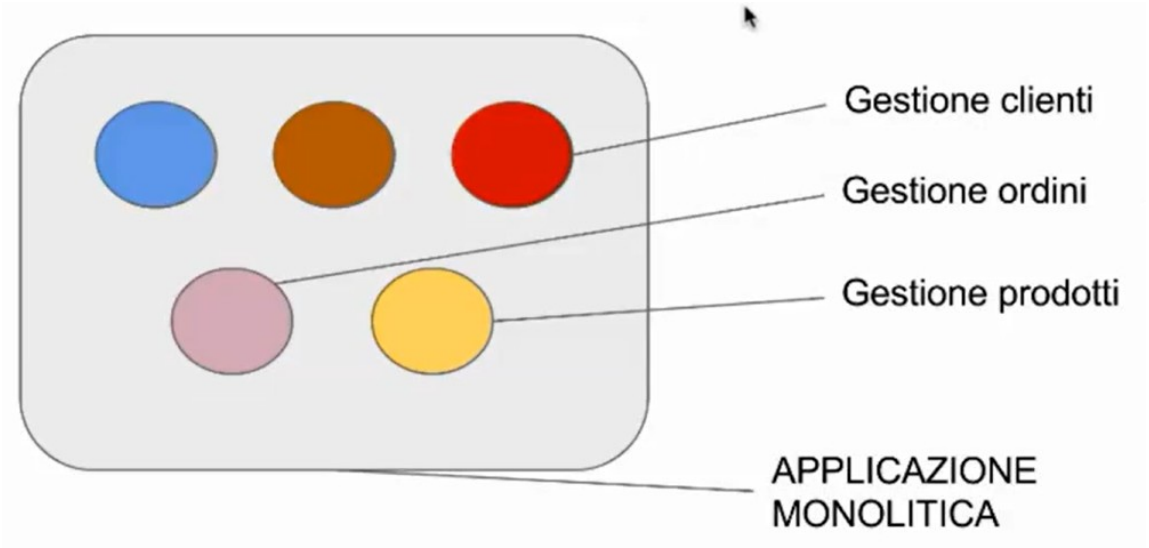
Sviluppo web a microservizi REST con Java Spring boot e AJAX

Indice generale

Applicazione monolitica vs microservizio: le differenze

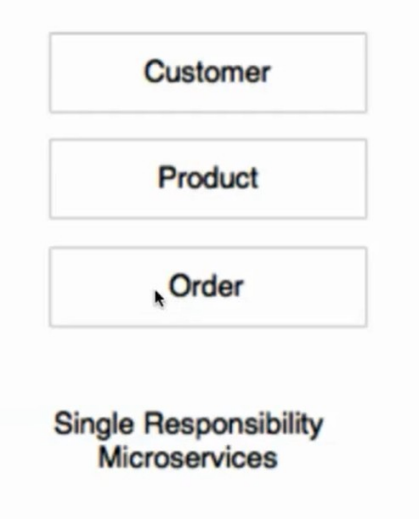
Le applicazioni monolitiche sono composte da varie funzioni, dette servizi, e ognuno di essi svolge un qualcosa per l'utente.

Tutti i servizi implementati sono contenuti tutti all'interno della stessa applicazione, come se fosse tutto avvolto in un unico contenitore.



Come si può vedere nell'esempio, ci sono vari servizi implementati nella stessa applicazione, una per la gestione dei clienti, un'altra per la gestione degli ordini e un'altra ancora per la gestione dei prodotti. Il tutto è contenuto in un'unica applicazione.

L'approccio ai microservizi è un approccio innovativo, perché permette la suddivisione dell'applicazione in più applicazioni indipendenti tra loro e interconnesse. Ciascuna si occupa di un aspetto ben preciso, implementando un particolare servizio.



Il vantaggio nell'utilizzare l'approccio a microservizi è quello di poter utilizzare per ogni microservizio delle tecnologie diverse, come ad esempio linguaggi di programmazione diversi.

Per ogni microservizio posso utilizzare anche database separati, o lo stesso database, e posso anche dividere l'applicazione in vari layer, a seconda del tipo di servizio che andiamo a implementare.

La stratificazione classica delle applicazioni con approccio a microservizi è la seguente:

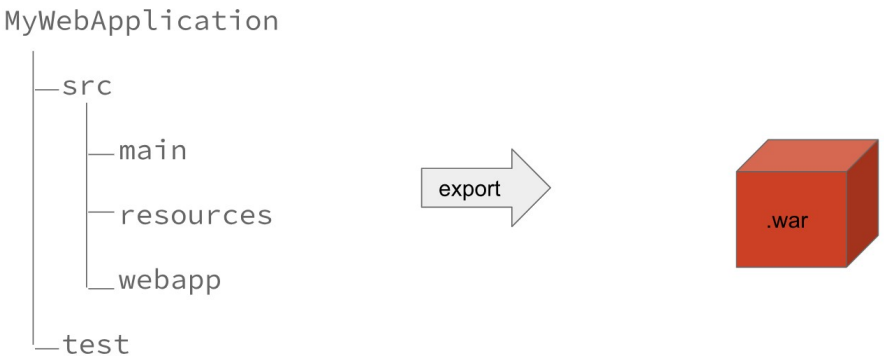
1. Presentation Layer, per la presentazione dell'applicazione all'utente.
2. Business Layer, dove è contenuta tutta la logica di business dell'applicazione.
3. Database Layer.

War vs Fat Jar

Il war è il formato classico in cui si deployano (rilasciano) le web application.

Tale formato va esportato dall'ambiente di sviluppo e inserito nella cartella specifica di Deploy del proprio Application Server (tomcat, jetty, glassfish, jboss, websphere, …).

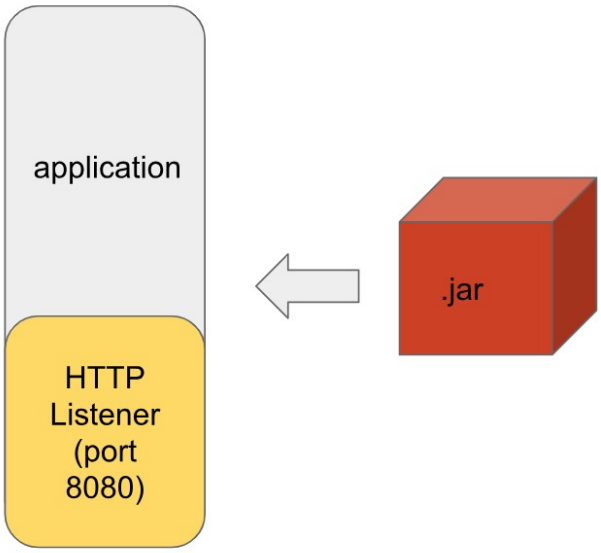
Quando il server è attivo, effettua il deploy della web application, ovvero la spacchetta e ne istanzia gli elementi, che entreranno in azione quando l'application server riceverà la richiesta specifica per quell'applicazione.



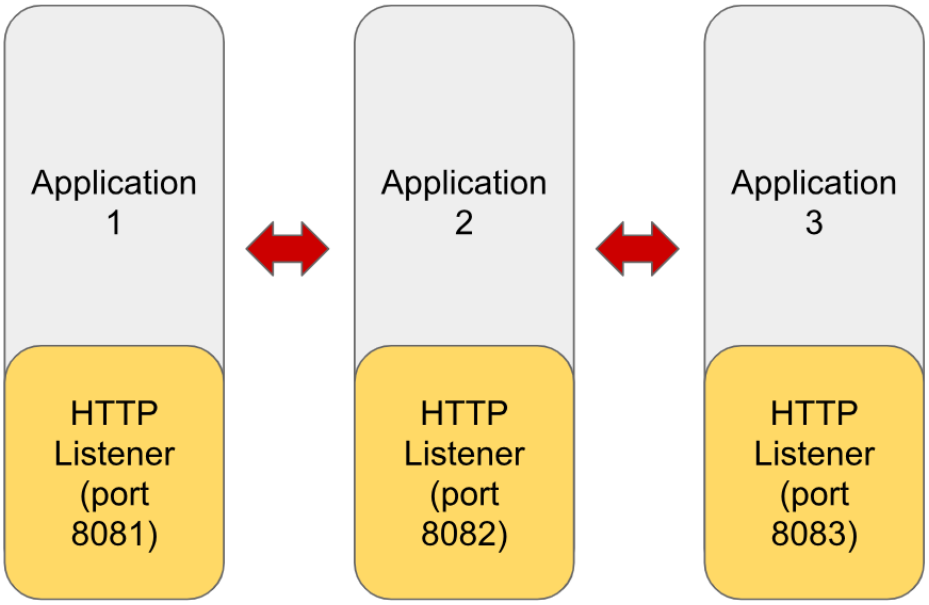
Normalmente i file war sono costituiti da un'applicazione con la seguente struttura, ovvero la cartella src, dove abbiamo tutto il codice, e la cartella test, dove abbiamo i vari test del codice per verificare la correttezza di ciò che abbiamo scritto.

Il formato fat jar (o uber jar) è invece il formato che ci fornisce Spring Boot e non ha bisogno di un'application server esterno per essere deployato, ma ha già al suo interno un'application server già installato, chiamato http listener. Quindi, le applicazioni nel formato fat jar si avviano normalmente con il metodo main(), come tutte le applicazioni classiche Java, avviando di conseguenza anche l'http listener.

Una volta eseguito il fat jar, si avvia la nostra applicazione web sotto forma di microservizio in ascolto su una specifica porta, senza dover deployare il file in un application server.



Dunque possiamo far interagire tra loro più microservizi (micro applicazioni), ognuno in ascolto su una porta specifica.



Model View Controller (MVC)

Il Model è la rappresentazione logica del dato e comprende tutte le operazioni che lo riguardano (create, read, update, delete), dette operazioni CRUD.

La View è quel layer del software che si occupa della rappresentazione del dato all'utente (pagine html, jsp, thymeleaf, JSON, XML, …).

Il Controller è il layer del software che si occupa della logica vera e propria (calcoli, decisioni e così via).

Model

Nelle moderne applicazioni il Model è composto da:

- Entità, che rappresenta il dato. Esso viene rappresentato attraverso una classe che avrà dei metodi che permettono l'accesso a degli attributi privati, ovvero i metodi getter e setter;

  Esempio: public class User{…}

- Data Access Object (DAO) è una classe che permette l'accesso al dato attraverso il richiamo di database, altri web service, o la memoria interna. Questa classe normalmente viene utilizzata per il salvataggio del dato, la modifica e tutte quelle operazioni che abbiamo detto prima (CRUD)

  Esempio: public class UserDao{...}

View

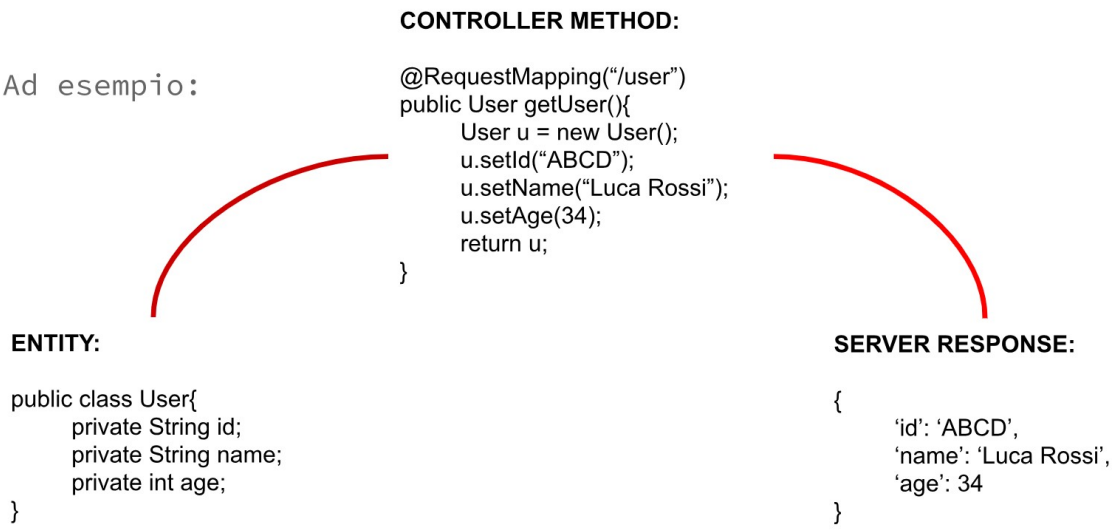
In Spring potrà succedere di vedere nei Controller dei metodi che restituiscono una stringa. Però essa non è realmente una stringa, ma spesso viene utilizzato un elemento tipico di Spring, che si chiama ViewResolver, che traduce la stringa in una pagina per la visualizzazione.

Esempio:

```
public String goToHomePage()
{
        return "home";                  //apre la pagina home.jsp
}
```

In Spring esiste anche un'altra libreria, oltre il ViewResolver, che si chiama Jackson Library (inserita automaticamente nelle dipendenze di progetto), e permette a Spring di trasformare l'oggetto, che andiamo a restituire attraverso il Controller, in un messaggio JSON che contiene tutti i dati di tale oggetto.

Ad esempio:



Controller

Il Controller layer in Spring è implementato grazie all'annotation @Controller, che smista le richieste http ai vari servizi (annotati in Spring con @Service) che andiamo a implementare e restituisce una risposta http al chiamante (un JSON, una pagina web e così via).

Creazione Primo Microservizio AccountMicroservice

Settaggio della porta del server e Command Line Runner

Application.properties

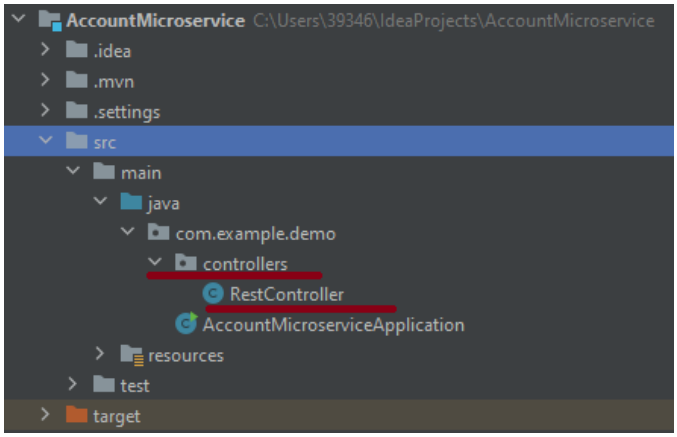
```
/* Di default la porta è 8080, ma è buona pratica
settarne una diversa */
server.port = 8094
```

AccountMicroserviceApplication.java

```
package com.example.demo;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication
public class AccountMicroserviceApplication implements CommandLineRunner {

	/* La classe Logger serve per loggare lo stato del sistema */
	private static final Logger log = LoggerFactory.getLogger(AccountMicroserviceApplication.class);

	public static void main(String[] args) {
		SpringApplication.run(AccountMicroserviceApplication.class, args);
	}

	/*Il metodo run(), ereditato dall'interfaccia CommandLineRunner, è il metodo
	* in cui possiamo implementare delle istruzioni che verranno eseguite prima
	* dell'avvio dell'applicazione*/
	@Override
	public void run(String... strings) throws Exception {
		//...
		log.info("Hello 1"); //Hello 1
	}
}
```

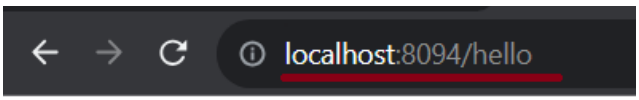
```
2023-11-18T16:25:35.101+01:00  INFO 1044 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'defau
2023-11-18T16:25:35.154+01:00  WARN 1044 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, databa
2023-11-18T16:25:35.725+01:00  INFO 1044 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8094 (http) with context path ''
2023-11-18T16:25:35.738+01:00  INFO 1044 --- [           main] c.e.demo.AccountMicroserviceApplication  : Started AccountMicroserviceApplication in 3.632 seconds (process
2023-11-18T16:25:35.742+01:00  INFO 1044 --- [           main] c.e.demo.AccountMicroserviceApplication  : Hello 1
2023-11-18T16:25:36.045+01:00  INFO 1044 --- [nio-8094-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2023-11-18T16:25:36.045+01:00  INFO 1044 --- [nio-8094-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
```

Primo Controller



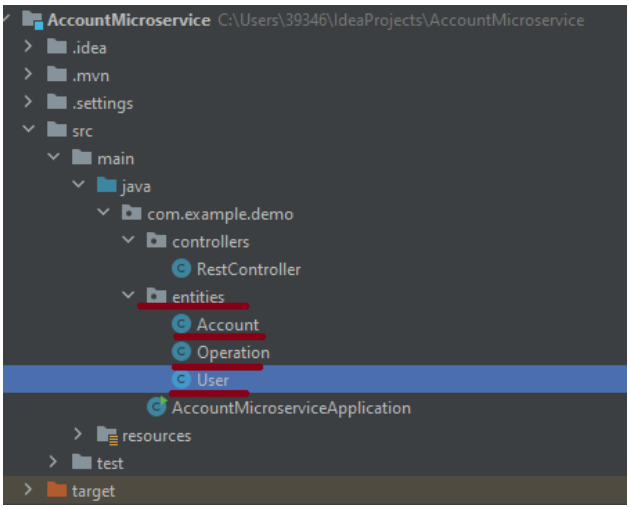
RestController.java

```java
package com.example.demo.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/* Il RestController è una classe di Spring i cui metodi si occupano
 di gestire tutte le richieste http */
@org.springframework.web.bind.annotation.RestController
public class RestController {

    /* @RequestMapping serve per mappare tutte le richieste http
    su degli indirizzi URL */
    @RequestMapping("/hello")
    /* @ResponseBody indica che ciò che verrà restituito dal metodo
    * che contrassegna, sarà direttamente il corpo del messaggio */
    public String sayHello(){
        return "Hello everyone!";
    }
}
```



Hello everyone!

Pojo (Plain Old Java Object)



User.java

```java
package com.example.demo.entities;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

/* Con @@AllArgsConstructor e @NoArgsConstructor
*  Spring ci definisce in automatico i costruttori
*  della classe con e senza argomenti*/
@AllArgsConstructor @NoArgsConstructor
/* @Entity mappa una tabella con il nome
 * definito in @Table */
@Entity
@Table(name = "users")
public class User {

    @Id
    @Column(name = "ID")
    @Getter @Setter
    private String id;

    @Column(name = "USERNAME")
    @Getter @Setter
    private String username;

    @Column(name = "PASSWORD")
    @Getter @Setter
    private String password;

    @Column(name = "PERMISSION")
    @Getter @Setter
    private String permission;
}
```

Account.java

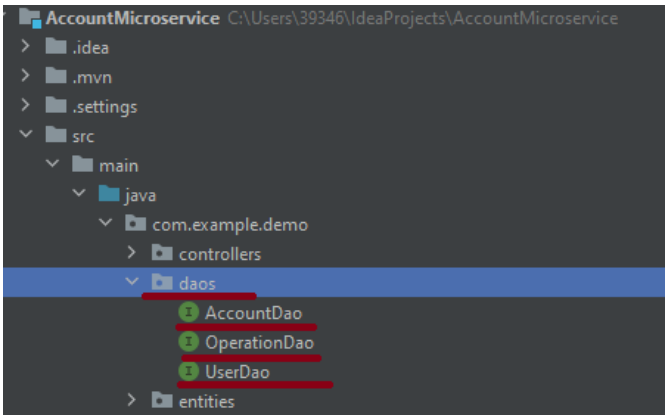
```java
package com.example.demo.entities;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Entity
@Table(name = "accounts")
@AllArgsConstructor @NoArgsConstructor
public class Account {

    @Id
    @Column(name = "ID")
    @Getter @Setter
    private String id;

    @Column(name = "FK_USER")
    @Getter @Setter
    private String fkUser;

    @Column(name = "TOTAL")
    @Getter @Setter
    private Double total;
}
```

Operation.java

```java
package com.example.demo.entities;

import jakarta.persistence.Id;
import jakarta.persistence.Entity;
import jakarta.persistence.Table;
import jakarta.persistence.Column;
import jakarta.persistence.PrePersist;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.util.Date;

@Entity
@Table(name = "operations")
@AllArgsConstructor @NoArgsConstructor
public class Operation {

    @Id
    @Column(name = "ID")
    @Getter @Setter
    private String id;

    @Column(name = "DATE")
    @Getter @Setter
```

```
    private Date date;

    @Column(name = "DESCRIPTION")
    @Getter @Setter
    private String description;

    @Column(name = "VALUE")
    @Getter @Setter
    private Double value;

    @Column(name = "FK_ACCOUNT1")
    @Getter @Setter
    private String fkAccount1;

    @Column(name = "FK_ACCOUNT2")
    @Getter @Setter
    private String fkAccount2;

    /* @PrePersist serve per settare la data direttamente dal sistema */
    @PrePersist
    void getTimeOperation(){
        this.date = new Date();
    }
}
```

Data Access Object (Dao)



UserDao.java

```java
package com.example.demo.daos;

import com.example.demo.entities.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

/*Le interfacce del Dao Layer devono estendere l'interfaccia JpaRepository,
* che verrà implementata automaticamente da Spring all'avvio dell'applicazione.
* Questo ci eviterà di dover creare le implementazioni di UserDao, AccountDao
* e OperationDao, quindi non implementeremo delle classi UserDaoImpl,
* AccountDaoImpl e OperationDaoImpl. I due parametri dell'interfaccia JpaRepository
* sono l'Entity che ci interessa e il tipo del suo id.*/
public interface UserDao extends JpaRepository<User, String> {

    /*Di seguito abbiamo una Named Query, che serve per cercare uno,
    * o più record, in base alla colonna indicata nel nome della query
    * findByColonna */
    Optional<User> findById(String id);
}
```

AccountDao.java

```java
package com.example.demo.daos;

import com.example.demo.entities.Account;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

/*Le interfacce del Dao Layer devono estendere l'interfaccia JpaRepository,
 * che verrà implementata automaticamente da Spring all'avvio dell'applicazione.
 * Questo ci eviterà di dover creare le implementazioni di UserDao, AccountDao
 * e OperationDao, quindi non implementeremo delle classi UserDaoImpl,
 * AccountDaoImpl e OperationDaoImpl. I due parametri dell'interfaccia JpaRepository
 * sono l'Entity che ci interessa e il tipo del suo id.*/
public interface AccountDao extends JpaRepository<Account, String> {
    /*Query normale*/
    @Query(value = "SELECT * FROM accounts WHERE FK_USER = :user", nativeQuery = true)
    List<Account> getAllAccountsPerUser(@Param("user") String user);

    /*Named Query*/
    List<Account> findByFkUser(String fkUser);
}
```

OperationDao.java

```java
package com.example.demo.daos;

import com.example.demo.entities.Operation;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

/*Le interfacce del Dao Layer devono estendere l'interfaccia JpaRepository,
 * che verrà implementata automaticamente da Spring all'avvio dell'applicazione.
 * Questo ci eviterà di dover creare le implementazioni di UserDao, AccountDao
 * e OperationDao, quindi non implementeremo delle classi UserDaoImpl,
 * AccountDaoImpl e OperationDaoImpl. I due parametri dell'interfaccia JpaRepository
 * sono l'Entity che ci interessa e il tipo del suo id.*/
public interface OperationDao extends JpaRepository<Operation, String> {
    /*Query normale*/
    @Query(value = "SELECT * FROM operations WHERE FK_ACCOUNT1 = :account " +
            "OR FK_ACCOUNT2 = :account", nativeQuery = true)
    List<Operation> findAllOperationsByAccount(@Param("account") String account);
}
```

Riempiamo il database

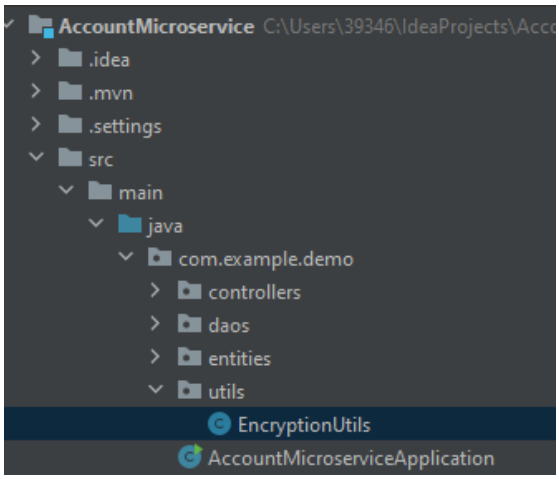
AccountMicroserviceApplication.java

```java
package com.example.demo;

import com.example.demo.daos.AccountDao;
import com.example.demo.daos.OperationDao;
import com.example.demo.daos.UserDao;
import com.example.demo.entities.Account;
import com.example.demo.entities.Operation;
import com.example.demo.entities.User;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import java.util.Date;

@SpringBootApplication
public class AccountMicroserviceApplication implements CommandLineRunner {

	/*Con @Autowired vado a iniettare, laddove serve, le classi implementate
	* nel progetto Spring. Nel nostro caso abbiamo di seguito delle interfacce,
	* ma Spring non inietterà ovviamente le interfacce, ma le classi che le
	* implementano, che sono appunto invisibili al programmatore,
	* perchè sono state implementate automaticamente da Spring*/
	@Autowired
	UserDao userDao;

	@Autowired
	AccountDao accountDao;

	@Autowired
	OperationDao operationDao;

	/* La classe Logger serve per loggare lo stato del sistema */
	private static final Logger log = LoggerFactory.getLogger(AccountMicroserviceApplication.class);

	public static void main(String[] args) {
		SpringApplication.run(AccountMicroserviceApplication.class, args);
	}

	/*Il metodo run(), ereditato dall'interfaccia CommandLineRunner, è il metodo
	* in cui possiamo implementare delle istruzioni che verranno eseguite prima
	* dell'avvio dell'applicazione*/
	@Override
	public void run(String... strings) throws Exception {
		//...
		log.info("Hello 1"); //Hello 1

		/*Il metodo save() è uno di quei metodi che appartengono all'interfaccia
		* JpaRepository, utilizzata per estendere le nostre 3 interfacce AccountDao,
		* OperationDao e UserDao. Con questo metodo possiamo riempire il nostro database H2,
		* che è un database in memoria. Un database in memoria è un tipo di database che
		* viene creato ogni qualvolta si avvia l'applicazione e viene distrutto quando
		* l'applicazione viene arrestata. Proprio per questo stiamo inserendo dei dati
		* nel metodo run(), perchè viene eseguito prima dell'avvio dell'applicazione,
		* per cui andrà a inserire i dati in fase di creazione del database H2*/

		userDao.save(new User("RGNLSN87H13D761R", "Alessandro Argentieri", "Abba", "user"));
		userDao.save(new User("FRNFBA85M08D761M", "Fabio Fiorenza", "melograno", "user"));
		userDao.save(new User("DSTLCU89R52D761R", "Lucia Distante", "salut", "user"));

		accountDao.save(new Account("cn4563df3", "RGNLSN87H13D761R", 3000.00));
		accountDao.save(new Account("cn7256su9", "RGNLSN87H13D761R", 4000.00));
		accountDao.save(new Account("cn6396dr7", "FRNFBA85M08D761M", 7000.00));
		accountDao.save(new Account("cn2759ds4", "DSTLCU89R52D761R", 2000.00));
		accountDao.save(new Account("cn2874da2", "DSTLCU89R52D761R", 8000.00));

		operationDao.save(new Operation("3452", new Date(),"Bonifico bancario", 100.00, "cn4563df3","cn4563df3"));
		operationDao.save(new Operation("3453", new Date(),"Pagamento tasse", -100.00, "cn4563df3","cn4563df3"));
		operationDao.save(new Operation("3454", new Date(),"Postagiro", 230.00, "cn4563df3","cn2759ds4"));
		operationDao.save(new Operation("3455", new Date(),"Vaglia postale", 172.00, "cn4563df3","cn4563df3"));
		operationDao.save(new Operation("3456", new Date(),"Acquisto azioni", -3400.00, "cn2759ds4",""));
		operationDao.save(new Operation("3457", new Date(),"Vendita azione", 100.00, "cn4563df3",""));
		operationDao.save(new Operation("3458", new Date(),"Prelevamento", -100.00, "cn4563df3",""));
		operationDao.save(new Operation("3459", new Date(),"Deposito", 1100.00, "cn4563df3",""));
		operationDao.save(new Operation("3460", new Date(),"Bonifico bancario", 100.00, "cn2874da2","cn4563df3"));
		operationDao.save(new Operation("3461", new Date(),"Bonifico bancario", 100.00, "cn4563df3","cn2874da2"));
		operationDao.save(new Operation("3462", new Date(),"Bonifico bancario", 100.00, "cn4563df3","cn4563df3"));
		operationDao.save(new Operation("3463", new Date(),"Postagiro", 230.00, "cn7256su9","cn2759ds4"));
		operationDao.save(new Operation("3464", new Date(),"Vaglia postale", 172.00, "cn4563df3","cn7256su9"));
		operationDao.save(new Operation("3465", new Date(),"Acquisto azioni", -3400.00, "cn7256su9",""));
	}
}
```

Criptiamo le password

pom.xml

```
<dependencies>
	...
	<dependency>
		<groupId>com.github.ulisesbocchio</groupId>
		<artifactId>jasypt-spring-boot-starter</artifactId>
		<version>3.0.3</version>
	</dependency>
</dependencies>
```



EncryptionUtils.java

```
package com.example.demo.utils;

import org.jasypt.util.text.BasicTextEncryptor;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@Component
public class EncryptionUtils {

    /*Il metodo textEncryptor() serve per istanziare un BasicTextEncryptor
     * per settare un chiave di criptaggio sulle password di tutti gli UserDao*/
    @Bean
    public BasicTextEncryptor textEncryptor(){
        BasicTextEncryptor textEncryptor = new BasicTextEncryptor();
        textEncryptor.setPassword("mySecretEncryptionKeyBlaBla1234");
        return textEncryptor;
    }

    /*Il metodo encrypt() provvede a criptare una stringa passata in ingresso*/
    public String encrypt(String data){
        return textEncryptor().encrypt(data);
    }

    /*Il metodo encrypt() provvede a criptare una stringa passata in ingresso*/
    public String decrypt(String encryptedData){
        return textEncryptor().decrypt(encryptedData);
    }
}
```

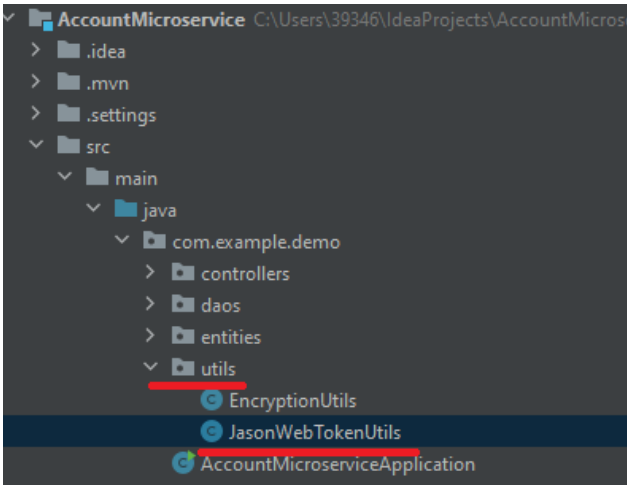
AccountMicroserviceApplication.java

```java
package com.example.demo;

import com.example.demo.daos.AccountDao;
import com.example.demo.daos.OperationDao;
import com.example.demo.daos.UserDao;
import com.example.demo.entities.Account;
import com.example.demo.entities.Operation;
import com.example.demo.entities.User;
import com.example.demo.utils.EncryptionUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.Date;


@SpringBootApplication
public class AccountMicroserviceApplication implements CommandLineRunner {

	...

	@Autowired
	EncryptionUtils encryptionUtils;

	...

	/*Il metodo run(), ereditato dall'interfaccia CommandLineRunner, è il metodo
	* in cui possiamo implementare delle istruzioni che verranno eseguite prima
	* dell'avvio dell'applicazione*/
	@Override
	public void run(String... strings) throws Exception {

		...

		/*UserDao con password non criptata*/
		//userDao.save(new User("RGNLSN87H13D761R", "Alessandro Argentieri", "Abba", "user"));
		//userDao.save(new User("FRNFBA85M08D761M", "Fabio Fiorenza", "melograno", "user"));
		//userDao.save(new User("DSTLCU89R52D761R", "Lucia Distante", "salut", "user"));

		/*UserDao con password criptata*/
		String encryptedPwd = encryptionUtils.encrypt("Abba");
		userDao.save(new User("RGNLSN87H13D761R", "Alessandro Argentieri", encryptedPwd, "user"));

		encryptedPwd = encryptionUtils.encrypt("melograno");
		userDao.save(new User("FRNFBA85M08D761M", "Fabio Fiorenza", encryptedPwd, "user"));

		encryptedPwd = encryptionUtils.encrypt("salut");
		userDao.save(new User("DSTLCU89R52D761R", "Lucia Distante", encryptedPwd, "user"));

		...
	}
}
```

Json Web Token

pom.xml

```
<dependencies>
	...
	<dependency>
		<groupId>io.jsonwebtoken</groupId>
		<artifactId>jjwt</artifactId>
		<version>0.7.0</version>
	</dependency>
</dependencies>
```



JasonWebTokenUtils.java

```
package com.example.demo.utils;

import io.jsonwebtoken.*;
import org.springframework.stereotype.Component;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

/**
 * Questa classe fornisce un metodo per generare e convalidare i JSON token web e
 * verrà automaticamente istanziata e iniettata da Spring
 */
public class JasonWebTokenUtils {
    /**
     * Il metodo generateJwt() genera il JSON web token da inviare al client
     */
    public static String generateJwt(String subject, Date date, String name, String scope) throws
java.io.UnsupportedEncodingException{

        String jwt = Jwts.builder()
                .setSubject(subject)
                .setExpiration(date)
                .claim("name", name)
                .claim("scope", scope)
                .signWith(
                        SignatureAlgorithm.HS256,
                        "myPersonalSecretKey12345".getBytes("UTF-8")
                )
                .compact();

        return jwt;
```

```
    }

    /**
     * Il metodo jwt2Map() converte il token in una mappa di Userdata, verificandone la validità
     */
    public static Map<String, Object> jwt2Map(String jwt) throws java.io.UnsupportedEncodingException,
ExpiredJwtException{

        Jws<Claims> claim = Jwts.parser()
                .setSigningKey("myPersonalSecretKey12345".getBytes("UTF-8"))
                .parseClaimsJws(jwt);

        String name = claim.getBody().get("name", String.class);
        String scope = (String) claim.getBody().get("scope");

        Date expDate = claim.getBody().getExpiration();
        String subj = claim.getBody().getSubject();

        Map<String, Object> userData = new HashMap<>();
        userData.put("name", name);
        userData.put("scope", scope);
        userData.put("exp_date", expDate);
        userData.put("subject", subj);

        Date now = new Date();
        if(now.after(expDate)){
            throw new ExpiredJwtException(null, null, "Session expired!");
        }

        return userData;
    }


    /**
     * Il metodo getJwtFromHttpRequest estrae il token dall'intestazione
     * o dal cookie nella richiesta Http
     */
    public static String getJwtFromHttpRequest(HttpServletRequest request){
        String jwt = null;
        if(request.getHeader("jwt") != null){
            jwt = request.getHeader("jwt");     //token presente nell'header
        }else if(request.getCookies() != null){
            Cookie[] cookies = request.getCookies();   //token presente nel cookie
            for (Cookie cookie : cookies) {
                if (cookie.getName().equals("jwt")) {
                    jwt = cookie.getValue();
                }
            }
        }
        return jwt;
    }

}
```

Classe JsonResponseBody

RestController.java

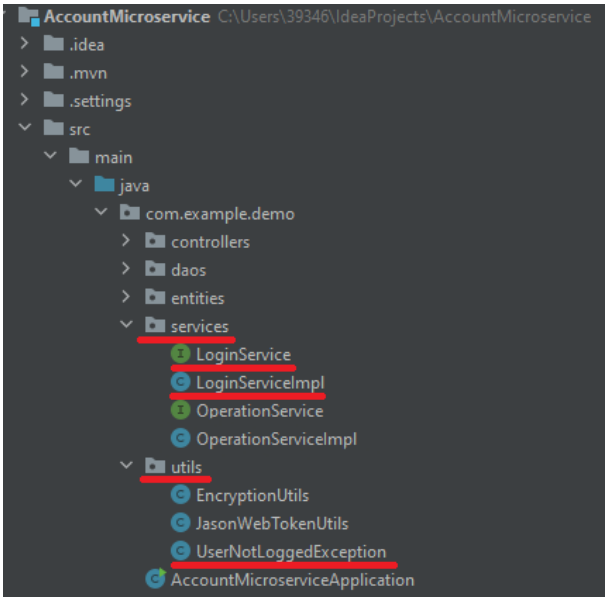
```java
package com.example.demo.controllers;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/* Il RestController è una classe di Spring i cui metodi si occupano
 di gestire tutte le richieste http */
@org.springframework.web.bind.annotation.RestController
public class RestController {

    ...

    /*L'Inner class JsonResponseBody() è utilizzata come oggetto
    * legato al body della ResponseEntity.
    * È importante avere questo oggetto perché è composto dal codice
    * di risposta del server e dall'oggetto di risposta.
    * Poi la libreria Jackson converte automaticamente questo
    * oggetto JsonResponseBody in una risposta JSON.*/
    @AllArgsConstructor
    public class JsonResponseBody{
        @Getter @Setter
        private int server;

        @Getter @Setter
        private Object response;

        /*Esempio JSON response:
        {
            server: 500,
            response: {...}
        }*/
    }
}
```

Strutturiamo il controller

RestController.java

```java
package com.example.demo.controllers;

import com.example.demo.entities.Operation;
import jakarta.servlet.http.HttpServletRequest;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

/* Il RestController è una classe di Spring i cui metodi si occupano
 di gestire tutte le richieste http */
@org.springframework.web.bind.annotation.RestController
public class RestController {

    ...

    @PostMapping("/login")
    public ResponseEntity<JsonResponseBody> loginUser(@RequestParam(value ="id") String id,
                                                      @RequestParam(value="password") String pwd) {
        //verifica se l'utente esiste nel DB -> se esiste, genera il JSON web token e lo invia al client
        return null;
    }

    @PostMapping("/operations/account/{account}")
    public ResponseEntity<JsonResponseBody> fetchAllOperationsPerAccount(HttpServletRequest request,
                                                                         @PathVariable(name = "account") String account) {
        //request -> recupero del JSON web token -> verifica della validità del token
        // -> ottenimento delle operazioni da parte dell'account utente
        return null;
    }

    @PostMapping("/accounts/user")
    public ResponseEntity<JsonResponseBody> fetchAllAccountsPerUser(HttpServletRequest request) {
        //request -> recupero del JSON web token -> recupero dei dati dell'utente
        //ottenimento di tutti gli account dell'utente dal DB
        return null;
    }

    @PostMapping("/operations/add")
    public ResponseEntity<JsonResponseBody> addOperation(HttpServletRequest request,
                                                         Operation operation,
                                                         BindingResult bindingResult) {
        //request -> recupero del JSON web token -> recupero dei dati dell'utente ->
        //salvataggio delle operazioni nel DB
        return null;
    }
}
```

Implementazione LoginService



UserNotLoggedException.java

```java
package com.example.demo.utils;

public class UserNotLoggedException extends Exception{

    public UserNotLoggedException(String errorMessage){
        super(errorMessage);
    }
}
```

LoginService.java

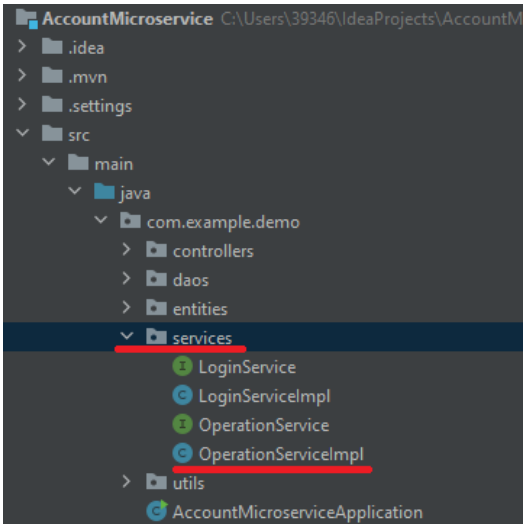
```java
package com.example.demo.services;

import com.example.demo.entities.User;
import com.example.demo.utils.UserNotLoggedException;

import javax.servlet.http.HttpServletRequest;
import java.io.UnsupportedEncodingException;
import java.util.Date;
import java.util.Map;
import java.util.Optional;

public interface LoginService {

    Optional<User> getUserFromDbAndVerifyPassword(String id, String password) throws UserNotLoggedException;
    //-> userDao, findById(id), encryptionUtils.decrypt(password)
    //-> UserNotLoggedException

    String createJwt(String subject, String name, String permission, Date date, Date datenow) throws
UnsupportedEncodingException;
    //-> JasonWebTokenUtils.generateJwt(...) -> UnsupportedEncodingException

    Map<String, Object> verifyJwtAndGetData(HttpServletRequest request) throws UserNotLoggedException,
UnsupportedEncodingException;
    //-> JasonWebTokenUtils.getJwtFromHttpRequest(request) -> UserNotLoggedException
    //-> JasonWebTokenUtils.jwt2Map(jwt) -> UnsupportedEncodingException
    //-> ExpiredJwtException
}
```

LoginServiceImpl.java

```java
package com.example.demo.services;

import com.example.demo.daos.UserDao;
import com.example.demo.entities.User;
import com.example.demo.utils.EncryptionUtils;
import com.example.demo.utils.JasonWebTokenUtils;
import com.example.demo.utils.UserNotLoggedException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.servlet.http.HttpServletRequest;
import java.io.UnsupportedEncodingException;
import java.util.Date;
import java.util.Map;
import java.util.Optional;

/**
 * Servizio di Login
 */
@Service
public class LoginServiceImpl implements LoginService{

    private static final Logger log = LoggerFactory.getLogger(LoginServiceImpl.class);

    @Autowired
    UserDao userDao;

    @Autowired
    EncryptionUtils encryptionUtils;

    /**
     * Il metodo getUserFromDbAndVerifyPassword() verifica che l'utente sia effettivamente
     * presente nel database, sia trovandone l'id, sia verificando la password.
     *
     * @param id
     * @param password
     * @return
     * @throws UserNotLoggedException
     */
    @Override
    public Optional<User> getUserFromDbAndVerifyPassword(String id, String password) throws UserNotLoggedException {
        //-> userDao, findById(id), encryptionUtils.decrypt(password)
        //-> UserNotLoggedException

        Optional<User> userr = userDao.findById(id);
        if(userr.isPresent()){
            User user = userr.get();
            if(encryptionUtils.decrypt(user.getPassword()).equals(password)) {
                log.info("Username and Password verified");
            }else{
                log.info("Username verified. Password not");
                throw new UserNotLoggedException("User not correctly logged in");
            }
        }

        return userr;
    }

    /**
     * Qualora l'utente esista nel database, il metodo createJwt() crea un nuovo Jason Web Token
     * alla prima richiesta http, utilizzando i dati dell'utente stesso recuperati dal metodo
     * getUserFromDbAndVerifyPassword()
     *
     * @param subject
     * @param name
     * @param permission
     * @param date
     * @param datenow
     * @return
     * @throws UnsupportedEncodingException
     */
    @Override
    public String createJwt(String subject, String name, String permission, Date date, Date datenow) throws
UnsupportedEncodingException {
        //-> JasonWebTokenUtils.generateJwt(...) -> UnsupportedEncodingException

        Date expDate = datenow;

        //Imposto un periodo di validità del token
        expDate.setTime(datenow.getTime() + (300+1000));
        log.info("Jason Web Token creation. Expiration time: " + expDate.getTime());

        String token = JasonWebTokenUtils.generateJwt(subject, expDate, name, permission);
```

```
        return token;
    }

    /**
     * IL metodo verifyJwtAndGetData() recupera il Jason Web Token e, se è valido,
     * vengono recuperati i dati dell'utente da utilizzare per successive operazioni.
     *
     * @param request
     * @return
     * @throws UserNotLoggedException
     * @throws UnsupportedEncodingException
     */
    @Override
    public Map<String, Object> verifyJwtAndGetData(HttpServletRequest request) throws UserNotLoggedException,
UnsupportedEncodingException {
        //-> JasonWebTokenUtils.getJwtFromHttpRequest(request) -> UserNotLoggedException
        //-> JasonWebTokenUtils.jwt2Map(jwt) -> UnsupportedEncodingException
        //-> ExpiredJwtException

        //Recupero il Jason Web Token e, se ancora valido, permetto una serie di operazioni
        String token = JasonWebTokenUtils.getJwtFromHttpRequest(request);
        if(token == null){
            throw new UserNotLoggedException("Authentication token not found in the request");
        }

        Map<String, Object> userData = JasonWebTokenUtils.jwt2Map(token);
        return userData;
    }
}
```

Implementazione OperationService



OperationServiceImpl.java

```java
package com.example.demo.services;

import com.example.demo.daos.AccountDao;
import com.example.demo.daos.OperationDao;
import com.example.demo.entities.Account;
import com.example.demo.entities.Operation;
import jakarta.transaction.Transactional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Date;
import java.util.List;

/*Oltre a @Service, possiamo notare l'aggiunta di @Transactional.
* @Transactional non è da utilizzare obbligatoriamente, ma è consigliato
* il suo utilizzo quando il @Service deve eseguire un accesso massivo
* ai database attraverso i dao. Utilizzando @Transactional permettiamo a
* Spring di automatizzare il processo delle transazioni e gestirle.
*/
@Service @Transactional
public class OperationServiceImpl implements OperationService{

    @Autowired
    AccountDao accountDao;

    @Autowired
    OperationDao operationDao;

    @Override
    public List<Operation> getAllOperationPerAccount(String accountId) {
        return operationDao.findAllOperationsByAccount(accountId);
    }

    @Override
    public List<Account> getAllAccountsPerUser(String userId) {
        return accountDao.getAllAccountsPerUser(userId);
    }

    @Override
    public Operation saveOperation(Operation operation) {

        if(operation.getDate() == null){
            operation.setDate(new Date());
        }

        return operationDao.save(operation);
    }
}
```

Implementazione RestController

RestController.java

```java
package com.example.demo.controllers;

import com.example.demo.entities.Operation;
import com.example.demo.entities.User;
import com.example.demo.services.LoginService;
import com.example.demo.services.OperationService;
import com.example.demo.utils.UserNotLoggedException;;
import io.jsonwebtoken.ExpiredJwtException;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import jakarta.servlet.http.HttpServletRequest;
import java.io.UnsupportedEncodingException;
import java.util.Date;
import java.util.Map;
import java.util.Optional;

/* Il RestController è una classe di Spring i cui metodi si occupano
 di gestire tutte le richieste http */
@org.springframework.web.bind.annotation.RestController
public class RestController {

    private static final Logger log = LoggerFactory.getLogger(RestController.class);

    ...

    @PostMapping("/login")
    public ResponseEntity<JsonResponseBody> loginUser(@RequestParam(value ="id") String id,
                                                      @RequestParam(value="password") String pwd) {
        //verifica se l'utente esiste nel DB -> se esiste, genera il JSON web token e lo invia al client
        try {
            Optional<User> userr = loginService.getUserFromDbAndVerifyPassword(id, pwd);
            if(userr.isPresent()){
                User user = userr.get();
                String jwt = loginService.createJwt(user.getId(), user.getUsername(), user.getPermission(), new Date());
                return ResponseEntity.status(HttpStatus.OK).header("jwt", jwt).body(new JsonResponseBody(HttpStatus.OK.value(),
                                                                                    "Success! User logged in!"));
            }
        } catch (UserNotLoggedException e1) {
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(),
                                                    "Login failed! Wrong credentials " + e1.toString()));
        } catch (UnsupportedEncodingException e2) {
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(),
                                                    "Token Error " + e2.toString()));
        }
        return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(),
                                                "No corrispondence in the database"));
    }

    @PostMapping("/operations/account/{account}")
    public ResponseEntity<JsonResponseBody> fetchAllOperationsPerAccount(HttpServletRequest request,
                                                                         @PathVariable(name = "account") String account) {
        //request -> recupero del JSON web token -> verifica della validità del token
        // -> ottenimento delle operazioni da parte dell'account utente
        try {
            loginService.verifyJwtAndGetData(request);
            //utente verificato
            return ResponseEntity.status(HttpStatus.OK).body(new JsonResponseBody(HttpStatus.OK.value(),
                                                                operationService.getAllOperationPerAccount(account)));
        } catch (UserNotLoggedException e1) {
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(),
                                                                "Unsupported Encoding: " + e1.toString()));
        } catch (UnsupportedEncodingException e2) {
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(),
                                                                "User not correctly logged encoding: " + e2.toString()));
        } catch(ExpiredJwtException e3) {
            //il Jason Web Token è scaduto
            return ResponseEntity.status(HttpStatus.GATEWAY_TIMEOUT).body(new JsonResponseBody(HttpStatus.GATEWAY_TIMEOUT.value(),
                                                                    "Session Expired!:" + e3.toString()));
        }
    }

    @PostMapping("/accounts/user")
    public ResponseEntity<JsonResponseBody> fetchAllAccountsPerUser(HttpServletRequest request) {
        //request -> recupero del JSON web token -> recupero dei dati dell'utente
        //ottenimento di tutti gli account dell'utente dal DB
        try {
            Map<String, Object> userData = loginService.verifyJwtAndGetData(request);
            return ResponseEntity.status(HttpStatus.OK).body(new JsonResponseBody(HttpStatus.OK.value(),
                                                    operationService.getAllAccountsPerUser((String) userData.get("subject"))));
        } catch (UserNotLoggedException e1) {
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(),
                                                                "User not logged! Login first: " + e1.toString()));
        } catch (UnsupportedEncodingException e2) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(new JsonResponseBody(HttpStatus.BAD_REQUEST.value(),
```

```
                                                                                "Bad Request: " + e2.toString()));
        } catch(ExpiredJwtException e3) {
            //il Jason Web Token è scaduto
            return ResponseEntity.status(HttpStatus.GATEWAY_TIMEOUT).body(new JsonResponseBody(HttpStatus.GATEWAY_TIMEOUT.value(),
                                                                                    "Session Expired!:" + e3.toString()));
        }
    }

    @PostMapping("/operations/add")
    public ResponseEntity<JsonResponseBody> addOperation(HttpServletRequest request,
                                                         Operation operation,
                                                         BindingResult bindingResult) {
        //request -> recupero del JSON web token -> recupero dei dati dell'utente ->
        //salvataggio delle operazioni nel DB
        if(bindingResult.hasErrors()){
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(),
                                                                                "Error! Invalid format of data"));
        }
        try {
            loginService.verifyJwtAndGetData(request);
            return ResponseEntity.status(HttpStatus.OK).body(new JsonResponseBody(HttpStatus.OK.value(),
operationService.saveOperation(operation)));
        } catch (UserNotLoggedException e1) {
            return ResponseEntity.status(HttpStatus.FORBIDDEN).body(new JsonResponseBody(HttpStatus.FORBIDDEN.value(),
                                                                                "User not logged! Login first: " + e1.toString()));
        } catch (UnsupportedEncodingException e2) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(new JsonResponseBody(HttpStatus.BAD_REQUEST.value(),
                                                                                "Bad Request: " + e2.toString()));
        } catch(ExpiredJwtException e3) {
            //il Jason Web Token è scaduto
            return ResponseEntity.status(HttpStatus.GATEWAY_TIMEOUT).body(new JsonResponseBody(HttpStatus.GATEWAY_TIMEOUT.value(),
                                                                                    "Session Expired!:" + e3.toString()));
        }
    }
}
```