti i metodi (API) di un controller. CORS è un meccanismo che consente alle risorse di essere richieste da un dominio diverso rispetto a quello in cui è ospitato il server. Senza configurare CORS, le richieste HTTP da origini diverse vengono bloccate per motivi di sicurezza.
* *@RequestMapping:* viene utilizzata per mappare le richieste HTTP a metodi del controller. Essa fornisce un modo flessibile e potente per definire come le richieste vengono instradate all'interno dell'applicazione web. In questo caso l’annotazione è a livello di classe che permette di mappare tutte le richieste che corrispondono a un certo pattern di URL a un determinato controller.

Spring offre diverse annotazioni per mappare le richieste HTTP ai metodi del Controller:

* *@GetMapping*: gestisce le richieste HTTP GET.
* *@PostMapping*: gestisce le richieste HTTP POST.
* *@PutMapping*: gestisce le richieste HTTP PUT.
* *@DeleteMapping*: gestisce le richieste HTTP DELETE.
* *@PatchMapping*: gestisce le richieste HTTP PATCH.

È possibile passare parametri o dati alla richiesta in vari modi:

* *@RequestParam*: per i parametri di query (?param=value).
* *@PathVariable*: per i parametri di percorso (/entity/{id}).
* *@RequestBody*: per il corpo della richiesta (in genere utilizzato per oggetti JSON).

Per quando riguarda la risposta, oltre a restituire semplici stringhe (message), un metodo del Controller può restituire oggetti più complessi, avvolti (wrapperd) tramite un oggetto *ResponseEntity* che consente di specificare lo stato http, gli headers di risposta e chiaramente il body della risposta.

## Gestione Copyright delle immagini

Due richieste fondamentali sono:

* la possibilità di far visualizzare immagini protette da watermark (copyright) ad utenti registrati ed ai quali erano state fornite le autorizzazioni di visualizzazione.
* Il sistema deve garantire l’applicazione di un watermark on demand dal lato back-end, nel momento in cui viene richiesta la visualizzazione di un’immagine a risoluzione standard

In questa sezione approfondiremo la questione della protezione delle immagini con copyright, nella sezione successiva approfondiremo la questione della protezione delle API (e quindi della visualizzazione delle immagini) a livello di autenticazione e autorizzazione.

La gestione del copyright delle immagini all'interno dell'applicazione è stata progettata con l'obiettivo di prevenire l'uso non autorizzato delle risorse fotografiche e garantire la protezione dei diritti d'autore. Per fare ciò, sono state adottate diverse misure tecniche in fase di caricamento, elaborazione e visualizzazione delle immagini.

Ricordiamo, inoltre, che per un caricamento più rapido delle risorse si è deciso di sfruttare le thumbnail (immagini ridotte al 30% rispetto all’originale) che verranno utilizzate per visualizzare le griglie delle gallerie, mentre le immagini a dimensioni standard verranno richieste al server solo al momento della richiesta da parte del client.

L'amministratore del sistema ha la possibilità di caricare immagini all'interno del sistema, salvando nel database l'URL locale che punta alla posizione del NAS dove sono salvate le immagini stesse.

Per proteggere le immagini in formato ridotto, alla richiesta dell’amministratore, il sistema genera automaticamente una versione ridotta dell'immagine (thumbnail). Questo processo di ridimensionamento riduce la dimensione dell'immagine al 30% rispetto all'originale, al fine di ridurre il consumo di banda e spazio di archiviazione.

Durante questa fase, viene anche applicato un watermark visibile all'interno delle thumbnail. Questo watermark ha la funzione di identificare l'immagine come protetta da copyright, disincentivando eventuali tentativi di utilizzo illecito. Le thumbnail così generate vengono salvate in locale sul server e sono pronte per essere inviate al client in formato MediaType png ogni volta che viene richiesta una galleria fotografica.

A differenza delle thumbnail, le immagini a dimensione standard non vengono modificate in modo permanente con l'applicazione di un watermark. Questo approccio permette di mantenere intatta la qualità e l'integrità dell'immagine originale, che potrebbe essere necessaria per usi legittimi e autorizzati.

Quando un utente richiede la visualizzazione di un'immagine nella sua dimensione reale, il watermark viene applicato dinamicamente a Runtime. Questo significa che l'immagine originale non viene mai sovrascritta con il watermark, ma il sistema genera una copia temporanea con il watermark applicato, che viene poi inviata al client in formato MediaType png.

Questo meccanismo offre una doppia protezione:

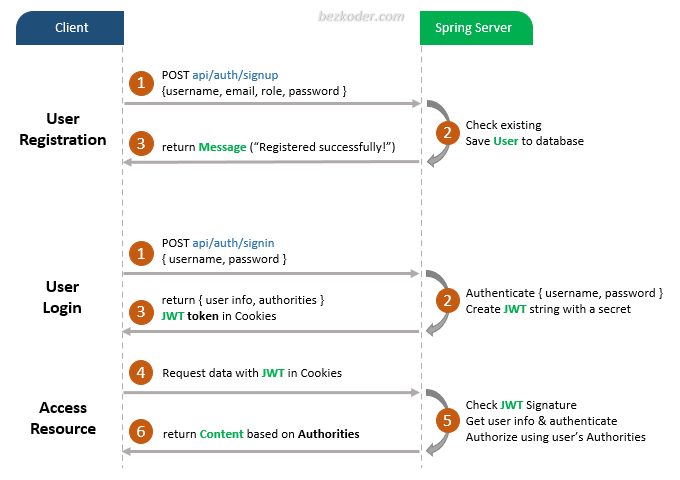
* **Protezione dei diritti d'autore**: Il watermark, applicato su ogni immagine visualizzata, scoraggia l'uso non autorizzato delle immagini. Anche se l'utente tenta di scaricare l'immagine dal sito, questa sarà protetta dal watermark.
* **Conservazione dell'originale**: L'immagine originale rimane invariata e disponibile per eventuali scopi legittimi che richiedano la versione senza watermark.

Queste misure complessive garantiscono un alto livello di protezione delle immagini e dei diritti d'autore all'interno del sistema, rispettando al contempo le esigenze di usabilità e accessibilità delle risorse visive.

## API security: gestione autenticazione e autorizzazione nelle API

In questa sezione si analizzerà la struttura delle API relative a Login e Registrazione con token JWT. Questo passo ci permetterà anche di capire come sono state protette a livello di autenticazione e autorizzazione le altre API.

Prima di entrare nei dettagli dei vari componenti che configurare vediamo come funziona il flusso della procedura di registrazione di un utente e quello del login



**User Registration:**

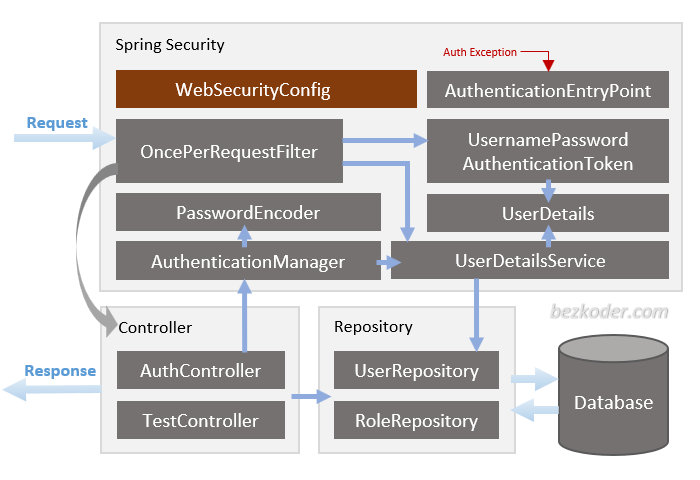
1. Dal client l’utente invia i propri dati (nome, cognome, username, e-mail, password)
2. Lato server viene verificata la presenza di un User già registrato sul database con i dati ricevuti, altrimenti viene persistito il nuovo User sul database e viene generato un nuovo token JWT
3. Se tutti i controlli hanno dato esito positivo, il client riceverà uno status code 200 con i dati dell’utente, il suo ruolo e il token JWT, che dovrà essere conservato e utilizzare per le richieste successive

**User Login:**

1. Dal client l’utente invia i propri dati per effettuare il login (e-mail, password)
2. Lato server viene verificata la correttezza dei dati ricevuti rispetto a quelli presenti sul DB
3. Se tutti i controlli hanno dato esito positivo, il client riceverà uno status code 200 con i dati dell’utente, il suo ruolo e il token JWT, che dovrà essere conservato e utilizzare per le richieste successive

**Access Resource:**

1. Il client effettuerà richieste API, inviando eventualmente il proprio token JWT nell’header della richiesta
2. Lato server ogni richiesta viene verificata: si controlla se quella risorsa è accessibile da tutti o dai solo soggetti in possesso di token JWT valido e si verifica che l’utente che ha fatto la richiesta abbia il ruolo per per poterla fare
3. Se tutti i controlli hanno dato esito positivo, il client riceverà uno status code 200 con i dati. Altrimenti riceverà uno status code 400 o 403.



Nell’immagine qui sopra possiamo vedere ad alto livello tutti i componenti necessari per proteggere (a livello di autenticazione e autorizzazione) le risorse API:

* **WebSecurityConfig**: è il componente principale di tutta l’architettura di sicurezza. Configura cors, csrf, gestione delle sessioni, regole per le risorse protette, garantisce e performa authentication e authorization.
* [**UserDetailsService**](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#tech-userdetailsservice)**:** l'interfaccia ha un metodo per caricare l'utente in base al *nome utente* e restituisce un UserDetail che Spring Security può utilizzare per l'autenticazione e la convalida.
* **UserDetails**: contiene le informazioni necessarie (ad esempio: nome utente, password, role) per creare un oggetto di autenticazione.
* [**UsernamePasswordAuthenticationToken**](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/UsernamePasswordAuthenticationToken.html)**:** è una classe utilizzata per rappresentare un token di autenticazione basato su nome utente e password. Questo token viene comunemente usato durante il processo di autenticazione di un utente. Il token viene poi passato al gestore di autenticazione (AuthenticationManager) per verificare la validità delle credenziali.
* [**AuthenticationManager**](https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#core-services-authentication-manager)**:** è un'interfaccia che rappresenta il componente centrale responsabile della gestione del processo di autenticazione. È utilizzato per verificare le credenziali degli utenti e determinare se un utente è autenticato con successo o meno.
* [**OncePerRequestFilter**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/filter/OncePerRequestFilter.html)**:** è progettata per essere utilizzata per implementare filtri che devono essere eseguiti una sola volta per richiesta API. Viene utilizzato per analizzare i token JWT (JSON Web Token) per autenticare le richieste.
* [**AuthenticationEntryPoint**](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/web/AuthenticationEntryPoint.html)**:** Quando un utente tenta di accedere a una risorsa protetta senza essere autenticato, l'AuthenticationEntryPoint viene invocato per gestire questo scenario. In generale, il suo compito è indirizzare l'utente verso una pagina di login o restituire una risposta HTTP adeguata (ad esempio, un errore 401 Unauthorized).

Vediamo adesso più nel dettaglio (con approccio bottom-up, dal modulo di configurazione all’endpoint che accoglie le richieste) tutte le singole classi realizzate per l’autenticazione degli utenti e della gestione dell’autorizzazione delle risorse:

* it.myportfolio.security -> WebSecurityConfig
* it.myportfolio.service -> UserDetailsServiceImpl implements UserDetailsService
* it.myportfolio.dto -> UserDetailsImpl implements UserDetails
* it.myportfolio.security.jwt -> AuthEntryPointJwt implements AuthenticationEntryPoint
* it.myportfolio.security.jwt -> AuthTokenFilter extends OncePerRequestFilter
* it.myportfolio.security.jwt -> JwtUtils
* it.myportfolio.controller **->** AuthController
* it.myportfolio.repository -> UserRepository extends JpaRepository<User, Long> (già analizzato in precedenza)
* it.myportfolio.repository -> RoleRepository extends JpaRepository<Role, Long> (già analizzato in precedenza)
* it.myportfolio.model -> User (già analizzato in precedenza)
* it.myportfolio.model -> Role: id, name (già analizzato in precedenza)
* it.myportfolio.security.payload  contiene le classi DTO per interfacciarsi con i client nelle operazioni di richiesta e risposta al login/registrazione

Per lavorare con il modulo Spring Security è necessario aggiungere le seguenti dipendenze del pom.xml:



### WebSecurityConfig

L'obiettivo principale di questa classe è definire la sicurezza a livello di applicazione, configurando il flusso di autenticazione e autorizzazione.

Vediamo in dettaglio le parti principali di questa classe:

**Annotazioni**

* **@Configuration**: Indica che questa classe è una classe di configurazione Spring. Consente di definire dei bean che saranno gestiti dal container di Spring.
* **@EnableMethodSecurity**: Abilita la sicurezza basata sui metodi. Questa annotazione consente di applicare regole di sicurezza a livello di metodo utilizzando annotazioni come @PreAuthorize.

**Dipendenze Autowired**

* **@Autowired UserDetailsServiceImpl**: viene utilizzata per caricare i dettagli di un utente durante l'autenticazione, tipicamente recuperandoli da un database.
* **@Autowired private AuthEntryPointJwt**: componente AuthenticationEntryPoint personalizzato, che gestisce le richieste non autorizzate. Quando una richiesta non autenticata tenta di accedere a una risorsa protetta, unauthorizedHandler gestirà la risposta (restituendo un errore 401 Unauthorized).

**Bean**

SecurityFilterChain filterChain(HttpSecurity http): questo metodo configura la catena di filtri di sicurezza (SecurityFilterChain) utilizzando l'API di HttpSecurity. Questo metodo viene chiamato per analizzare tutte le richieste che arrivano al web server. Vediamo questo metodo passo-passo:

* **Disabilitazione CSRF**: http.csrf(csrf -> csrf.disable())
  + CSRF (Cross-Site Request Forgery) viene disabilitato. È comune in applicazioni RESTful, dove l'autenticazione si basa su token (come JWT) anziché su sessioni di login.
* **Gestione delle Eccezioni**: exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))
  + viene configurato il gestore delle eccezioni, utilizzando unauthorizedHandler come AuthenticationEntryPoint. Questo componente sarà invocato quando una richiesta non autenticata tenta di accedere a una risorsa protetta.
* **Session Management**: sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
  + Viene configurata la gestione delle sessioni come **stateless**, il che significa che l'applicazione non manterrà lo stato delle sessioni tra le richieste. Questa configurazione è comune nelle API RESTful che utilizzano JWT o altri tipi di token per autenticare le richieste.
* **Autorizzazione delle Richieste**: authorizeHttpRequests(auth -> auth...
  + Vengono definite le rotte (/api/auth/\*\*, /api/user/\*\*, ecc.) pubbliche.
* **Configurazione del Provider di Autenticazione**: http.authenticationProvider(authenticationProvider())
  + Viene registrato il provider di autenticazione configurato in precedenza (DaoAuthenticationProvider) come provider di autenticazione da utilizzare.
* **Aggiunta del Filtro JWT**: http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class)
  + Viene aggiunto il filtro JWT alla catena dei filtri di sicurezza, prima del filtro UsernamePasswordAuthenticationFilter. Questo significa che il filtro JWT verrà eseguito prima della gestione dell'autenticazione basata su nome utente e password.



### UserDetailsServiceImpl

Layer service che permette di recuperare le informazioni di un utente passato in input un username. Viene controllato anche con l’utente sia abilitato, altrimenti solleva un’eccezione.



### UserDetailsImpl

La classe UserDetailsImpl implementa l'interfaccia UserDetails di Spring Security, che rappresenta i dettagli di un utente che verranno utilizzati per l'autenticazione e l'autorizzazione. Questa classe viene spesso utilizzata per personalizzare il modo in cui Spring Security gestisce i dati degli utenti durante il processo di autenticazione.

Dettaglio da notare è **Collection<? extends GrantedAuthority> authorities**: una collezione di autorizzazioni (GrantedAuthority) che rappresentano i ruoli o le autorizzazioni assegnati all'utente. Queste sono utilizzate per gestire l'autorizzazione all'interno dell'applicazione.

Implementando UserDetails, questa classe è obbligata a fornire delle implementazioni per i metodi definiti da questa interfaccia, che sono utilizzati da Spring Security per autenticare e autorizzare gli utenti. Anche se questi metodi non sono visibili nel codice fornito, sono:

1. **getAuthorities()**: Restituisce la collezione di GrantedAuthority (ruoli/permessi) dell'utente.
2. **getPassword()**: Restituisce la password dell'utente.
3. **getUsername()**: Restituisce il nome utente dell'utente.



### AuthTokenFilter

La classe AuthTokenFilter estende la classe OncePerRequestFilter, progettata per garantire che il filtro venga eseguito una sola volta per ogni richiesta HTTP. Questo filtro viene utilizzato per gestire l'autenticazione basata su JWT (JSON Web Token).

Il filtro verifica la presenza all’interno della richiesta di un token valido. Si costruisce il modello dell’utente autenticato, recuperando i valori dal database.

Viene settato il *SecurityContextHolder.getContext().setAuthentication(authentication)*: l'oggetto Authentication viene impostato nel SecurityContextHolder, che è il meccanismo di Spring Security per mantenere il contesto di sicurezza per il thread corrente. In altre parole, Spring Security ora riconosce l'utente come autenticato per il resto della richiesta.

Infine la richiesta viene passata al filtro successivo nella catena di filtri, indipendentemente dal fatto che l'autenticazione sia avvenuta o meno. Questo è importante per garantire che la richiesta possa proseguire attraverso altri filtri e componenti del sistema.



### AuthEntryPointJwt

La classe AuthEntryPointJwt implementa l'interfaccia AuthenticationEntryPoint di Spring Security, che viene utilizzata per gestire le richieste non autenticate o per rispondere quando un utente tenta di accedere a una risorsa protetta senza essere autenticato.

Il metodo *commence()* gestisce la risposta HTTP quando si verifica un errore di autenticazione risponde con uno status code HTTP 401 (Unauthorized) indicando che l'utente non è autorizzato.



### AuthController

Questo controller è il punto di ingresso per le richieste da parte dei client per login, registrazione e logout.



### JwtUtils

La classe JwtUtils è una classe di utilità per la gestione dei token JWT (JSON Web Token). Questa classe fornisce metodi per creare, validare e estrarre informazioni dai token JWT, e per gestire i cookie che contengono tali token.

Questa classe è fondamentale per un'architettura basata su JWT in cui l'autenticazione e l'autorizzazione sono gestite in modo stateless (senza sessione) attraverso token che viaggiano tra client e server.

**Variabili di Classe**

* **jwtSecret**: Una stringa segreta utilizzata per firmare i token JWT. Questa chiave è crittografata utilizzando l'algoritmo HMAC SHA.
* **jwtExpirationMs**: Indica la durata in millisecondi della validità del token JWT (in questo caso 24 ore).
* **jwtCookie**: Nome del cookie che contiene il token JWT (impostato a "Bearer").

**Metodi Principali**

**1. getJwtFromCookies(HttpServletRequest request)**

Questo metodo estrae il token JWT dai cookie presenti nella richiesta HTTP.

**2. generateJwtCookie(UserDetailsImpl userPrincipal)**

Utilizza il metodo generateTokenFromUsername() per creare un token JWT basato sul nome utente e sull'ID dell'utente. Poi, costruisce un cookie HTTP-only (contenente il token JWT, con una durata massima di 24 ore.

**3. getCleanJwtCookie()**

Questo metodo può essere utilizzato per rimuovere il cookie JWT impostandolo a null.

**4. getUserNameFromJwtToken(String token)**

Utilizza la libreria JJWT (io.jsonwebtoken) per analizzare il token JWT, decodificandolo con la chiave segreta e restituendo lo username memorizzato nel corpo del token.

**5. getUserIdFromJwtToken(String token)**

questo metodo analizza il token JWT e restituisce l'ID utente che è memorizzato.

**6. key()**

Decodifica la stringa jwtSecret in un array di byte utilizzando Decoders.BASE64.decode() e genera una chiave HMAC SHA (hmacShaKeyFor) che verrà utilizzata per firmare o verificare i token.

**7. validateJwtToken(String authToken)**

Prova a eseguire il parsing del token JWT. Se l'operazione ha successo, il token è valido e viene restituito true. Se si verifica un'eccezione, come un token scaduto, malformato o non supportato, viene restituito false.

**8. generateTokenFromUsername(String username, Long id)**

Costruisce un token JWT impostando il subject (nome utente), aggiungendo l'ID dell'utente come claim personalizzato, impostando la data di emissione e la data di scadenza del token, e infine firmando il token con l'algoritmo HMAC SHA256 e la chiave generata dal metodo key().

## API security: gestione sicurezza immagini

Come visto in precedenza la visualizzazione dei Work è protetta in dalla visualizzazione non autorizzata a livello di query. Le singole immagini sono protette dal watermark, ma un’utente potrebbe tentare l’accesso alle immagini a dimensione intera facendo richiesta all’API sotto, provando una serie di id.

Per evitare che visualizzi immagini appartenenti a Work per i quali non ha accesso, viene recuperato dal DB il Work a cui appartiene l’immagine e viene verificato che l’utente ne abbia effettivamente diritto di visualizzazione. In caso positivo viene restituita l’immagine protetta dal watermark, in caso contrario viene restituito uno status code FORBIDDEN.



## Descrizione componenti di utility

Oltre ai componenti visti in precedenza che rappresentano i pilastri del progetto sono stare realizzate anche le due classi di “servizio” BlockchainTransactionService e ThumbnailGenerator, che fornisco metodi ausiliari per la registrazione di nuovi blocchi all’interno della Blockchain e l’applicazione di watermark sulle immagini.

La classe ThumbnailGenerator assolve due compiti:

* La generazione Thumbnail a partire da un oggetto ImageProject. Estrae il path in cui è contenuta l’immagine originale e il path dove andare a salvare le Thumbnail generate con la classe Image del package java.awt.Image. La classe permette di ridurre la dimensione al 30% dell’immagine originale.
* L’applicazione del watermark "© MyPortfolio 2024" in rosso, con il testo semi-trasparente con un'opacità dell'80% e nella parte bassa di un’immagine di tipo BufferedImage fornita in input



Snippet 4: funziona makeThumbnail che si occupa di generare e salvare nel path indicato, una nuova immagine ridotta del 30% rispetto all’orignale. La partedi verifica della presenza dell’immagine, la verifica e/o creazione del path di destinazione dell’immagine creata è omesso.



Snippet 5: funzione che presa in input una Buffered Image restituisce un nuovo oggetto BufferedImage, ma con l’aggiunta di una filigrana testuale (con testo semi-trasparente con un'opacità dell'80%.) applicata nella parte bassa dell’immagine e centrato rispetto all’ascisse.

La classe BlockchainTransactionService assolve due compiti:

* Registrazione di un nuovo acquisto come nuovo blocco all’interno della Blockchain
* Verifica della veridicità di un hash di un blocco

Vista l’importanza della parte dedicata alla Blockchain, per non rischiare i fare troppa sintesi di questi aspetti, tutte le componenti in gioco verranno riprese e espanse nella sezione *“Sezione shop e blockchain”*

# Implementazione lato DB

I tool utilizzati per lo strato di persistenza sono:

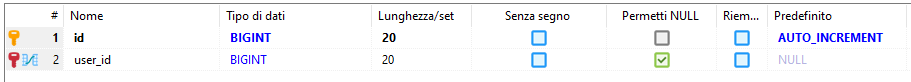
* ORM: Hibernate 6.4.4 Final
* RDBMS: MariaDB 11.2.
* Tool di amministrazione: HeidiSQL 12.3.0.6589

In questa sezione analizziamo nel dettaglio come il framework Hibernate ha gestito (sia con le annotazioni inserite nelle classi dei Model, sia con i propri automatismi) la [persistenza](https://it.wikipedia.org/wiki/Persistenza_(informatica)) dei dati sul database dei modelli definiti come oggetti Java.

Le tabelle create sul database, a partire dai model sono le seguenti:

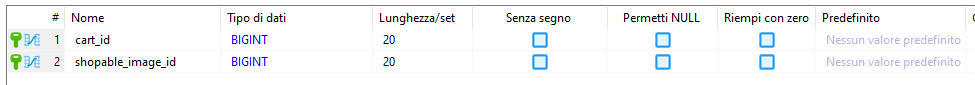


Segue adesso un dettaglio delle singole tabelle, riportate nello stesso ordine dell’immagine precedente, con alcuni commenti nelle tabelle più interessanti

**cart**

**cart\_shopable\_image**

Questa tabella permette di registrare quali sono le ShopableImage che un certo utente ha deciso di aggiungere al proprio carello, per poi eventualmente procedere con l’acquisto.



**hibernate\_sequences**

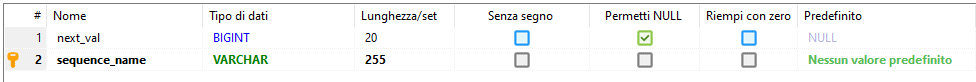
La tabella hibernate\_sequences viene creata automaticamente da Hibernate quando si utilizza la strategia di generazione delle chiavi primarie GenerationType.TABLE, che permette di generare identificatori unici per le entità gestite da Hibernate.  
Quando si decide di utilizzare GenerationType.TABLE viene utilizzata una tabella specifica per tenere traccia dei numeri di sequenza utilizzati per generare gli ID delle entità. Questo approccio è utile quando si desidera avere un controllo personalizzato su come vengono generati gli ID o quando il database non supporta altre strategie di generazione come AUTO o SEQUENCE.

La tabella hibernate\_sequences contiene due colonne:

1. ***sequence\_name:*** il nome della sequenza (o tabella) per la quale si stanno generando gli ID.
2. ***next\_val:*** il valore successivo che verrà utilizzato per l'ID.

Quando Hibernate genera un nuovo ID per un'entità, consulta questa tabella per ottenere il valore corrente, incrementa il valore e aggiorna la tabella con il nuovo valore. Questo meccanismo garantisce che ogni entità abbia un ID univoco.

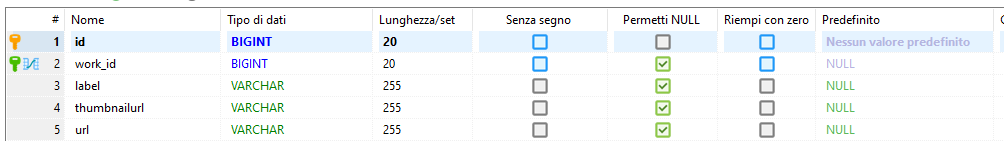
Hibernate: create table hibernate\_sequences (next\_val bigint, sequence\_name varchar(255) not null, primary key (sequence\_name)) engine=InnoDB

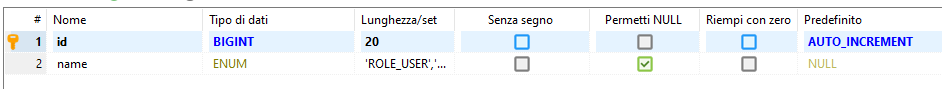


**image**

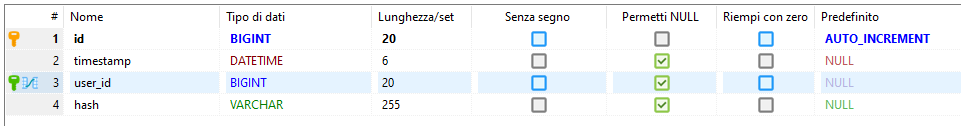
Su questa tabella vengono salvate le entità create tramite il model ImageProject, il parametro work\_id specifica a quale Work deve essere associata l’immagine.

Per defzione di Image all’interno dei requisiti, tutti i parametri in questa tabella non possono essere NULL.

**roles**



**sales\_order**

Questa tabella permette di registrare gli acquisti fatti dagli utenti; oltre all’ID dell’utente che ha fatto l’acquisto, viene salvato il timestamp e l’hash (o digest) che viene restituito dallo script di registrazione nuovo block dettagliato nella sezione *“Sezione shop e blockchain”* 

**shopable\_image**

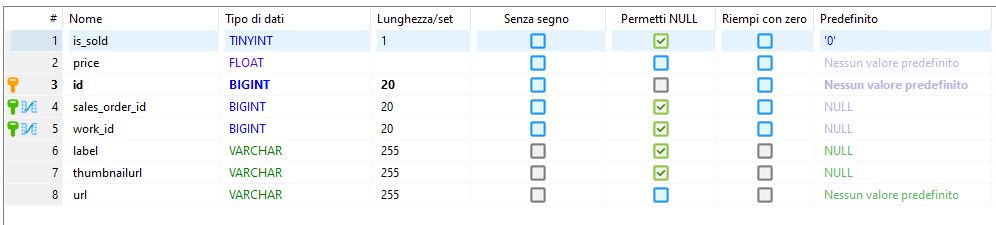
Questa tabella memorizza le immagini che l’admin ha deciso di mettere in vendita all’interno dello shop. Poiché la classe (model) ShopableImage è realizzata sfruttando l’ereditarietà a partire dal model ImageProject e poiché è stata utilizzata l'annotazione @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS), allora all’interno del database ogni classe concreta nella gerarchia di ereditarietà ha la propria tabella separata nel database.

Non esiste una tabella condivisa che rappresenti la superclasse astratta (se presente), e ogni tabella contiene tutte le colonne necessarie per rappresentare completamente le istanze delle rispettive sottoclassi.

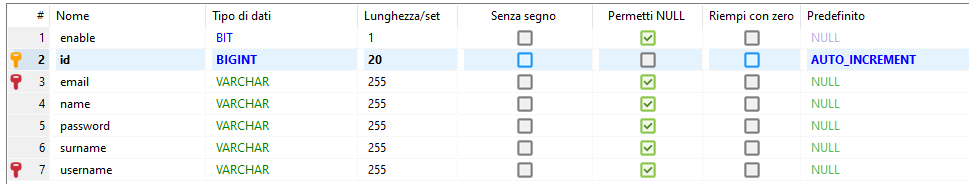
Oltre ai campi ereditati, questa classe ha in più i seguenti parametri:

* ***price:*** che indica il prezzo di vendita
* ***sales\_order\_id:*** se compilato indica l’id dell’ordine con il quale è stata acquista l’immagine
* ***is\_sold:*** indica se quell’immagine è stata acquistata o meno

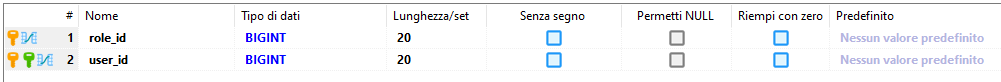
Quando viene aggiunta una nuova immagine alla vendita, questa farà parte del Work con ID 0.



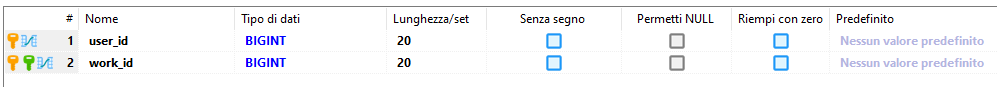
**user**



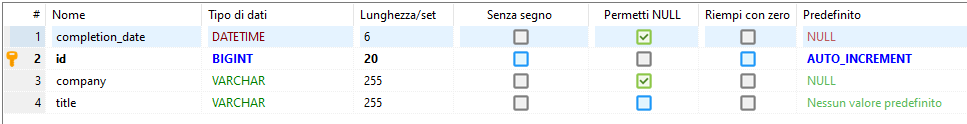
**user\_roles**



**visible\_work**



**work**



# Implementazione front-end

Il front-end non è ancora completo

# Sezione shop e blockchain

-circa la parte dedicata alla Blockchain, ricordati poi di documentare bene la piattaforma che selezionerai per la certificazione del dato e le API che adotterai).

# API

















# Analisi API - tabella di copertura

- nei diagrammi dei casi d'uso ti consigliamo di indicare per ciascuno di essi un identificativo (tipo UC#1) e anche di predisporre una "tabella di copertura" tra i casi d'uso e i requisiti stilati (i quali andranno anche essi numerati).

Aspetto di avere anche il front-end per inserire eventuali screen