

Aggiornamento dei Sistemi Gestionali per un'azienda di
servizi Sartoriali



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Renild Gega & Lorenzo Arnetoli

February 22, 2024

Contents

1	Introduzione	2
2	Progettazione	3
2.1	Use Case Diagram	3
2.2	Use Case Templates	4
2.2.1	Casi d'uso principali di un cliente	4
2.2.2	Casi d'uso principali dell'Admin	5
2.2.3	SuperUser	6
2.3	Class Diagram	6
2.4	DAO and DTO pattern	7
2.5	Modello ER	7
2.6	Mockups	9
2.6.1	Punto di vista di un Cliente	9
2.6.2	Punto di vista dell'Admin	10
2.6.3	punto di vista del SuperUser	12
2.7	Sicurezza , Autenticazione ed Autorizzazione	12
3	Implementazioni delle classi	13
3.1	Domain Model	13
3.1.1	Account	13
3.1.2	Costumer	13
3.1.3	Job	13
3.1.4	Order	14
3.1.5	Role	14
3.1.6	UserEntity	14
3.2	Controller	14
3.2.1	CostumerController	14
3.2.2	JobController	15
3.2.3	OrderController	15
3.2.4	AuthController	15
3.3	Repository	16
3.4	Service	16
3.5	DTO	16
3.6	Security	17
3.7	MailService	17
4	Database	18
4.1	DBMS: MySQL	18
4.2	Connessione al DBMS	18
4.3	Mappatura del Domain Model	18
5	Test	20
6	Deployment	22
7	Sviluppi Futuri	23

Chapter 1

Introduzione

La seguente relazione documenta un tentativo di risoluzione del problema concreto dell'aggiornamento degli impianti informatici per una piccola azienda di Firenze. In particolare, l'azienda - specializzata in servizi sartoriali, come la realizzazione di abiti su misura e riparazioni varie - ha identificato la necessità di aggiornare i propri sistemi gestionali per migliorare l'efficienza e offrire nuove funzionalità ai clienti.

Il seguente paragrafo riassume gli obiettivi in termini di requisiti funzionali.

Gestione degli Ordini: Gli ordini dei clienti devono includere informazioni dettagliate come i dati personali (nome, cognome, email, telefono), l'elenco dei lavori richiesti, i relativi prezzi unitari, il totale dell'ordine, la data di ritiro, eventuali sconti, lo stato di pagamento (pagato in consegna o al ritiro) e lo stato dell'ordine. Comunicazioni Automatiche: Quando lo stato dell'ordine viene impostato a "completato", il sistema deve inviare automaticamente una email al cliente per informarlo che l'ordine è pronto per il ritiro anticipato, se possibile.

Gestione dei Clienti: L'azienda deve mantenere un database dei clienti per facilitare ordini futuri e offrire un servizio personalizzato.

Integrazione con il Sito Web: L'azienda possiede un sito web statico, gestito da un team di designer esterni. Dopo l'aggiornamento dei sistemi gestionali, l'azienda desidera continuare a collaborare con i designer per migliorare l'esperienza utente sul sito e sulle interfacce delle applicazioni gestionali.

Interazione Cliente: Il nuovo sito web deve offrire ai clienti maggiori possibilità di interagire con i propri ordini, incluso la creazione di un account utente per gestire gli ordini, accumulare punti fedeltà e ricevere sconti e vantaggi personalizzati.

Il programma prevede l'utilizzo di SpringBoot, con SpringDataJPA per la gestione dei dati, ed espone degli endpoint REST per interagire tramite interfacce grafiche, il cui sviluppo è delegato a terzi.

Chapter 2

Progettazione

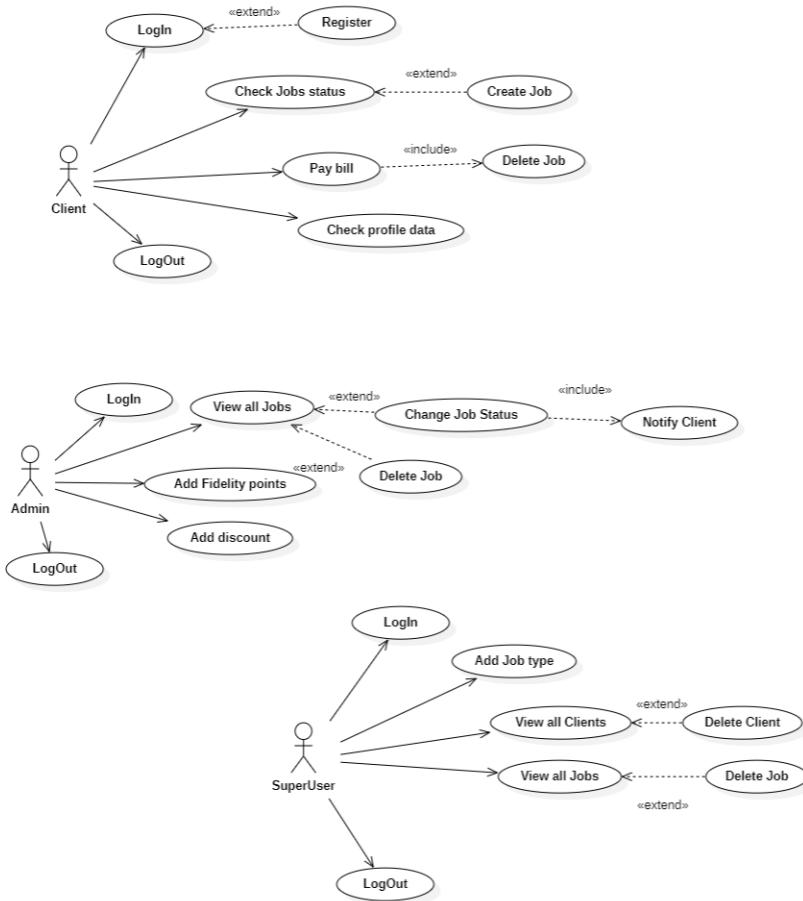
Il software ha 3 diversi attori: il cliente (Customer), il negoziante (Admin) e il proprietario (SuperUser).

- **Il cliente** può controllare gli stati di tutti i capi portati in negozio e prenotare un lavoro da fare su un capo che porterà in un secondo momento in negozio;
- **l'Admin** ovvero il negoziante, può aggiungere nuove richieste di lavori, o aggiornare quelle già esistenti, e registrare il cliente che ha fatto le varie richieste se non è già registrato;
- **Il SuperUser** si occupa della gestione delle varie tipologie di lavori realizzabili e degli account dei clienti registrati.

Lo schema dei casi d'uso, è realizzato secondo lo standard UML mediante StarUML.

2.1 Use Case Diagram

Schema Use Case creato con StarUML



2.2 Use Case Templates

In questo paragrafo illustriamo dei casi d'uso che pensiamo siano i più importanti.

2.2.1 Casi d'uso principali di un cliente

UC-1	Register
Descrizione	Il cliente si può registrare sul sito del negozio inserendo username ed e-mail per ricevere dei servizi
Livello	User Goal
Attore principale	Client
Azioni	<ol style="list-style-type: none"> 1. Il cliente accede alla home-page del sito dove troverà e cliccherà il tasto "REGISTRATI" 2. Il sistema visualizzerà una schermata dove chiederà i dati necessari alla registrazione 3. Il cliente inserirà, negli appositi spazi, i dati richiesti 4. Il cliente cliccherà il tasto "REGISTRA" 5. Il sistema visualizzerà, in un pop-up temporaneo, il messaggio "registrazione riuscita" e tornerà alla schermata iniziale
Casi speciali	5. in caso non si fossero inseriti i dati necessari il sistema visualizzerà un pop-up temporaneo con un messaggio dove indica i campi senza dati e rimarrà nella stessa schermata

Table 2.1: Registrazione di un account da parte di un cliente

UC-2	Check Jobs status
Descrizione	Il cliente, tramite il proprio account, controlla lo stato dei lavori richiesti
Livello	User Goal
Attore principale	Client
Azioni	<ol style="list-style-type: none"> 1. Il cliente accede al proprio account tramite la funzione di LogIn 2. Il sistema mostra una tabella con i lavori associati al cliente
Casi speciali	2. In caso il cliente non abbia lavori richiesti in attesa di lavorazione o in attesa di pagamento, verrà mostrato un messaggio di assenza di lavori

Table 2.2: Consultazione dello stato dei lavori richiesti al sarto

2.2.2 Casi d'uso principali dell'Admin

UC-3	View all Jobs
Descrizione	Il negoziante controlla quanti e quali sono i lavori da fare
Livello	User Goal
Attore principale	Admin
Azioni	<ol style="list-style-type: none"> 1. Il negoziante clicca sul tasto dell'applicazione "vedi lavori" 2. l'applicazione mostra la lista di lavori attualmente salvati nel sistema

Table 2.3: Consultazione lavori (Order) da parte del negoziante

UC-4	Change Job Status
Descrizione	Il negoziante cambia lo stato di un lavoro da "pending" a "done"
Livello	User Goal
Attore principale	Admin
Azioni	<ol style="list-style-type: none"> 1. il negoziante clicca sul tasto "cerca lavoro tramite id" 2. l'applicazione mostra una finestra con tutti i dati del lavoro e un tasto con scritto "LAVORO FINITO" 3. il negoziante clicca sul tasto "LAVORO FINITO" 4. il sistema cambia lo stato del lavoro da "pending" a "finito" 5. il sistema manda una notifica al cliente associato al lavoro

Table 2.4: Cambio di stato del lavoro (Order) una volta finito

UC-5	Create Job
Descrizione	Il negoziante crea un nuovo lavoro da fare e lo collega ad una mail e/o numero di telefono del cliente associato
Livello	User Goal
Attore principale	Admin
Azioni	<ol style="list-style-type: none"> 1. Il negoziante clicca, dalla schermata iniziale dell'applicazione, il tasto "CREA NUOVO LAVORO" 2. L'applicazione apre una nuova finestra dove chiede l'inserimento di tutti i dati necessari per salvare il lavoro 3. Il negoziante inserisce tutti i dati richiesti 4. Il negoziante clicca sul tasto "INSERISCI" 5. L'applicazione salva i dati e mostra a schermo una finestrella a pop-up con scritto il risultato dell'inserimento e un tasto "OK" per chiudere la finestra
Casi speciali	<ol style="list-style-type: none"> 5. in caso non venga inserito un dato necessario, la finestra mostrerà i nomi dei dati mancanti e una volta chiusa si torna al punto 3

Table 2.5: Creazione di un nuovo lavoro (Order) da parte del negoziante

2.2.3 SuperUser

UC-6	Add new Job type
Descrizione	Il SuperUser aggiunge una nuova tipologia di lavoro che può essere eseguita da un Admin
Livello	User Goal
Attore principale	SuperUser
Azioni	<ol style="list-style-type: none"> 1. Colui che ha l'accesso da SuperUser effettua il LogIn nell'applicazione vista anche da Admin, con le credenziali SuperUser 2. L'applicazione mostrerà ora una schermata uguale a quella principale da Admin, ma con un tasto aggiuntivo "AGGIUNGI TIPOLOGIA LAVORO" 3. Il SuperUser clicca sul tasto sopra menzionato 4. L'applicazione mostrerà una finestra dove chiederà tutti i campi da riempire per il salvataggio del nuovo tipo di lavoro 5. Il SuperUser inserisce i dati richiesti 6. Il SuperUser clicca sul tasto "SALVA" 7. L'applicazione mostra un pop-up con il messaggio "lavoro inserito con successo" e un tasto "OK" per chiudere 8. L'applicazione torna alla schermata iniziale
Casi speciali	<ol style="list-style-type: none"> 7. In caso il SuperUser non avesse inserito tutti i dati necessari, L'applicazione mostra una finestra pop-up con scritto i nomi dei campi di cui mancano i dati e un tasto "OK" che riporta alla schermata di inserimento dati

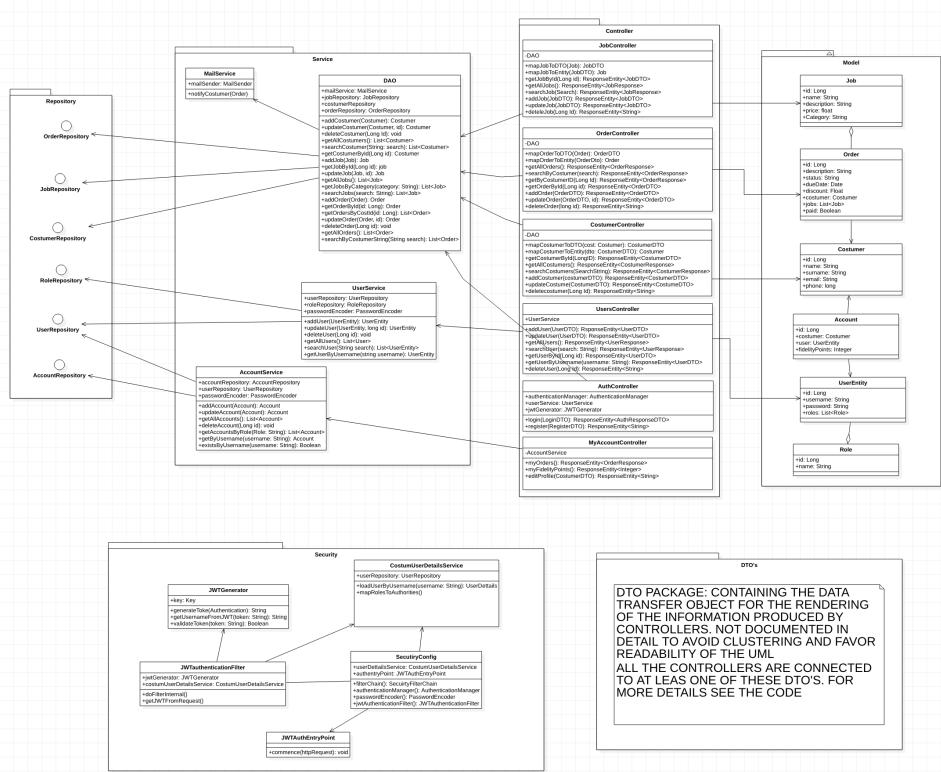
Table 2.6: Aggiunta di una tipologia di lavoro (Job) da parte del SuperUser

2.3 Class Diagram

Le classi sono state divise in 6 package differenti:

1. **Repository:** questo package contiene tutte le classi che interagiscono direttamente con il Database;
2. **Model:** questo package contiene le classi del Domain model, ovvero le classi che definiscono le varie entità che verranno salvate nel Database;
3. **DTO:** questo package contiene le classi usate per salvare i dati estratti dal database e manipolarli senza rischio di perdita di dati;
4. **Controller:**
5. **Service:**
6. **Security:** questo package contiene classi che si occupano delle autorizzazioni di accesso alle varie parti dell'applicazione, in base al ruolo mostreranno o meno alcune finestre e funzionalità.

Pagina di registrazione nuovo account Cliente



2.4 DAO and DTO pattern

DAO (Data Access Object) è un pattern molto diffuso per la gestione degli oggetti persistenti. Il DAO costituisce un layer di accesso ai dati e risulta molto comodo in applicazioni gestionali.

L'acronimo **DTO** invece sta per **Data Transfer Object** e rappresenta il nome di oggetti usati per trasportare i dati tra i vari processi, evitando di esporre direttamente le classi, e costruendo magari rappresentazioni diversamente strutturate dell'informazione contenuta nelle classi - simile a view in MVC.

Un altro vantaggio presentato dai DTO è quello di encapsulare la logica di serializzazione, ovvero il meccanismo che traduce la struttura dati in uno specifico formato che poi viene trasferito o salvato.

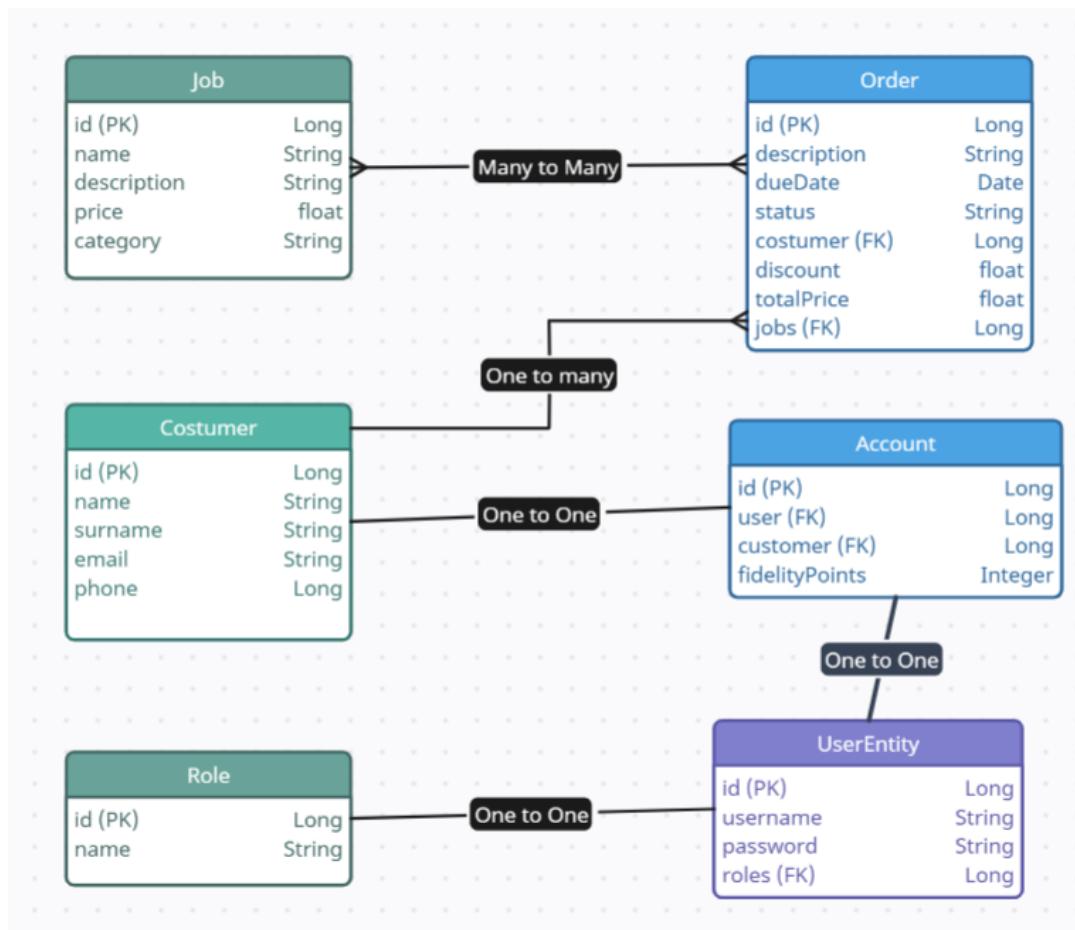
Le classi che sfruttano questo design pattern sono quelle definite nel Domain model e sono:

- Customer
- Job
- Order
- Role
- User

in quanto sono queste le classi salvate nel database; il pattern viene implementato attraverso le classi Service.

2.5 Modello ER

Includiamo un modello Entity–Relationship del progetto di seguito.



2.6 Mockups

Ecco alcuni Mockups di come abbiamo pensato possa essere una possibile GUI

2.6.1 Punto di vista di un Cliente

Pagina di registrazione nuovo account Cliente

CREATE ACCOUNT

USERNAME	BillyJones_
PASSWORD	*****
NAME	Billy
SURNAME	Jones
EMAIL	billy_jones@pistolmail.com
PHONE	+1 0000007777

SIGN UP

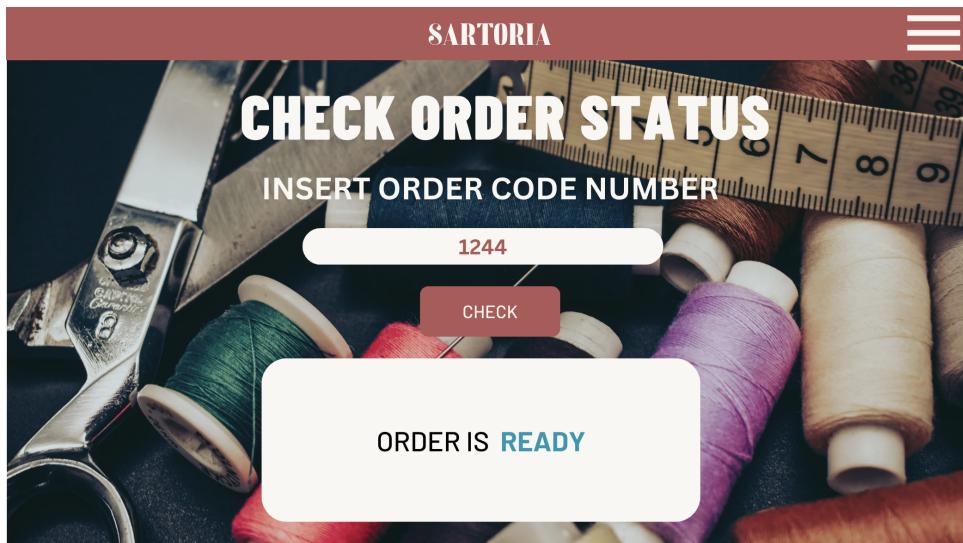
Pagina per la visualizzazione di tutti gli ordini

MY ORDERS

ORDER NUMBER	DATE	STATE	DESCR.	JOBS
1	11/10/2025	PENDING	SLEEVES TOO LONG	GET JOBS
2	01/01/2024	READY	BROKEN ZYP	GET JOBS
3	14/02/2023	SERVER	NEW JACKET	GET JOBS

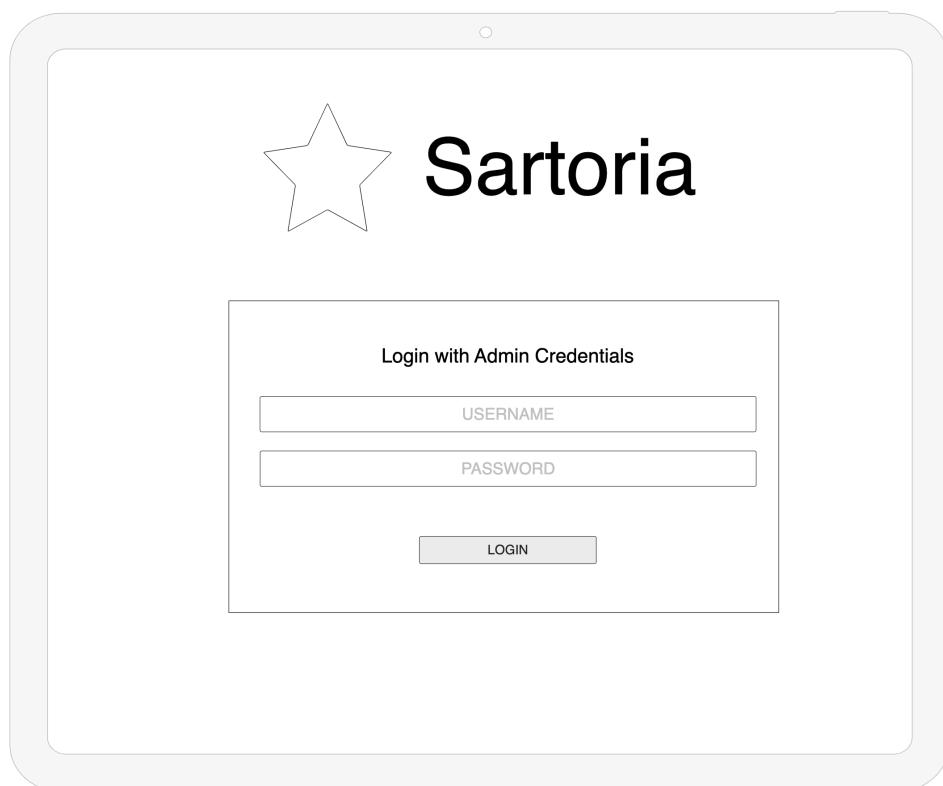
1 2 ... 10

Pagina per la vista dello status di un lavoro

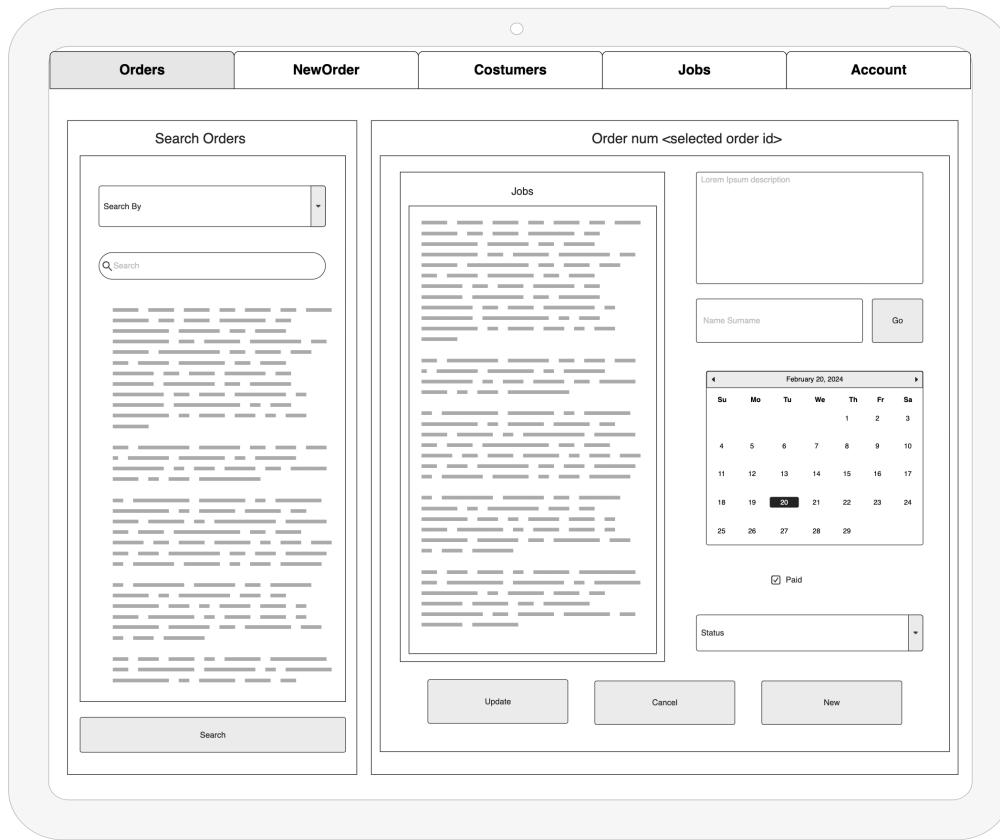


2.6.2 Punto di vista dell'Admin

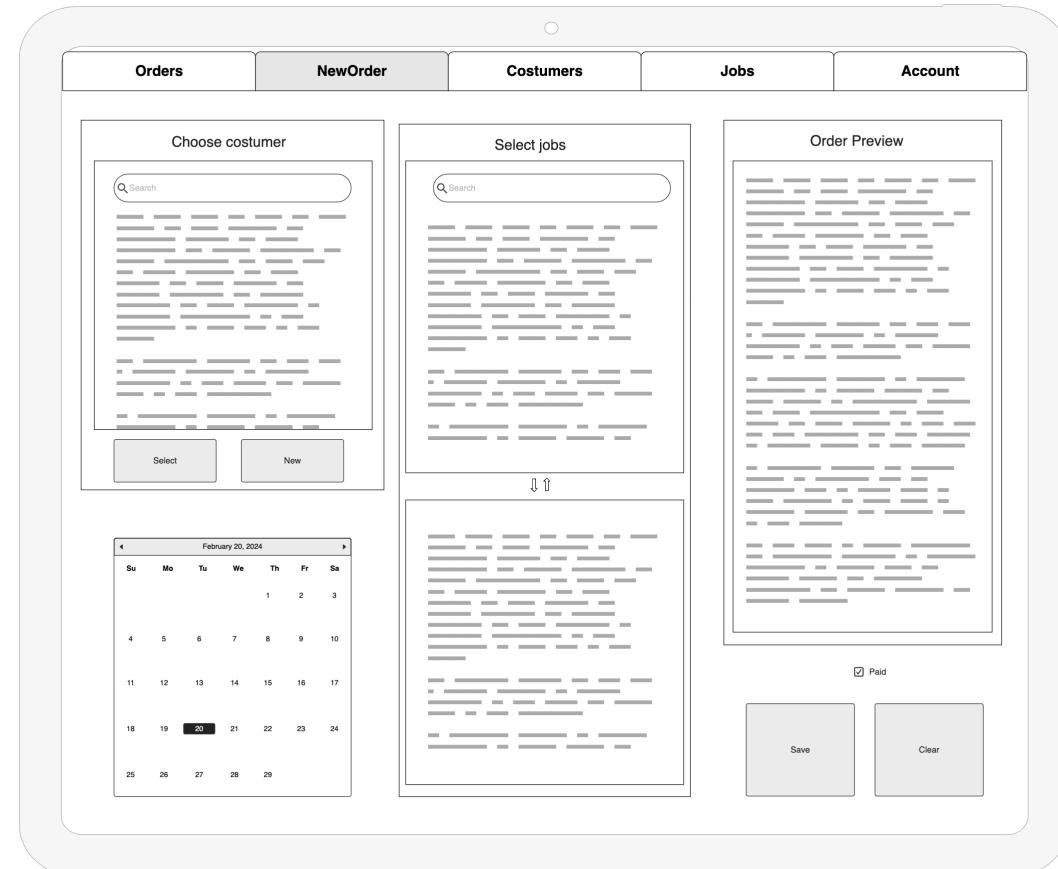
Pagina iniziale



Pagina di visualizzazione di tutti gli ordini da fare e in attesa di pagamento

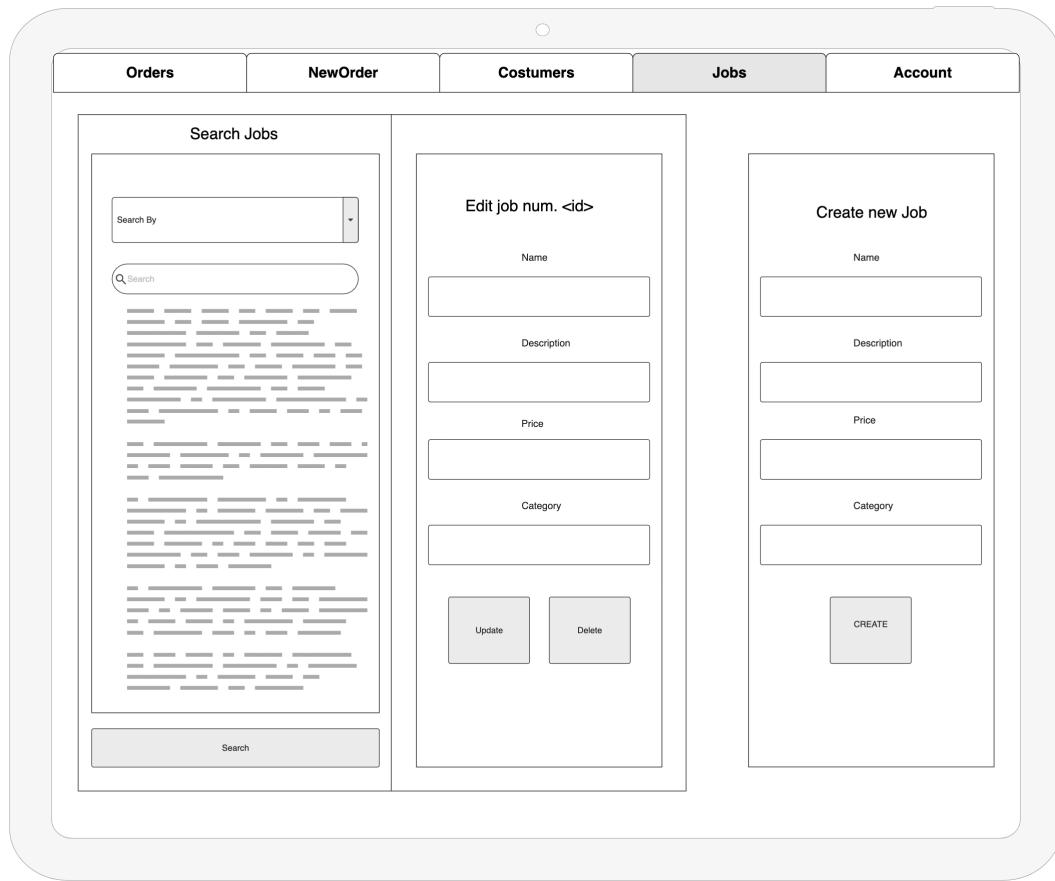


Pagina per la creazione di un nuovo ordine



2.6.3 punto di vista del SuperUser

Pagina per la creazione e la modifica di lavori (Job) possibili



2.7 Sicurezza , Autenticazione ed Autorizzazione

La gestione della sicurezza e degli accessi profilizzati è ottenuta tramite Spring Security. I controller (REST endpoints) permettono la chiamata ai loro metodi in base alle informazioni fornite dallo user attualmente autenticato, con Ruoli e Autorità personalizzate.

Sotto uno snippet della classe di configurazione

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .exceptionHandling() ExceptionHandlingConfigurer<HttpSecurity>
        .authenticationEntryPoint(authEntryPoint)
        .and() HttpSecurity
        .sessionManagement() SessionManagementConfigurer<HttpSecurity>
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and() HttpSecurity
        .authorizeRequests() ExpressionInterceptUrlRegistry
        .antMatchers("api/account/register").permitAll()
        .antMatchers("api/auth/**").permitAll()
        .antMatchers("api/orders/getOrderStatusByld").permitAll() // anyone should be able to see the state of their order
        .antMatchers("api/costumer/**", "api/jobs/**", "api/orders/**").hasAnyAuthority( ...authorities: "ADMIN", "SUPERUSER")
        .antMatchers("api/users/deleteUser").hasAuthority("SUPERUSER")
        .antMatchers("api/users/deleteUser/{id}").hasAuthority("SUPERUSER")
        .anyRequest().authenticated()
        .and() HttpSecurity
        .httpBasic();
    http.addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

Chapter 3

Implementazioni delle classi

3.1 Domain Model

3.1.1 Account

Rappresenta una istanza di un account di un cliente che può accedere al sito per vedere lo stato di tutti i suoi ordini. Gli attributi sono:

- **id:** un identificativo univoco dell'account
- **user:** le credenziali e il ruolo associati all'account
- **costumer:** il cliente a cui è associato l'account
- **fidelityPoints:** i punti fedeltà che danno accesso a eventuali sconti

3.1.2 Costumer

Rappresenta una singola istanza di un cliente che ha fatto almeno una richiesta di lavoro o che ha un account registrato. Gli attributi sono:

- **id:** l'identificativo univoco del cliente
- **name:** una stringa contenente il nome del cliente
- **surname:** una stringa contenente il cognome del cliente
- **email:** una stringa che contiene la e-mail del cliente sulla quale il suddetto cliente riceverà una notifica quando il lavoro sarà finito
- **phone:** un numero Long che conterrà il numero di telefono del cliente dove può ricevere anche qua una notifica quando il lavoro sarà finito

3.1.3 Job

Rappresenta una istanza di una tipologia di lavoro che viene eseguita dal sarto in servizio in negozio. Gli attributi sono:

- **id:** un numero univoco identificativo del lavoro
- **name:** una stringa contenente il nome della tipologia del lavoro
- **description:** una stringa che contiene una descrizione riassuntiva del lavoro
- **price:** contiene il prezzo del lavoro richiesto
- **category:**

3.1.4 Order

Rappresenta una richiesta di lavoro su un capo portato da un cliente, questa istanza è quella che viene modificata dall'Admin quando il lavoro viene finito. Gli attributi sono:

- **id:** un identificativo univoco dell'ordine
- **description:** una stringa contenente una descrizione dell'ordine ricevuto dal cliente
- **status:** una stringa che può contenere lo stato di "PENDING" o "DONE" e che definisce lo stato completato o meno del lavoro
- **dueDate:** una data che contiene la data massima per quando deve essere finito il lavoro
- **discount:** una percentuale che indica lo sconto che questo lavoro riceverà
- **totalPrice:** una variabile numerica che contiene il prezzo finale del lavoro
- **costumer:** contiene il cliente a cui è associato il lavoro
- **jobs:** una lista di lavori che devono essere fatti sul capo consegnato

3.1.5 Role

Rappresenta il ruolo che ogni account ha e rappresenta il livello di autorizzazioni che ha l'account a cui è associato. gli attributi sono:

- **id:** contiene un identificativo univoco per rappresentare il ruolo
- **name:** una stringa che contiene il nome del ruolo

3.1.6 UserEntity

Rappresenta le credenziali di accesso di un utente, al sito e/o all'applicazione, a differenza dei ruoli che possiede. Gli attributi sono:

- **id:** identificativo univoco per ogni istanza di account
- **username:** una stringa che contiene il nome dell'account
- **password:** una stringa che contiene la password associata all'account
- **roles:** i ruoli associati alle credenziali di accesso all'applicazione

3.2 Controller

Il package Controller contiene le classi che si occupano della gestione dei dati. Espone metodi per la creazione, modifica ed eliminazione dei dati dal Database, senza mai accedervi direttamente.

3.2.1 CostumerController

La suddetta classe espone i metodi per creare e modificare gli oggetti di tipo Costumer. I principali metodi implementati nella classe sono i seguenti:

- **getAllCostumers:** restituisce una lista di istanze di oggetti Customer presenti nel database;
- **searchCostumer:** dato nome o cognome del cliente cercato restituisce una lista di istanze di tutti i clienti trovati nel database che hanno quei criteri;
- **addCostumer:** dato in ingresso un oggetto CostumerDTO, descritta più avanti, aggiunge una nuova riga nel database con i dati salvati in CustomerDTO;
- **updateCostumer:** dato in ingresso un oggetto CostumerDTO, modifica il corrispondente Costumer nel database con i dati salvati nell'oggetto passato in ingresso;
- **deleteCostumer:** dato in ingresso un numero rappresentante l'id del customer da eliminare, elimina la riga corrispondente nel database.

3.2.2 JobController

La suddetta classe espone i metodi per creare e modificare i vari Job, i metodi sono accessibili solo attraverso un dato ruolo, descritto più tardi. I principali metodi implementati nella classe sono i seguenti:

- **getAllJobs:** restituisce la lista di tutti i Job salvati nel database, sottoforma di oggetti DTO, descritti più avanti;
- **getJobsByCategory:** passando in ingresso una stringa che contiene il nome della categoria voluta, il metodo restituisce la lista di tutti i Job di quella categoria presenti nel database;
- **searchJob:** passando in ingresso una stringa, il metodo cerca e restituisce tra i vari Job presenti quelli che hanno una stringa corrispondente al loro interno;
- **addJob:** passandogli un oggetto di tipo JobDTO, il metodo salva i vari dati corrispondenti in una nuova riga del database;
- **updateJob:** passandogli un oggetto di tipo JobDTO, il metodo salva i vari dati nella riga che corrisponde al Job passato in ingresso;
- **deleteJob:** passando in ingresso un id, il metodo cerca il Job corrispondente nel database e lo elimina.

3.2.3 OrderController

La suddetta classe espone i metodi per creare e modificare i vari Order, alcuni metodi sono accessibili solo attraverso un dato ruolo, descritto più tardi; contiene inoltre un riferimento alla classe MailService poiché è questo controller che si occupa di mandare mail di notifica ai clienti. I principali metodi implementati nella classe sono i seguenti:

- **getAllOrders:** restituisce tutti i lavori sui capi richiesti da tutti i clienti;
- **searchOrdersByCosumerString:** passando in ingresso una stringa che rappresenta il nome di un cliente, il metodo restituisce tutti i lavori richiesti da quei clienti con quel nome;
- **getOrdersByCostumerId:** passando in ingresso un numero intero, che rappresenta l'identificativo univoco di un cliente, il metodo restituisce tutti i lavori associati a quel preciso cliente;
- **getOrderById:** passando in ingresso l'id dell'ordine, il metodo restituisce il lavoro a cui corrisponde tale id;
- **addOrder:** passando in ingresso un oggetto di tipo OrderDTO, descritto più avanti, il metodo si occupa di salvare tutti i dati necessari in una nuova riga del database;
- **updateOrder:** passando in ingresso un oggetto di tipo OrderDTO il metodo si occupa di modificare i dati del lavoro a cui corrisponde nel database;
- **deleteOrder:** passando in ingresso l'id del lavoro, il metodo si occupa di eliminare tale lavoro dal database.

3.2.4 AuthController

La suddetta classe si occupa della autenticazione dei vari utenti e della registrazione di nuovi, ogni account ha un ruolo assegnato che gli permette poi di accedere o meno a determinati metodi; vedi la descrizione dei ruoli nella sezione corrispondente. I metodi implementati sono:

- **register:** passando in ingresso un oggetto RegisterDTO, che contiene tutte le informazioni di un nuovo utente, il metodo controlla che non ci siano username uguali già salvati, in caso ci sia restituisce un messaggio di errore e non fa altro, in caso tutto sia in ordine salva username e password inseriti, criptando prima la password;
- **login:** passando in ingresso un oggetto LoginDTO, che contiene le credenziali di accesso con cui qualcuno sta provando ad accedere, il metodo controlla che esista quell'username e che ne corrisponda anche la password, in caso corrispondano viene garantito l'accesso al sistema con il ruolo a cui corrisponde l'account.

3.3 Repository

Queste repository sono fornite dall'API appena menzionata. Sono delle interfacce che vengono implementate dal momento in cui vengono specificati la classe su cui operare e il tipo dell'id (chiave primaria) degli oggetti di quella classe. I principali metodi che vengono forniti dalle JPA Repository sono i seguenti:

- **save** : Dato un oggetto, lo salva/lo aggiorna nel Database
- **delete**: Dato un oggetto, lo cancella nel Database
- **findById**: Dato l'ID, restituisce l'oggetto corrispondente (se esiste)
- **existsByID**: Dato l'ID, restituisce un booleano che comunica se l'oggetto esiste all'interno del Database

Inoltre, le Repository permettono di scrivere query (di sola lettura) nel linguaggio SQL relative alle entità della classe a cui si riferiscono, tramite l'annotazione Spring Boot `@Query`. Essendo 4 le entità rappresentate nel nostro database, le JPA Repository sono 5:

- CustomerRepository
- JobRepository
- OrderRepository
- RoleRepository
- UserRepository

3.4 Service

Le classi Service sono interfacce progettate per fornire metodi utilizzati per l'interrogazione, il salvataggio ed l'eliminazione dei dati dal database. Queste classi rappresentano l'implementazione del pattern DTO, in quanto facilitano la comunicazione tra le classi esterne e il database, incapsulando le operazioni di accesso ai dati. I metodi principali esposti sono di 4 tipi:

- **addObject**: i metodi che hanno come prefisso "add" sono usati per aggiungere nuovi record alla tabella nel database
- **updateObject**: i metodi ce hanno come prefisso "update" si occupano della modifica di tuple nel database
- **getObject**: i metodi che hanno come prefisso "get" si occupano della interrogazione del database in vari modi, in base a quale metodo viene richiamato il database viene interrogato su alcuni specifici campi o senza nessun filtro
- **deleteObject**: i metodi che hanno come prefisso "delete" si occupano della cancellazioni delle tuple e funzionano tutti cercando la tupla attraverso l'id univoco che funziona da chiave primaria

Tre queste classi possiamo distinguere come più caratteristica la DAO - "Data Access Object", che rappresenta il pattern più notevole usato per implementare il problema.

3.5 DTO

Questo package si occupa della dichiarazione di oggetti usati dalle varie classi nel programma, questi oggetti rappresentano i dati salvati nel database e permettono al programma di usare sudetti dati e anche manipolarli, senza rischio di perdita di informazioni nel database; ogni volta che un campo viene modificato, l'oggetto viene poi salvato nel database tramite la classe service a cui è associato. Abbiamo 2 tipologie di DTO:

1. **Response**: usato dai metodi a più basso livello per salvarvi tutti i risultati che il database ha restituito, dato che contiene al suo interno una lista di oggetti DTO. Esistono 3 tipi di Response:

- (a) CostumerResponse
- (b) JobResponse
- (c) OrderResponse

2. **DTO:** oggetti che rappresentano 1 a 1 un oggetto del tipo corrispondente salvato nel database. Abbiamo tipi di oggetti DTO:

- (a) CostumerDTO
- (b) JobDTO
- (c) OrderDTO

Infine abbiamo un ultima classe chiamata **AuthResponseDTO** che, nonostante non rappresenti un oggetto salvato nel database, è importante per l'autenticazione degli account durante i logIn, questo oggetto contiene il token di accesso dell'account che ha effettuato un accesso.

3.6 Security

Questo package si occupa delle autorizzazioni dei vari account divise per ruolo, controllando se il percorso Url a cui l'utente sta cercando di accedere è ammissibile per il ruolo che l'utente attivo ha. Le aree accessibili per ruolo sono:

- **Nessun ruolo:** ha accesso solo alla pagina iniziale, a quella di logIn e alla pagina di registrazione di un nuovo utente
- **User:** ha accesso alle informazioni personali del proprio profilo, alle informazioni circa i lavori ordinati all'azienda
- **Admin:** ha accesso alle informazioni superficiali (nome, cognome, email e id_cliente) di tutti i clienti registrati, con o senza account al sito, e a tutti i lavori ordinati dai clienti; in più può modificare lo stato di un lavoro
- **SuperUser:** ha accesso alle stesse funzionalità di un Admin, con la capacità aggiuntiva di accedere alle informazioni criptate, inclusa la password di accesso al sito, ai dati degli Admin e ai file di configurazione delle autorizzazioni. Inoltre, può aggiungere o rimuovere account Admin, account User e tipologie di lavori (Job).

3.7 MailService

Questa classe sfrutta un @Service offerto dal framework Spring Boot Java **MailSender**. Si occupa di stabilire una connessione SMTP e inviare una email. Il metodo contenuto in essa è sendMail, che prende in ingresso 3 stringhe contenenti destinatario, oggetto e contenuto, inviando correttamente l'email e notificando il terminale del successo dell'operazione.

Dunque la classe **MailService** semplicemente crea un oggetto di tipo **SimpleMailMessage**, impostando gli attributi in questo modo (l'oggetto SimpleMailMessage lo chiamo *message*):

- **message.setFrom(*)**: questo metodo imposta il mittente, cioè l'azienda, e quindi lo imposteremo come valore costante;
- **message.setTo(*)**: questo metodo imposta il destinatario, che sarà il cliente di cui è pronto l'ordine, infatti gli passiamo proprio la mail del cliente associato all'ordine di cui è stato cambiato lo stato;
- **message.setSubject(*)**: questo metodo imposta l'oggetto della mail verrà inviata e lo impostiamo come costante, dato che l'obiettivo sarà sempre quello di notificare al cliente che il suo lavoro è pronto;
- **message.setText(*)**: questo metodo imposta il testo che verrà inviato nella mail, che sarà modificato solo in alcune parti che dipendono dall'ordine e dal cliente ad esso associato.

Una volta impostato tutti i campi dell'oggetto SimpleMailMessage, usiamo il metodo della classe *MailSender* nominata prima.

Chapter 4

Database

In questo capitolo parleremo di come è stata implementata la connessione al Database.

4.1 DBMS: MySQL

Il sistema di gestione di database (DBMS) utilizzato è MySQL, sviluppato da Oracle Corporation. Si tratta di un sistema RDBMS (Relational Database Management System), dove i dati sono organizzati in tabelle che sono in relazione tra loro. In questo contesto, le classi Java descritte nel Domain model corrispondono alle entità (con gli attributi rappresentati nelle colonne) e gli oggetti alle istanze.

```
# Database connection
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=postgres
```

Figure 4.1: Snippet del file ”application.properties” sulla connessione del DBMS

4.2 Connessione al DBMS

Come menzionato in precedenza, abbiamo utilizzato il framework Spring Boot, concentrandoci specificamente su uno dei suoi servizi chiamato Spring Data JPA. Questo servizio semplifica il mapping delle classi Java all'interno di un database e ne garantisce la persistenza. Inoltre, facilita la connessione tra l'applicazione Java e il database; questo avviene attraverso l'uso del comando `SpringApplication.run`, il quale restituisce un oggetto di tipo `ApplicationContext`. Quest'ultimo è essenziale perché permette l'inizializzazione di qualsiasi classe che utilizza i repository per comunicare con il database. Nel file `application.properties`, vengono specificati il DBMS, l'indirizzo IP, la porta e le credenziali di accesso al database.

4.3 Mappatura del Domain Model

Spring Data JPA mette a disposizione delle annotazioni (nella forma ”@Annotazione”) che forniscono delle funzionalità per le classi che devono far parte del Database: nel nostro caso specifico le classi appartenenti al Domain Model. Le annotazioni che sono state usate sono:

- **@Entity:** da posizionare prima della dichiarazione della classe. Specifica che questa classe diventerà un'entità (tabella) nel Database.
- **@Table:** specifica che nome deve avere la tabella corrispondente (se si vuole con nome diverso dalla classe) e lo schema relazionale a cui fa riferimento
- **@Id:** da posizionare prima dell'attributo che diventerà la chiave primaria, solitamente un Longint

```

14  | @Entity
15  | @Table(name = "accounts")
16  public class Account {
17  |     no usages
18  |     @Id
19  |     @GeneratedValue(strategy = GenerationType.IDENTITY)
20  |     private Long id;
21  |
22  |     no usages
23  |     @OneToOne
24  |     @JoinColumn(name = "user_id")
25  |     private UserEntity user;
26  |
27  |     no usages
28  |     @OneToOne
29  |     @JoinColumn(name = "costumer_id")
30  |     private Costumer costumer;
31  |
32  |     no usages
33  |     private Integer fidelityPoints = 0;
34  }

```

Figure 4.2: Snippet del codice della classe Account del Domain model

- **@GeneratedValue:** opzionale, indica che la chiave primaria è generata automaticamente. Specifica anche la strategia di generazione (nel nostro caso la strategia adottata è IDENTITY, ovvero le chiavi primarie vengono generate con valori contigui)
- **@ManyToOne / @OneToMany:** indica che l'attributo in questione ha un vincolo di integrità referenziale con uno di un'altra tabella (indicata dopo l'annotazione @JoinColumn)

La prima volta in cui viene inserita l'annotazione @Entity in una classe ed eseguito il programma, il DBMS procederà a effettuare automaticamente una query di creazione (CREATE TABLE) e inserirà come colonne tutti gli attributi

Chapter 5

Test

La suite di Test per l'applicazione è divisa in tre parti:

1. **Repository test:** i primi test vengono fatti sulle *Repository* per garantire l'effettiva persistenza dei dati sul database;
2. **Service test:** dopodiché sono stati testati i *Service* per garantire la connessione tra applicazione e Database. Queste due fasi dello unit testing sono state eseguite tramite SpringTest, con "h2Database", un database in memory che non inquina l'effettivo database delle entità;

```
@Test
public void RepositoryTest_findById_Costumer() {
    Costumer costumer = Costumer.builder().name("SaveTestName").surname("SaveTestSurname").email("renigega@outlook.it").phone(3280119573L).build();
    costumer = costumerRepository.save(costumer);
    System.out.println(costumer.toString());

    Costumer result = costumerRepository.findById(costumer.getId()).orElse(null);

    Assertions.assertNotNull(result);
    Assertions.assertEquals(costumer.getId(), result.getId());
}
```

Figure 5.1: find by id in repository

3. **Controller test:** infine vengono testati i *Controller* tramite chiamate da *Postman* (o in alternativa cURL) per garantire che anche il livello più alto sia funzionante.

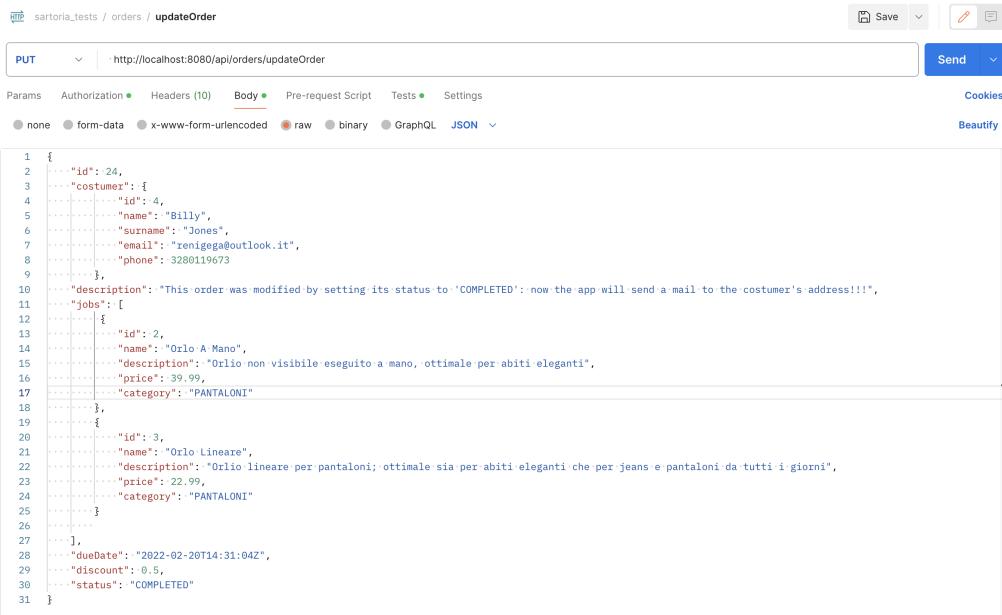
The screenshot shows the Postman application interface. The top bar has a 'sartoria_tests / costumers / addCostumer' tab. Below it, the 'POST' method is selected, and the URL is 'http://localhost:8080/api/costumer/addCostumer'. The 'Body' tab is active, showing a JSON payload:

```
1 {
2     "name": "TestCostumerName",
3     "surname": "TestCostumerSurname",
4     "email": "renigega@outlook.it",
5     "phone": 3280119673
6 }
```

Below the body, the response status is shown as 'Status: 201 Created Time: 478 ms Size: 466 B'. The response body is also displayed in JSON format:

```
1 {
2     "id": 7,
3     "name": "TestCostumerName",
4     "surname": "TestCostumerSurname",
5     "email": "renigega@outlook.it",
6     "phone": 3280119673
7 }
```

Figure 5.2: adding a new costumer



The screenshot shows the Postman application interface. At the top, it displays the URL `http://localhost:8080/api/orders/updateOrder`. Below the URL, there are tabs for Params, Authorization, Headers (10), Body (selected), Pre-request Script, Tests, and Settings. Under the Body tab, the content type is set to JSON. The body of the request contains a JSON object representing an order update:

```

1  {
2   "id": 24,
3   "costumer": {
4     "id": 4,
5     "name": "Billy",
6     "surname": "Jones",
7     "email": "renilegge@outlook.it",
8     "phone": 3280119673
9   },
10  "description": "This order was modified by setting its status to 'COMPLETED': now the app will send a mail to the costumer's address!!!",
11  "jobs": [
12    {
13      "id": 2,
14      "name": "Orlo A Mano",
15      "description": "Orlio non visibile eseguito a mano, ottimale per abiti eleganti",
16      "price": 39.99,
17      "category": "PANTALONI"
18    },
19    {
20      "id": 3,
21      "name": "Orlo Lineare",
22      "description": "Orlio lineare per pantaloni; ottimale sia per abiti eleganti che per jeans e pantaloni da tutti i giorni",
23      "price": 22.99,
24      "category": "PANTALONI"
25    }
26  ],
27  "dueDate": "2022-02-20T14:31:04Z",
28  "discount": 0.5,
29  "status": "COMPLETED"
30 }
31

```

Figure 5.3: updating an order



Dear Billy
 Your order with number 24 is ready for pickup!!!
 Thank you for choosing us!
 Sartoria Team

Figure 5.4: updating an order: email received from modifying the order status to "COMPLETE"

Chapter 6

Deployment

Per il deployment dell'applicazione, si potrà scegliere tra diversi servizi cloud come Microsoft Azure, Google Cloud Platform o Amazon Web Services. La scelta dipenderà dalle specifiche esigenze dell'azienda e dal budget a disposizione. Inoltre, il database utilizzato dall'applicazione sarà anch'esso gestito in cloud, con la selezione del servizio basata sui parametri precedentemente menzionati e sull'infrastruttura cloud scelta per l'applicazione stessa.

Chapter 7

Sviluppi Futuri

Tra gli sviluppi futuri pianificati ci sono l'implementazione di un sistema di assegnamento di punti fedeltà, che potrebbe richiedere l'utilizzo di un pattern Strategy. Un altro sviluppo potrebbe essere quello di inviare come notifica anche un messaggio o addirittura una telefonata automatica. Queste idee appena menzionate hanno già una implementazione iniziale che riguarda un paio di campi al momento superflui, o impostati manualmente da un Admin, all'interno di alcune classi (come ad esempio il campo *fidelityPoints* nella classe *Account*).