



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

# MyPortfolio

Implementazione di una piattaforma Backend +  
Frontend con architettura API RESTful

Andrea Neri – 7060638

Docenti:

Enrico Vicario

Jacopo Parri

Samuele Sampietro

# Indice

- Introduzione
- Obiettivi
- Requisiti
- Analisi del Domain Model
- Implementazione Back-End
  - Deployment Diagram
  - Model – DAO
  - DTO
  - Repository
- Gestione copyright delle immagini
- API Security
- Implementazione front-end
- Sezione Shop
  - Blockchain

# Introduzione

- Il progetto nasce da una richiesta di un fotografo per condividere cartelle del NAS in modo selettivo e sicuro.
- Prime soluzioni valutate:
  - Utilizzo dell'app del produttore del NAS che consente l'accesso ai file e la gestione dei permessi. Scartata per la necessità di condividere l'intera libreria del NAS sulla rete pubblica.
  - Utilizzo di piattaforme come WeTransfer, soluzione scomoda e temporanea per la condivisione dei file.
- Esigenze principali:
  - Visualizzazione di immagini con watermark per utenti registrati con permessi specifici.
  - Gestione delle foto tramite raccolte (Work) che rappresentano progetti fotografici o temi comuni, per una gestione accurata degli accessi.

## Obiettivi

- 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).
- L'utenza standard, una volta registrata e loggata, potrà accedere in visualizzazione alla/e cartella/e in base alle autorizzazioni fornite dall'amministratore.
- La visualizzazione della foto avviene tramite una galleria fotografica (una per ogni cartella o area di lavoro alla quale l'utente ha accesso) sfruttando le thumbnail in modo da rendere più veloce il caricamento della pagina, che al click verranno mostrate a tutto schermo.

## Requisiti (1)

### Vincoli:

- Le foto vengono memorizzate su un NAS, e nel database sono salvati solo gli URL delle immagini.
- Sono accettati solo formati fotografici standard (PNG, JPEG, JPG, BMP); non sono supportati i formati non renderizzati (NEF, CR2, ecc.).

## Requisiti (2)

### Requisiti non funzionali

- L'architettura del sistema deve essere basata sul modello MVC.
- Il back-end deve essere sviluppato utilizzando JakartaEE con il framework Spring.
- Il front-end deve essere sviluppato utilizzando Vue JS e Bootstrap.
- L'architettura deve garantire la portabilità del sistema.
- Gestione del Copyright: il sistema deve garantire il rispetto del copyright delle immagini

## Requisiti (3)

### Requisiti funzionali

- Due tipologie di utenti: Admin e User, con permessi differenziati.
- Pannello di controllo per l'amministratore per gestire work, immagini, autorizzazioni e la sezione Shop.
- Le immagini saranno catalogate in "work" e supportate in formati standard, con generazione automatica di miniature e watermark.
- Gestione fine dei permessi di autorizzazione e accesso
- La sezione Shop permette agli utenti di acquistare immagini, con registrazione degli acquisti sulla blockchain.

## Analisi del Domain Model (1)

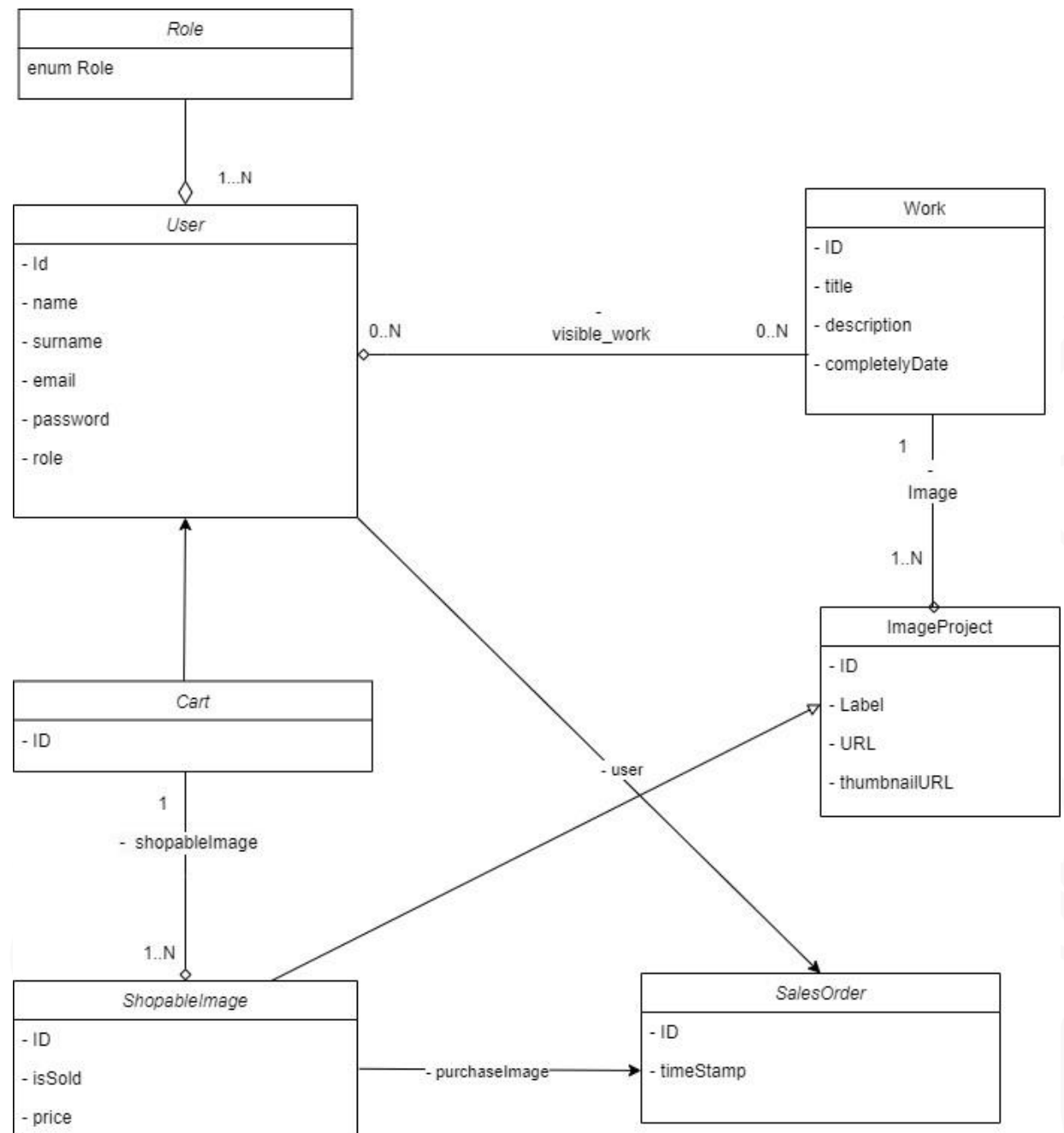
A partire quindi da vincoli, requisiti non funzionali e requisiti funzionali sono state estrapolate le seguenti entità:

- **UserWork**: rappresenta un lavoro fotografico o meglio un contenitore logico per raggruppare più immagini legate a uno stesso progetto fotografico.
- **ImageProject**: rappresenta ogni scatto fotografico all'interno di un progetto specifico (Work). Non può esistere senza un Work di riferimento.
- **ShoppableImage**: rappresenta una specializzazione di un ImageProject arricchita da informazioni aggiuntive per essere venduta nello shop online.
- **Cart**: rappresenta il carrello dell'utente.
- **SalesOrder**: rappresenta un ordine di acquisto.
- **Role**: rappresenta il ruolo di un utente (attualmente i ruoli presenti sono Admin e User).

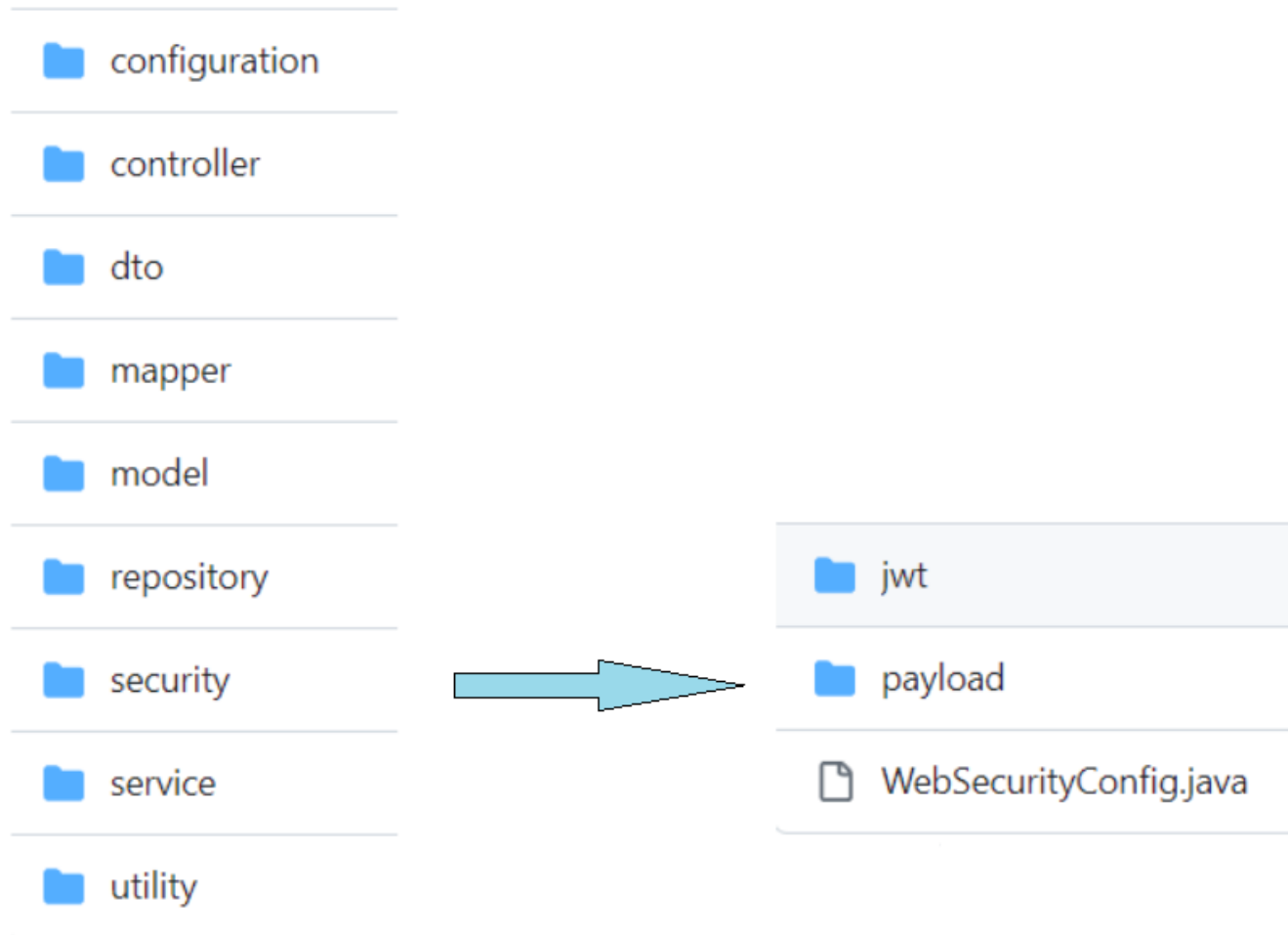


## Analisi del Domain Model (2)

- 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 zero o più SalesOrder (associazione 0..N).



## Struttura package back-end



## Implementazione back-end

- Spring Boot 3.3.2.
- IDE Spring Tool Suite 4.21.0.
- 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 e Insomnia 10.0.0

## Implementazione back-end (Model 1)

```
1  @Entity
2  @Table(name = "work")
3  public class Work {
4
5      @Id
6      @GeneratedValue(strategy = GenerationType.IDENTITY)
7      @Column(name = "id")
8      private Long Id;
9
10     @Column(nullable = false)
11     private String title;
12
13     private String company;
14
15     @Column(name = "completion_date")
16     private Date completionDate;
17
18     @OneToMany(cascade = CascadeType.ALL)
19     @JoinColumn(name = "work_id")
20     private Set<ImageProject> image;
21
22     @ManyToMany(fetch = FetchType.LAZY)
23     @JoinTable(name = "visible_work",
24               joinColumns = @JoinColumn(name = "work_id"),
25               inverseJoinColumns = @JoinColumn(name = "user_id"))
26     private Set<User> users = new HashSet<>();
27
28     ...
29 }
```

## Implementazione back-end (Model 2)

```
1  @Entity
2  @Table(name="image")
3  @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
4  public class ImageProject {
5
6      @Id
7      @GeneratedValue(strategy = GenerationType.TABLE)
8      @Column(name="id")
9      private Long Id;
10
11     private String label;
12
13     private String URL;
14
15     private String thumbnailURL;
16
17     ...
18 }
```

```
1  @Entity
2  @Table(name="shopable_image")
3  public class ShopableImage extends ImageProject{
4
5      @Column(name="is_sold", columnDefinition = "boolean default false")
6      private boolean isSold;
7
8      @Column(nullable = false)
9      private float price;
10
11      @ManyToMany(mappedBy = "shopableImages")
12      private List<Cart> cart;
13
14      ...
15  }
16 }
```

## Implementazione back-end (Repository/DAO - 1)

- **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)
- L'ambiente Spring mette a disposizione il modulo Spring Data che permette di semplificare lo stato di persistenza rimuovendo completamente l'implementazione dei DAO dall'applicazione.
- L'interfaccia DAO deve estendere JpaRepository in modo tale che Spring Data creerà automaticamente un'implementazione dotata dei metodi CRUD più rilevanti per l'accesso ai dati.
- Per l'utilizzo basta includere nel pom.xml spring-boot-starter-data-jpa

```
1 spring.datasource.url=jdbc:mariadb://localhost:3306/myportfolio-db
2 spring.datasource.username=root
3 spring.datasource.password=root
4 spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
5 spring.jpa.hibernate.ddl-auto=create-drop
6 spring.jpa.show-sql=true
```

## Implementazione back-end (Repository/DAO – 2)

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

```
public interface JpaRepository <T, ID> extends PagingAndSortingRepository <T, ID>
{
    }
}
```

Dove:

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

## Implementazione back-end (Repository/DAO – 3)

- *JpaRepository* dispone di alcuni metodi CRUD, 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.
- Per impostazione predefinita, i metodi ereditati da *crudrepository* hanno la seguente configurazione:
  - i metodi di lettura (come *findbyid*, *findall*, etc.) sono considerati Read-Only
  - i metodi di scrittura (come *save*, *delete*, *deletebyid*, etc.) sono transazionali: le operazioni di scrittura vengono eseguite all'interno di una transazione



## Implementazione back-end (Metodi CRUD inclusi)

- CRUD Repository: Spring Data JPA fornisce metodi già pronti per le operazioni CRUD (Create, Read, Update, Delete) attraverso interfacce come *CrudRepository*, *PagingAndSortingRepository*, e *JpaRepository*.
- Metodi CRUD comuni:
  - *deleteById*: Elimina un'entità dal database tramite il suo ID
  - *existsById*: Verifica l'esistenza di un'entità tramite
  - *findById*: Cerca un'entità per ID
  - *save*: Salva o aggiorna un'entità
  - *findAll*: Restituisce tutte le entità
  - *count*: Conta il numero totale di entità
  - *exists*: Controlla l'esistenza di un'entità.

## Implementazione back-end (Query Methods)

- Query Methods: Metodi che eseguono query SQL automaticamente basati su convenzioni nei nomi. Non è necessario scrivere manualmente query SQL o JPQL
- Keyword supportate:
  - Per la selezione/conteggio: *findBy*, *countBy*, *deleteBy*, *existsBy*
  - Per concatenare combinazioni logiche e confronti: *And*, *Or*
  - Per confronti numerici e pattern: *GreaterThan*, *LessThan*, *Between*, *Like*, *In*
  - Per ordinare i risultati: *OrderBy*
- 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

## Implementazione back-end (UserRepository)

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT w FROM User u JOIN u.visibleWorks w WHERE u.id = :userId")
    public Set<Work> findVisibleWorksById(@Param("userId") Long userId);

    Optional<User> findByUsername(String username);

    Boolean existsByUsername(String username);

    Boolean existsByEmail(String email);

    Optional<User> findByUsernameAndEnable(String username, boolean enable);
}
```

## Implementazione back-end (DTO 1)

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
2. Sicurezza dei dati
3. Ridurre il sovraccarico delle comunicazioni: i DTO consentono di inviare solo i dati necessari, riducendo la quantità di informazioni che viaggiano sulla rete.
4. Separazione dei concetti
5. Aggiornamenti/funzionalità future

## Implementazione back-end (DTO 2)

```
public class DetailsSalesOrderDTO {  
  
    private Long Id;  
    private Date timestamp;  
    private List<ShopableImageDTO> purchaseImage;  
    private Float totalPrice;  
    private String username;  
    private int piece;  
    private String hash;  
  
    ...  
}
```

```
public class UserPersonalDetailsDTO {  
  
    private Long Id;  
    private String surname;  
    private String name;  
    private String email;  
    private Set<Role> role;  
  
    ...  
}
```

## Implementazione back-end (Controller) 1

- Gestisce le richieste HTTP (GET, POST, PUT, DELETE, PATCH), elabora le operazioni e restituisce risposte appropriate
- `@RestController`: Trasforma una classe Java in un controller RESTful in Spring Boot. È una versione specializzata di `@Controller` che include l'annotazione `@ResponseBody`, consentendo di restituire direttamente i dati nel corpo della risposta (es. JSON o XML), senza passare da una vista.
- Sono stati realizzati i seguenti controller:
  - `AuthController`
  - `CartController`
  - `ImageController`
  - `ShopableImageController`
  - `ShopController`
  - `ThumbnailController`
  - `UserController`
  - `WorkController`

# Implementazione back-end (Controller) 2

```
@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/work")
public class WorkController {

    @Autowired
    WorkService workService;

    @Autowired
    private UserService userService;

    @Autowired
    JwtUtils jwtUtils;

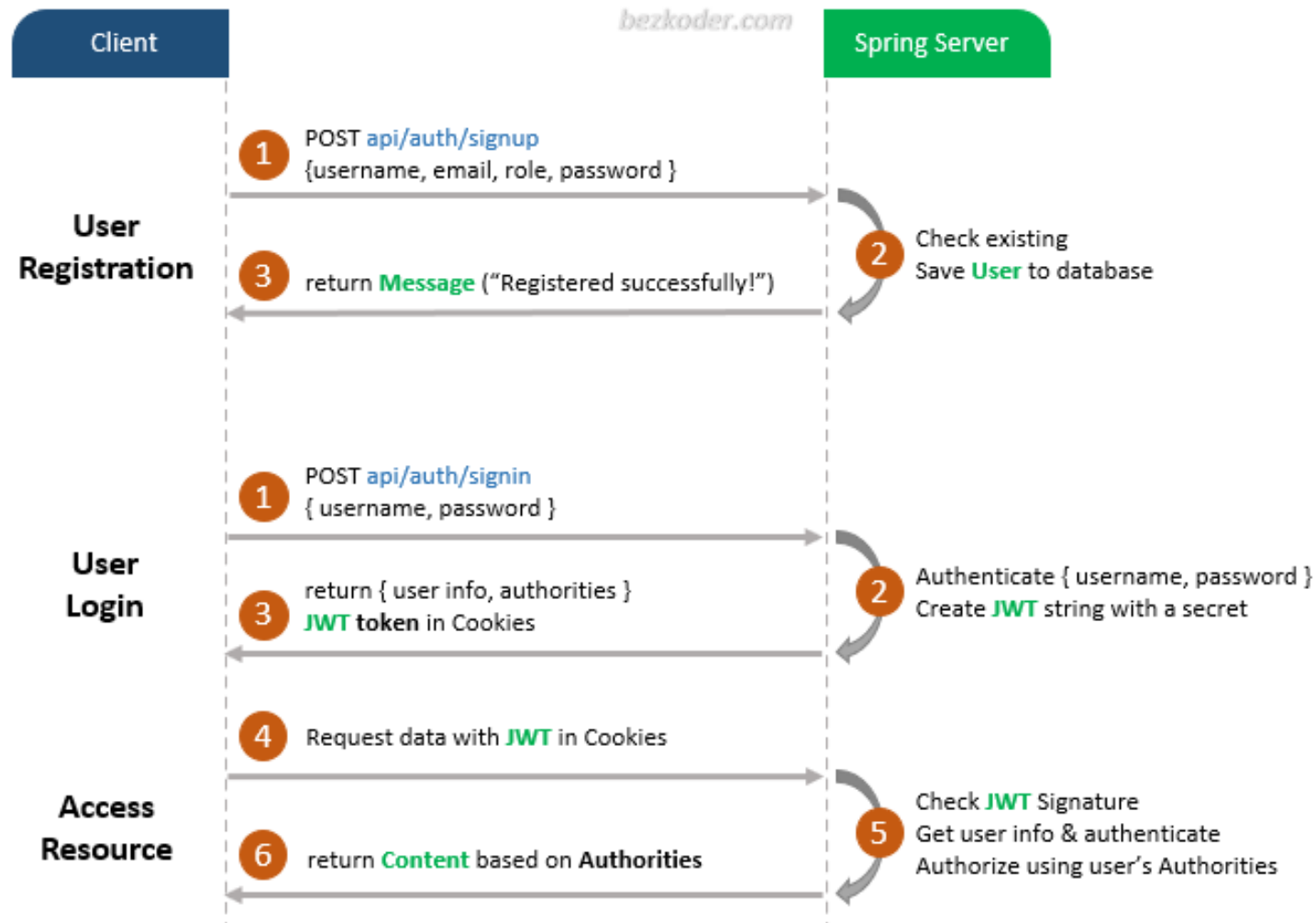
    @PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
    @GetMapping("/mywork")
    public ResponseEntity<Set<WorkDTO>> getVisibleWorksByUserId(HttpServletRequest request) {
        String token = jwtUtils.getJwtFromCookies(request);
        if (jwtUtils.validateJwtToken(token)) {
            Long userId = (Long) jwtUtils.getUserIdFromJwtToken(token);
            Set<Work> works = userService.findVisibleWorksByUserId(userId);
            Set<WorkDTO> workDtos = new HashSet<>();

            for (Work work : works) {
                workDtos.add(WorkDTO.fromWork(work));
            }

            if (workDtos.isEmpty()) {
                return ResponseEntity.notFound().build();
            }

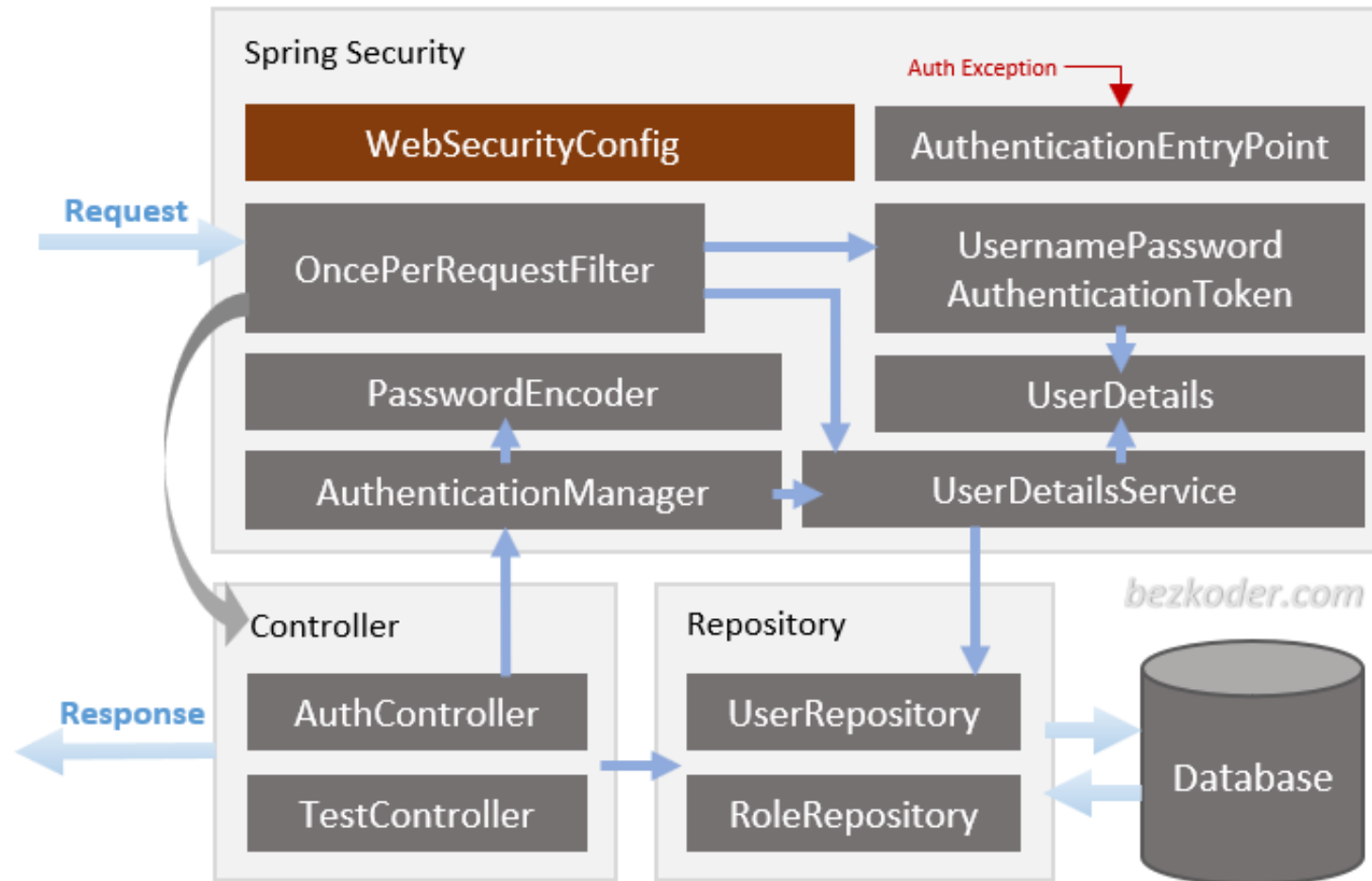
            return ResponseEntity.ok(workDtos);
        } else {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }
    }
}
```

# Implementazione back-end (API Security) 1





## Implementazione back-end (API Security) 2



## Implementazione back-end (Gestione copyright immagini)

- Requisiti fondamentali:
  - Visualizzazione di immagini protette per utenti autorizzati
  - Prevenzione dell'uso non autorizzato attraverso watermark
  - Creazione di Thumbnail: versioni ridotte (30%) delle immagini, generate per ridurre consumo di banda e spazio, con l'applicazione di watermark visibile per disincentivare usi illeciti.
  - Applicazione del watermark lato back-end per immagini a risoluzione standard applicato **dinamicamente a Runtime** senza modificare l'originale, mantenendo la qualità e l'integrità.
- Vantaggi del sistema:
  - **Protezione del copyright:** Watermark applicato su tutte le immagini visualizzate.
  - **Conservazione dell'originale:** L'immagine senza watermark rimane disponibile per usi legittimi.

## Implementazione back-end (Gestione autorizzazioni immagini)

- La visualizzazione dei Work è protetta 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 provando una serie di id.
- Per evitare che un utente 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.

## Implementazione back-end (visione d'insieme gestione immagini)

```
@GetMapping()  
@PreAuthorize("hasRole('USER') or hasRole('ADMIN')")  
public ResponseEntity<byte[]> getImageById(@RequestParam Long id, HttpServletRequest request) throws IOException {  
  
    ImageProject image = imageService.getImageById(id);  
    if (image != null) {  
        String token = jwtUtils.getJwtFromCookies(request);  
        if (jwtUtils.validateJwtToken(token)) {  
            Long userId = (Long) jwtUtils.getUserIdFromJwtToken(token);  
            Optional<User> user = userService.getUserById(userId);  
            Work work = workService.getWorkByImageId(userId);  
  
            if (work.getUsers().contains(user.get())) {  
                BufferedImage sourceImage = ImageIO.read(new File(image.getUrl()));  
  
                // Add the watermark using the addTextWatermark function  
                BufferedImage watermarkedImage = ThumbnailGenerator.addTextWatermark(sourceImage);  
  
                // Converti BufferedImage in array di byte  
                byte[] imageBytes = convertImageToBytes(watermarkedImage);  
  
                return ResponseEntity.status(HttpStatus.OK)  
                    .header(org.springframework.http.HttpHeaders.CONTENT_DISPOSITION)  
                    .contentType(MediaType.IMAGE_PNG).body(imageBytes);  
            } else {  
                return ResponseEntity.status(HttpStatus.FORBIDDEN).build();  
            }  
        } else {  
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();  
        }  
    }  
    return ResponseEntity.notFound().build();  
}
```

## Sezione Shop

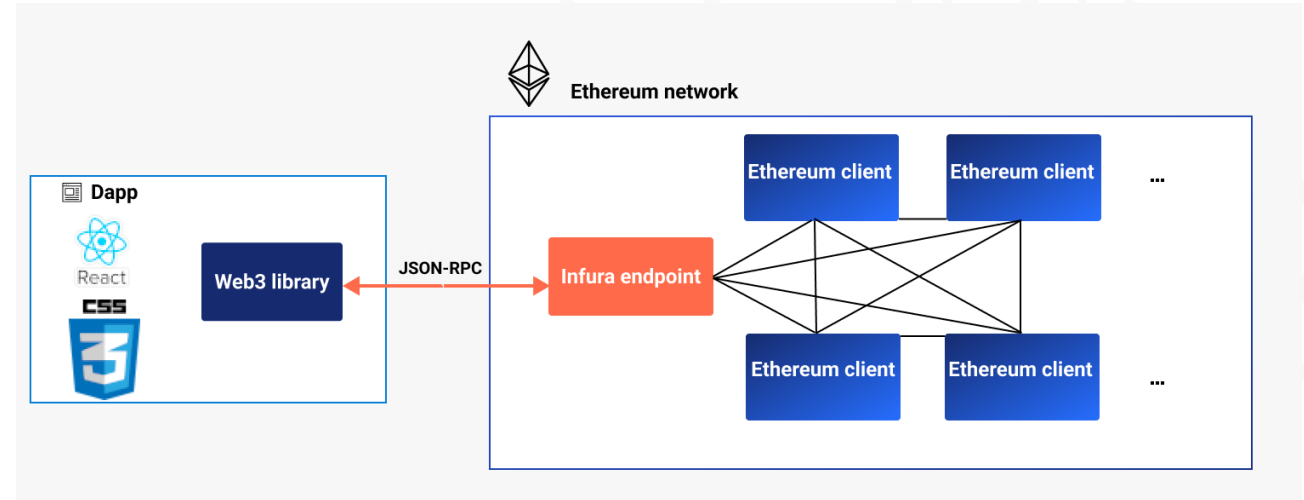
- L'amministratore può caricare immagini in vendita, denominate ShopableImage, disponibili in edizione unica (opere fotografiche a tiratura unica).
- Gli utenti registrati possono acquistare queste immagini.
- Ogni transazione (acquisto) viene registrata come un blocco su una blockchain Ethereum tramite l'integrazione della piattaforma Infura.
- Questa soluzione garantisce la sicurezza e la tracciabilità di ogni transazione, conferendo unicità agli acquisti effettuati all'interno dell'applicazione.

## Sezione Shop – Piattaforma Infura

- Infura è una piattaforma che fornisce accesso a diverse blockchain, permettendo agli sviluppatori di interagire con queste reti senza dover configurare un nodo completo
- Accesso semplificato alle principali blockchain (Ethereum, Polygon, Optimism, Arbitrum, IPFS) tramite API HTTP e WebSocket senza necessità di gestire nodi locali.
- Può essere utilizzata per effettuare/registrare transazioni senza essere controllato da un'autorità centrale, utilizzando il meccanismo di consenso Proof of Stake (PoS).
- Testnet utilizzate:
  - Holesky: Progettata per test su larga scala, simula la mainnet con oltre 1,4 milioni di validatori. Ideale per simulazioni realistiche e test intensivi.
  - Sepolia: Ambiente di test leggero e stabile, ideale per test mirati e veloci con meno overhead

## Sezione Shop – Transazioni (1)

- Protocollo **JSON-RPC** (metodo che permette a un client di invocare funzioni in esecuzione su un server remoto).
- libreria **web3.py** per interagire con Ethereum  
[<https://web3py.readthedocs.io/en/stable/>]
- La transizione utilizzata è *eth.estimate\_gas* che genera e restituisce una stima di quanto gas è necessario per consentire il completamento della transazione. La transazione non verrà aggiunta alla blockchain, ma restituisce comunque un hash.
- Il gas è la quantità di **calcolo** necessaria per eseguire operazioni sulla rete Ethereum. Ogni operazione richiede una certa quantità di calcolo da parte dei nodi della rete. Il gas viene utilizzato per compensare i miner (o i validatori, in caso di Ethereum 2.0) che forniscono la potenza computazionale per eseguire queste operazioni.



## Blockchain – Transazioni (2)

```
from web3 import Web3, exceptions
import sys
import random

infura_url = 'https://holesky.infura.io/v3/5cb88f299e974e9082c695c5fb3e9b13'
private_key = '#' #metamask
from_account = # #metamask
to_account = '0x00C007CFf2bAe0b56d667e5ce421FBA83B007'+
              str(random.randint(100, 999)) # random
web3 = Web3(Web3.HTTPProvider(infura_url))

try:
    from_account = web3.to_checksum_address(from_account)
except exceptions.InvalidAddress:
    print(f"Invalid 'from_account' address: {from_account}")

try:
    to_account = web3.to_checksum_address(to_account)
except exceptions.InvalidAddress:
    print(f"Invalid 'to_account' address: {to_account}")

data = sys.argv[1]+";"+sys.argv[2]+";"+sys.argv[3]
nonce = web3.eth.get_transaction_count(from_account)
tx = {
    'type': '0x2',
    'nonce': nonce,
    'from': from_account,
    'to': to_account,
    'value': web3.to_wei(0, 'ether'),
    'maxFeePerGas': web3.to_wei('0', 'gwei'),
    'maxPriorityFeePerGas': web3.to_wei('0', 'gwei'),
    'chainId': 17000,
    'data': data.encode("utf-8").hex()
}
gas = web3.eth.estimate_gas(tx)
tx['gas'] = gas
signed_tx = web3.eth.account.sign_transaction(tx, private_key)
tx_hash = web3.eth.send_raw_transaction(signed_tx.rawTransaction)
print(str(web3.to_hex(tx_hash)))
```

```
import sys
from web3 import Web3

infura_url = 'https://holesky.infura.io/v3/5cb88f299e974e9082c695c5fb3e9b13'
web3 = Web3(Web3.HTTPProvider(infura_url))
print(web3.eth.get_transaction(sys.argv[1]))
```



## Front-end

Vediamolo in azione!



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

# Conclusione

