

DOCUMENTAZIONE SOFTWARE GESTIONE OSPEDALIERA

Andrea Potì & Lorenzo Castorina | Politecnico di Milano

Per una visualizzazione migliore del testo e di alcune immagini è consigliato utilizzare google docs sul browser tramite questo [link](#)

Sommario

1. Analisi dei requisiti
 - 1.1. Obiettivi
 - 1.2. Struttura
 - 1.3. Uml Diagrams
2. Progettazione
 - 2.1. Application design
 - 2.1.1. API
 - 2.2. Front-end
 - 2.3. Back-end
3. Testing
 - 3.1. Classi di Test con SpringBootTest e JUnit 5
 - 3.2. Rapporto sull'andamento dei test
4. Istruzioni per l'avvio dell'applicazione
 - 4.1. Credenziali di accesso
 - 4.2. Link utili

Analisi dei requisiti

Obiettivi

L'idea del progetto nasce da un'esigenza dei moderni ospedali. La mancanza di posti letto, la lentezza dei processi manuali e le lunghe attese dei pazienti ci hanno ispirato nella realizzazione di questo applicativo web semplice ed intuitivo, con il principale scopo di rendere la gestione ospedaliera più efficiente e performante, specialmente in questo difficile periodo di pandemia.

Nella prima fase del progetto abbiamo pensato a quali fossero le possibili esigenze degli operatori sanitari e quali mansioni giornalmente occupano più tempo.

Di queste si sono scelte quelle più meccaniche e ripetitive che potessero essere rese automatiche tramite l'utilizzo di un terminale.

Una volta individuate le esigenze più comuni, abbiamo proseguito alla realizzazione di una bozza grafica che desse un'idea generale sull'utilizzo del software.

Struttura

Nella seconda fase abbiamo pensato alla tipologia di applicativo.

In questo caso abbiamo scelto di utilizzare un modello client-server che permettesse all'operatore sanitario l'accesso diretto ai dati ospedalieri localizzati su un database.

Dal punto di vista client-side, l'operatore potrà interfacciarsi tramite un portale web disponibile sui browser più comuni. Proprio per questo la decisione di utilizzare Angular (versione 9.0), framework open-source per la creazione di applicazioni web, è stata la più opportuna, non solo per la possibilità di comporre progetti di elevata complessità, ma anche per le performance che permettono di renderizzare questo software anche tramite una semplice finestra del browser. Inoltre la possibilità di utilizzare HTML5, CSS3 e TypeScript (linguaggio derivato da Javascript) rendono l'esperienza di programmazione tramite Angular ancora più intuitiva e semplice.

Dal punto di vista server-side, si è scelto di utilizzare Spring Boot, un framework per lo sviluppo di applicazioni web basate su codice Java che offre un ulteriore livello di astrazione rispetto a Spring Framework, di più complicato utilizzo e adatto a progetti aziendali più elaborati.

Spring Boot ha numerosi vantaggi tra cui:

- Incorporare direttamente applicazioni web server/container come Apache Tomcat o Jetty, per cui non è necessario l'uso di file WAR (Web Application Archive);
- Configurazione di Maven semplificata grazie ai POM "Starter" (Project Object Models);
- Caratteristiche non funzionali come metriche o configurazioni esterne automatiche.

La memorizzazione delle informazioni è affidata ad un Database SQL, in particolare MySQL, che viene gestito a livello di persistenza dall'applicazione tramite l'uso di JPA (Java Persistence API) e della sua più famosa implementazione framework Hibernate.

I dati contenuti vengono monitorati tramite l'uso di MySQL Workbench, un visual tool che permette di verificare che i contenuti del database siano coerenti con le operazioni eseguite.

Uml Diagrams

Class Diagram

Il Class Diagram mostra le classi che compongono il software e come esse si relazionano l'una con l'altra.



Il diagramma in figura mostra le classi contenute all'interno del package “model” del progetto. Per questioni di visibilità grafica non sono stati riportati i costruttori e i metodi all'interno delle singole classi.

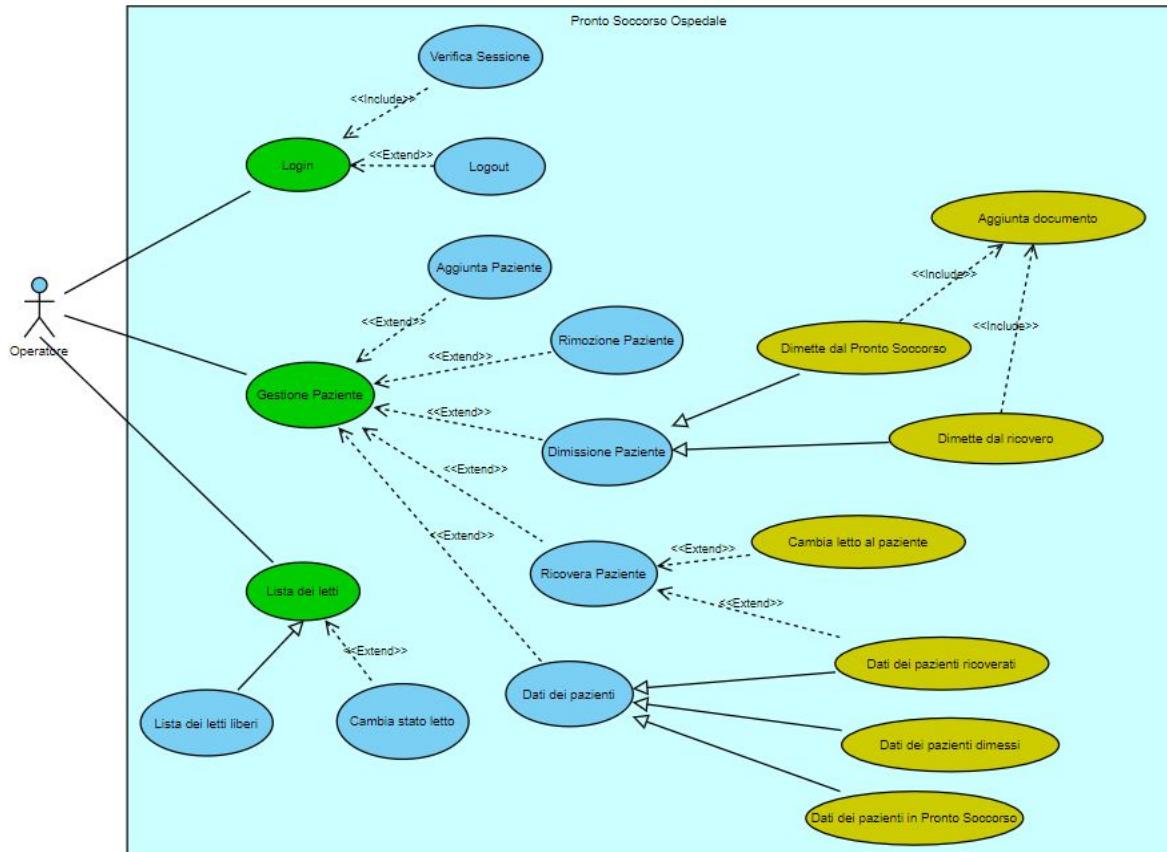
Le classi rappresentano le Entità di cui è composto il Database e ne riprendono le relazioni, in particolare:

- La classe **Operatore** possiede come attributi “username” e “password” e non è relazionata a nessun’ altra classe
- La classe **Paziente** ha come attributi l’intera anagrafica della persona, i “documenti” associati e si collega con la classe Documenti attraverso una relazione 1:*
- La classe **Letto** possiede l’attributo “stato” che indica se esso è occupato oppure libero e si collega alla classe Ricovero con una relazione 1:1
- La classe **Documento** possiede gli attributi “Paziente”, “nome” e “data creazione” e si collega alla classe Paziente con una relazione 1:1
- La classe **Ricovero** associa due altre classi cioè Paziente e Letto (nel Database rappresenta una tabella di associazione). Il ricovero ha come attributi il “paziente” e il “letto” dove quest'ultimo viene ricoverato. La classe ha una relazione di 1:1 sia con la classe Paziente sia con la classe Letto

Use Case Diagram

Gli Use Cases catturano il comportamento del sistema, illustrando quali sono i suoi requisiti funzionali.

Lo use case realizzato in figura permette di visualizzare tutte le funzionalità che sono collegate al ruolo dell'operatore all'interno del sistema di gestione dell'intero applicativo.

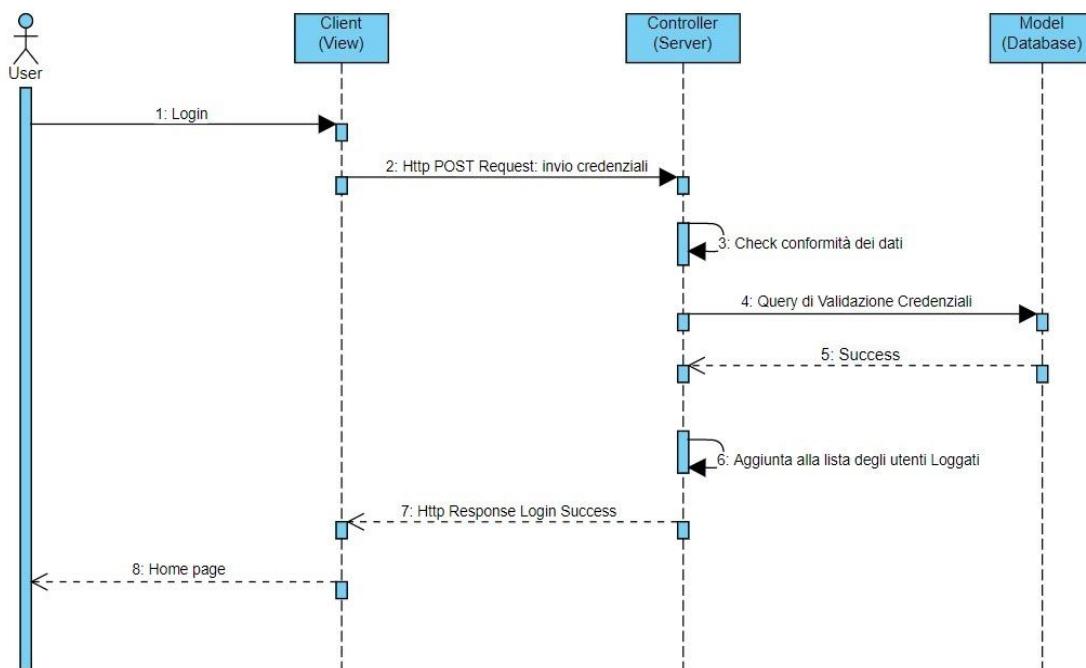


Come si può osservare, un operatore si autentica sul sistema, cioè effettua il login e dopo di che può eseguire una serie di funzionalità, sia per quanto riguarda la gestione dei pazienti, sia per quanto riguarda la gestione dei posti letto.

Sequence Diagram

Un Sequence Diagram mostra come i vari oggetti interagiscono per determinare un preciso comportamento del sistema.

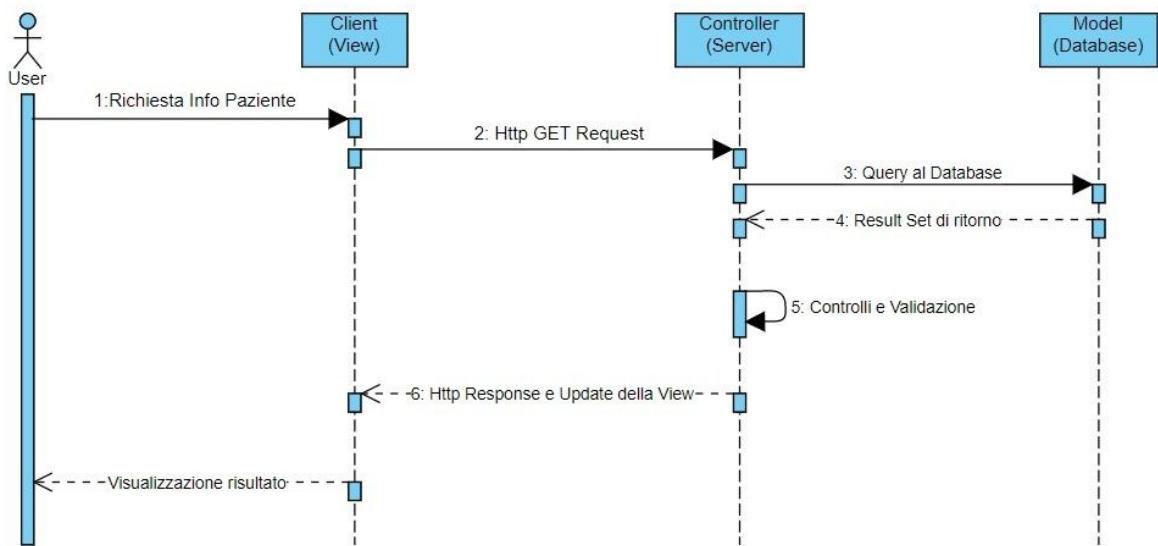
Si è scelto di mostrare con questa tipologia di diagramma l'operazione di Login avvenuta con successo e l'operazione di acquisizione dei dati dei pazienti presenti nel database.



Come si può osservare, gli oggetti che scambiano messaggi in questo scenario sono:

- **User**: l'utente, in questo caso l'operatore, che deve eseguire il login.
- **Client**: cioè il browser che l'operatore utilizza per interfacciarsi con il sistema.
- **Controller**: rappresenta il server che gestisce ed elabora le chiamate agli endpoint, effettuate attraverso l'interfaccia del client.
- **Model**: rappresenta il database da cui si estraggono le informazioni necessarie.

Client, Controller e Model richiamano il paradigma MVC su cui si basa l'intero lato server-side dell'applicativo.



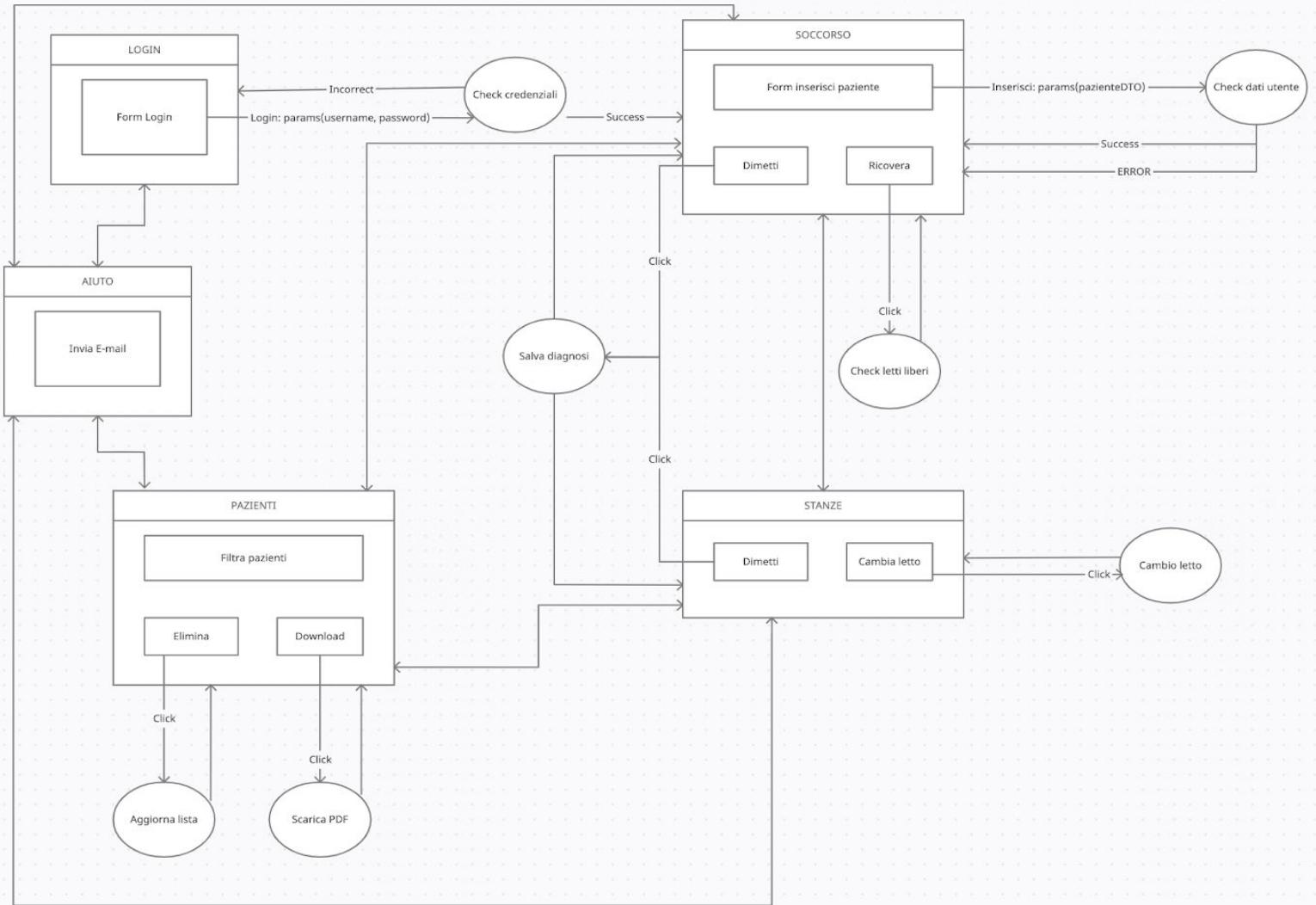
Il secondo Sequence Diagram mostra come i quattro partecipanti interagiscono mandando opportuni messaggi per soddisfare la richiesta del client, in questo caso ottenere tutti i dati dei pazienti nel database.

Il processo è il seguente:

- Lo **User** richiede interagendo con l'interfaccia grafica (la View) informazioni.
- La **View** traduce questa esigenza in una richiesta di tipo HTTP da inviare al Server.
- Il **Controller** elabora la richiesta eseguendo opportune query al Database.
- Il **Model** ritorna i risultati della query rispondendo al controller.
- Infine il Controller, dopo elaborazioni e controlli, rispedisce i dati alla view che le mostra all'utente che li ha richiesti.

Progettazione

Application design



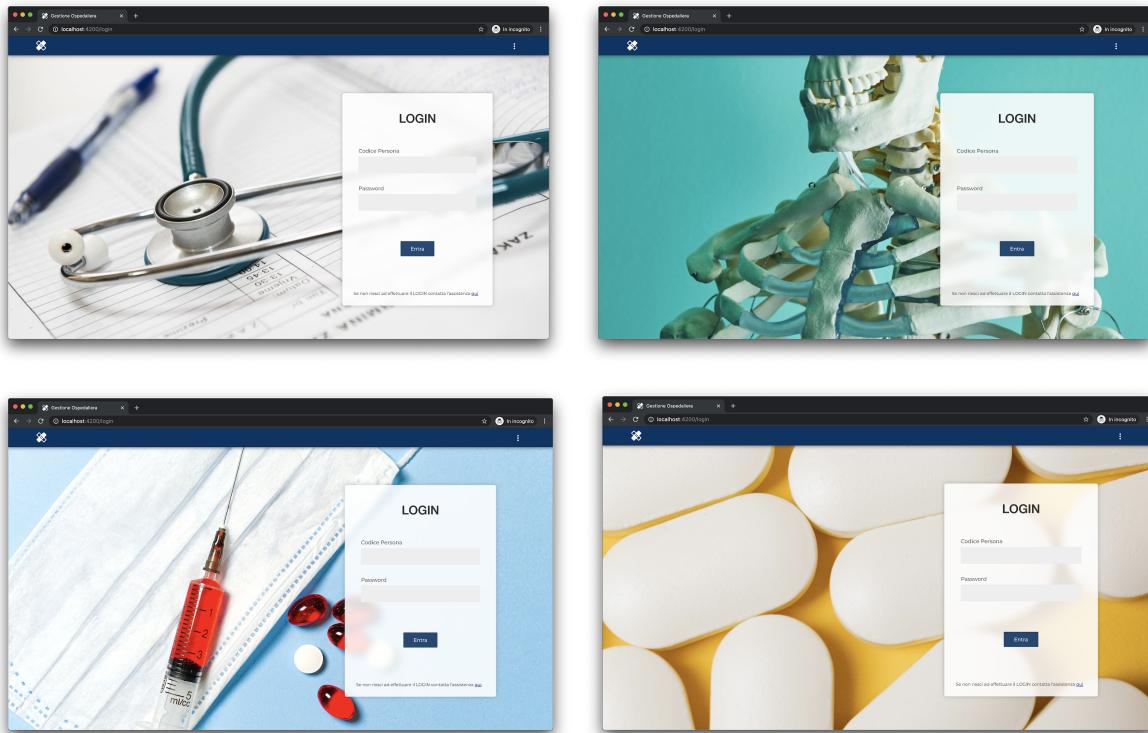
API

Per questioni di formattazione del testo e semplicità della documentazione, le API sono state messe su di un sito apposito consultabile presso questo [link](#)

Front-end

In questa sezione della documentazione verranno spiegate le scelte di implementazione da parte del front-end, come sono state realizzate le varie interfacce grafiche e le varie scelte di UX e UI design.

Login



Durante tutto l'utilizzo della piattaforma, l'utente (in questo caso l'operatore sanitario) si ritroverà in un ambiente intuitivo, familiare, e di facile utilizzo. Il design e lo stile dei componenti, che utilizzano un approccio minimal e pulito, prendono spunto ed ispirazione dal [Material Design](#). Anche l'utilizzo di pochi colori ma ben precisi, come il blu “`rgb(3,51,102)`”, il grigio chiaro, ed il giallo ambra “`mat-palette($mat-amber, A200, A100, A400)`” fanno risultare tutte le pagine ben leggibili e molto ordinate.

La pagina di login possiede un tema “ospedaliero”, permettendo di eseguire le operazioni classiche, come l'autenticazione, il recupero password e l'accesso alla pagina di supporto.

Pronto Soccorso

La piattaforma si suddivide in poche pagine, ma funzionali. Nella prima, dopo che viene effettuato l'accesso, l'utente si troverà dinanzi ad una dashboard che, grazie ad Angular 9 ed alla sua tecnologia di presentazione dei dati, gli "observable", sempre aggiornata ed in tempo reale, è in continua comunicazione con il server.

The screenshot shows a web application titled 'Gestione Ospedaliera' running on 'localhost:4200/soccorso'. The interface is divided into two main sections. On the left, there is a vertical list of patient names and their arrival times, each accompanied by a small colored circle (red, yellow, green, or grey). On the right, there is a large, light-grey rectangular area containing two smaller forms. The top form has a button labeled '+ Aggiungi paziente manualmente' and a QR code icon. The bottom form has fields for 'Nome:' and 'Cognome:', 'Orario entrata:' and 'Diagnosi:', and a 'Codice:' field. At the top of the right section, there are two more buttons: '+ Aggiungi paziente tramite scanner' and a QR code icon.

Qui l'operatore potrà aggiungere i vari pazienti al pronto soccorso, scegliendo tra un inserimento manuale o tramite scannerizzazione (che per motivi pratici è stata realizzata con il Qr Code).

[Link](#) per i Qr Code di prova

Il risultato finale sarà sempre il medesimo, un form dove i campi necessari sono costantemente controllati con delle regex che obbligano l'operatore ad inserire correttamente i dati.

Una volta aggiunto il paziente la dashboard verrà aggiornata.

Codice fiscale: Codice fiscale non corretto

CSTLNZ9C19H501&

Nazionalità:

Non specificato...

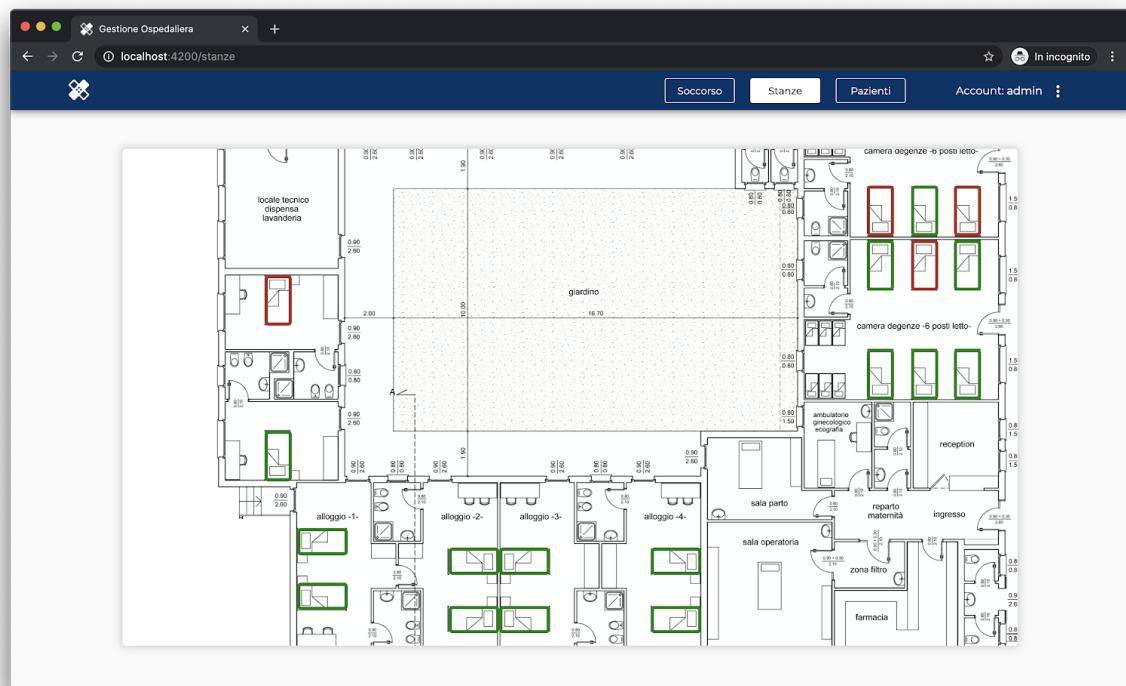
Infine è possibile visualizzare nel dettaglio le informazioni generali del paziente e decidere se ricoverarlo o dimetterlo senza passare tramite la struttura ospedaliera.

In questa sezione sono infine presenti delle piccole accortezze, come il controllo del codice fiscale doppio (quindi se il paziente è già presente in ospedale), l'ordinamento per codice colore (dal rosso al bianco) e per data di entrata, ed infine vari controlli sullo stato delle stanze (se tutte piene o meno) e sui pazienti già dimessi.

Nome:	Lorenzo	Cognome:	Castorina
Orario entrata:	10:20	Diagnosi:	Frattura gamba DX, è richiesta la visita di un medico specializzato. Cura tramite antibiotici per i primi 5gg, visita di ispezione tra 3 sett
Codice:	●		
		Ricovera	Dimetti

Stanze

Nella schermata successiva l'operatore si ritrova di fronte ad una "piantina" interattiva delle stanze dell'ospedale, dove possono essere visualizzati i vari letti liberi e i vari pazienti ricoverati, il tutto indicato da un semplice contorno colorato attorno al letto (rosso: occupato, verde: libero)



In questa sezione l'operatore avrà la possibilità di visualizzare le informazioni del paziente ricoverato, tra cui la diagnosi e la data di entrata, ma avrà anche la possibilità di cambiare letto al paziente qualora fosse necessario o di dimettere il paziente dalla clinica.

Lista Pazienti

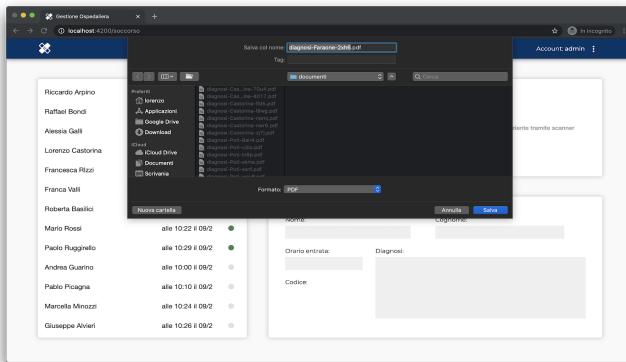
The screenshot shows a web-based application titled 'Gestione Ospedaliera'. The main area displays a table titled 'Lista pazienti' with 25 results. The columns show the patient's name, gender (M or F), and status (In pronto soccorso, In ricovero, Dimesso). The patients listed include Lorenzo Castorina, Andrea Poti, Mario Rossi, Caterina Verdi, Gianluca Ricci, Samuel Afghi, User Test, Alberto Manuguerra, Paola Ruggirello, Francesca Rizzi, Francesca Noto, and Riccardo Arpino. To the right of the table is a sidebar titled 'Filtri' (Filters) which includes fields for searching by name, selecting gender (Maschio or Femmina), specifying age range (Eta minore di - anni), and choosing general status (Pronto Soccorso, Ricoverato, Dimesso).

L'ultima schermata, ma non la meno importante, riguarda la visualizzazione dei pazienti. In questa sezione l'operatore avrà la possibilità di vedere TUTTI i pazienti, sia quelli attualmente in pronto soccorso, sia quelli in ricovero e infine quelli che sono stati dimessi.

È possibile inoltre eseguire dei filtri che permettono all'operatore di semplificare le ricerche.

I filtri vengono gestiti lato front-end, alleggerendo il carico al server e aumentando notevolmente le performance del software. Questo perchè il filtro viene applicato direttamente sui dati già scaricati precedentemente. Ciò permette di velocizzare il processo, senza bisogno di eseguire delle query ridondanti.

This screenshot is similar to the one above, showing the 'Gestione Ospedaliera' application. The main table 'Lista pazienti' still shows 25 results. The filter sidebar has been updated to search for patients named 'Caterina'. The other filter options remain the same: gender (Maschio or Femmina), age range (Eta maggiore di - anni), and general status (Pronto Soccorso, Ricoverato, Dimesso).



Con la dimissione del paziente, sia dal pronto soccorso che dal ricovero, verrà salvato un file pdf associato univocamente a quest'ultimo. Al suo interno si troverà la diagnosi, così da avere sempre a portata di mano uno storico dei documenti.

Per concludere è stata data la possibilità all'operatore di rimuovere i dati del paziente.

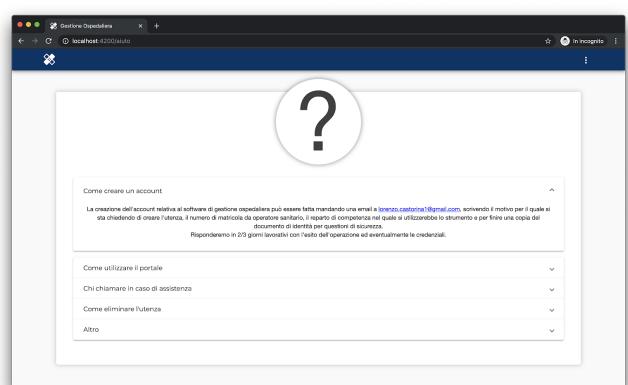
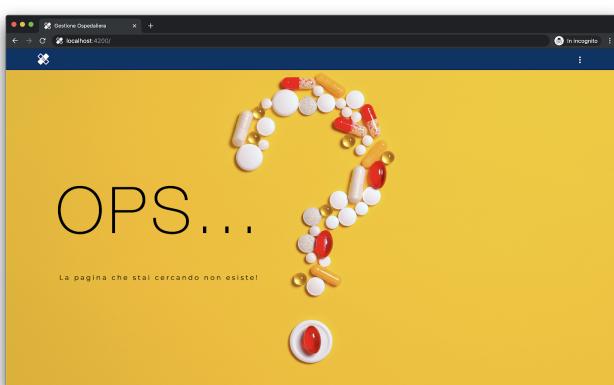
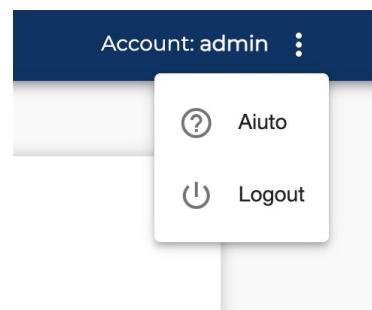
Questo risulta utile quando il paziente ne fa richiesta o risultasse deceduto.

Altro

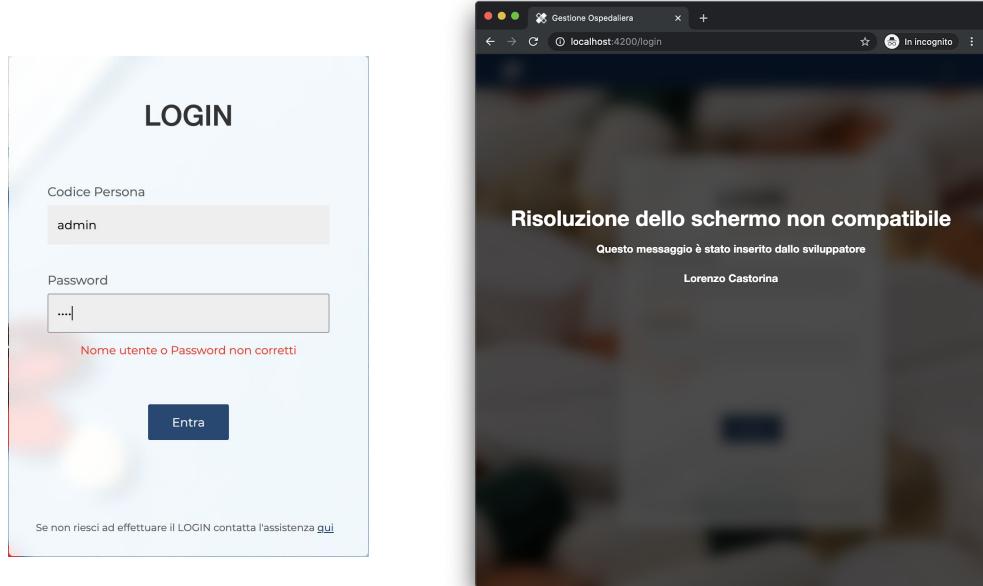
Infine l'applicazione è stata arricchita con delle features che rendono l'esperienza più piacevole e dinamica.

Ad esempio è possibile accedere alla pagina di aiuto da ogni schermata dell'applicazione, lo stesso vale anche per il logout

È stata inserita una pagina a supporto dell'utente con delle tabelle informative dinamiche con i vari link abilitati all'invio di email ed una pagina di "page not found"



Infine si possono notare piccoli dettagli, come la gestione della casistica dei letti tutti pieni, il recupero password con reindirizzamento, l'icona che permette di resettare la webapp, la possibilità di scaricare i vari documenti pdf dei pazienti ed uno script che disabilita la modalità mobile.



Back-end

Spring Boot e il modello MVC

Spring è un framework utilizzato dalle imprese medio/grandi per lo sviluppo di software che soddisfino requisiti di performance, sicurezza e affidabilità.

Più nello specifico, in questo progetto si è fatto uso di SpringBoot, una variante di Spring semplificata che permette di gestire in maniera automatica alcuni passaggi di configurazione e deployment, come ad esempio quella del Web Server Apache Tomcat, rendendo l'app più snella e veloce.

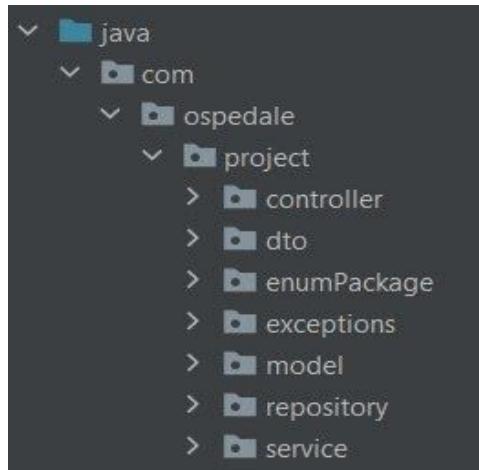
SpringBoot è stato affiancato a Maven, progetto open-source che permette di organizzare al meglio il progetto Java, poichè ha la caratteristica di essere modulare, cioè in fase di progettazione si scelgono in base alle funzionalità che deve avere l'applicativo, le librerie (chiamate dependencies) da aggiungere al progetto in un file denominato pom.xml (Project Object Model).

Di fondamentale importanza sono le Annotations di SpringBoot, indicate nelle classi Java con la @ che precede il loro nome e che specificano informazioni aggiuntive utili a definire valori, comportamenti, proprietà delle classi o degli attributi.

Il pattern che è stato utilizzato per il progetto è l'MVC (Model-View-Controller). L'applicazione deve separare le logiche di business dalle logiche di presentazione e di controllo che utilizzano tali funzionalità.

Dopo questa breve introduzione si approfondirà il modello accennato mostrando la struttura dei package e il ruolo delle classi all'interno dell'architettura applicativa.

Struttura dei package e funzionalità delle classi



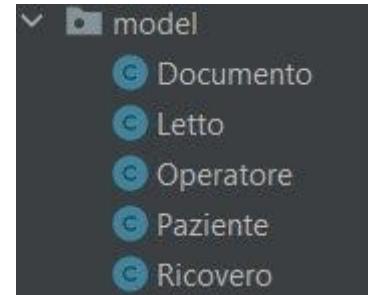
Il progetto è suddiviso in 7 package mostrati in figura

Model

Il package “model” contiene le classi Entity che rappresentano le tabelle del Database.

Ogni classe è dichiarata con l’annotazione @Entity facente parte di Jpa Hibernate, un framework per la gestione della persistenza.

Le Entity devono implementare l’interfaccia Serializable e avere un costruttore vuoto.



Ecco un esempio di Entity:

```
1  package com.ospedale.project.model;
2
3  import javax.persistence.Entity;
4  import javax.persistence.GeneratedValue;
5  import javax.persistence.GenerationType;
6  import javax.persistence.Id;
7  import java.io.Serializable;
8
9  @Entity
10 public class Operatore implements Serializable {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Long id;
15
16     private String username;
17     private String password;
18
19     public Operatore() {
20     }
21
22     public Operatore (String username, String password) {
23         this.username = username;
24         this.password = password;
25     }
26
27     public Long getId() { return id; }
28
29     public void setId(Long id) { this.id = id; }
30
31     public String getUsername() { return username; }
32
33     public void setUsername(String username) { this.username = username; }
34
35     public String getPassword() { return password; }
36
37     public void setPassword(String password) { this.password = password; }
38 }
```

Repository

Il package “repository” contiene le interfacce che estendono `JpaRepository<T, ID>` e sono annotate con `@Service`.

Esse si occupano di eseguire operazioni CRUD sulle Entity nel Database.

L’interfaccia `JpaRepository` offre alcuni metodi base come `findById()` e `findAll()`, invece se si ha bisogno di eseguire una query SQL nativa si può utilizzare l’annotazione `@Query`.



Ecco un esempio di Repository:

```
import com.ospedale.project.model.Paziente;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

@Repository
public interface PazienteRepository extends JpaRepository<Paziente, Long> {

    List<Paziente> findAllByIsActiveIsTrue();

    Optional<Paziente> findByCfAndIsActiveIsTrue(String cf);

    Optional<Paziente> findByCfAndIsActiveIsFalse(String cf);

    Optional<Paziente> findByIdAndIsActiveIsTrue(Long id);

    @Query("SELECT DISTINCT p FROM Paziente p WHERE p.isActive=true AND p.stato = 'ProntoSoccorso'")
    List<Paziente> findAllProntoSoccorso();

    @Query("SELECT DISTINCT p FROM Paziente p WHERE p.isActive=true AND p.stato = 'Ricoverato'")
    List<Paziente> findAllRicoverati();

    @Query("SELECT DISTINCT p FROM Paziente p WHERE p.isActive=true AND p.stato = 'Dimesso'")
    List<Paziente> findAllDimessi();

    @Query("SELECT DISTINCT p FROM Paziente p WHERE p.isActive=false")
    List<Paziente> findAllNonAttivi();
}
```

Service

Il package “service” contiene le classi che implementano le logiche di business dell'applicativo e che vengono richiamate principalmente nel controller.

Le classi service fanno uso delle repository per ottenere le informazioni dalla base di dati, elaborarle opportunamente e infine ritornarle al chiamante.

I vari repository vengono dichiarati all'interno dei service con l'annotazione @Autowired che permette di cercare le dipendenze giuste e importarle all'interno della classe.

Ecco un esempio di Service:

```
@Service
public class PazienteService {

    @Autowired
    PazienteRepository pazienteRepository;

    @Autowired
    LettoRepository lettoRepository;

    public void delete(Optional<Paziente> paziente) {
        paziente.get().setActive(false);
    }

    public List<Paziente> findAll() { return pazienteRepository.findAllByIsActiveIsTrue(); }

    public Optional<Paziente> findEliminatiByCf(String cf) { return pazienteRepository.findByCfAndIsActiveIsFalse(cf); }

    public Optional<Paziente> findByCf(String cf) {
        return pazienteRepository.findByCfAndIsActiveIsTrue(cf);
    }

    public Optional<Paziente> findById(Long id) { return pazienteRepository.findByIdAndIsActiveIsTrue(id); }

    public void save(Paziente paziente) {
        pazienteRepository.save(paziente);
    }

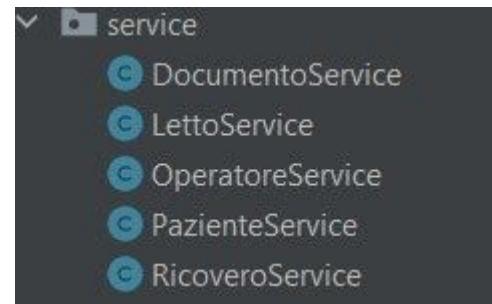
    public List<Paziente> findAllProntoSoccorso() { return pazienteRepository.findAllProntoSoccorso(); }

    public List<Paziente> findAllEliminati() { return pazienteRepository.findAllEliminati(); }

    public List<Paziente> findAllRicoverati() { return pazienteRepository.findAllRicoverati(); }

    public List<Paziente> findAllDimessi() { return pazienteRepository.findAllDimessi(); }

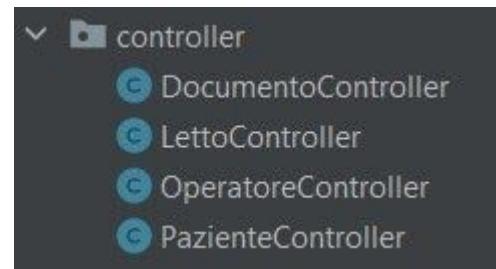
    public void aggiornaPaziente(Paziente paziente, PazienteDTO pazienteDTO) {
        paziente.setCf(pazienteDTO.cf);
        paziente.setNome(pazienteDTO.nome);
        paziente.setCognome(pazienteDTO.cognome);
        paziente.setCodice(pazienteDTO.codice);
        paziente.setDiagnosi(pazienteDTO.diagnosi);
        paziente.setLuogo_nascita(pazienteDTO.luogo_nascita);
    }
}
```



Controller

Il package “controller” contiene le classi Controller, annotate con @RestController. I controller fanno uso dei service (richiamati con @Autowired) per prelevare i dati secondo le logiche di business, mappando le richieste HTTP chiamate dalla view ai microservizi sviluppati e rispondendo utilizzando il medesimo protocollo e lo standard JSON.

Le richieste GET vengono mappate usando @GetMapping() con tra parentesi indicato il path dell'endpoint. Le richieste POST vengono mappate usando @PostMapping().



Ecco un esempio di Controller:

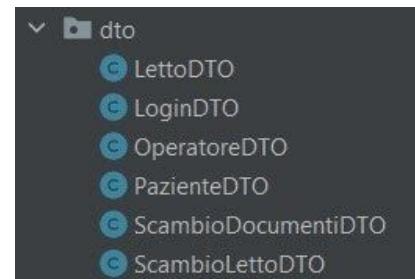
```
27     @CrossOrigin(origins = "*") //Per fare le richieste sulla stessa rete del frontend, "*" significa tutte
28     @RestController
29     @RequestMapping(path = "/paziente")
30     public class PazienteController {
31
32         @Autowired
33         PazienteService pazienteService;
34
35         @Autowired
36         RicoveroService ricoveroService;
37
38         @Autowired
39         LettoService lettoService;
40
41         @GetMapping(path = "/getPazienti")
42         public List<PazienteDTO> getPazienti () {
43             return pazienteService.findAll().stream().map(PazienteDTO::new).collect(Collectors.toList());
44         }
45
46         @GetMapping(path = "/soccorso")
47         public List<PazienteDTO> getPazientiProntoSoccorso () {
48             List<Paziente> pazienti = pazienteService.findAllProntoSoccorso();
49             //Ordino in base al codice e in base alla data a parità di codice
50             Collections.sort(pazienti, Comparator.comparing((Paziente p) -> (p.getCodice())).thenComparing(p -> p.getData_entrata()));
51             return pazienti.stream().map(PazienteDTO::new).collect(Collectors.toList());
52         }
53
54         @GetMapping(path = "/ricovero")
55         public List<PazienteDTO> getPazientiRicoverati () {
56             List<Paziente> pazienti = pazienteService.findAllRicoverati();
57             return pazienti.stream().map(PazienteDTO::new).collect(Collectors.toList());
58         }
59
60         @GetMapping(path = "/dimessi")
61         public List<PazienteDTO> getPazientiDimessi () {
62             List<Paziente> pazienti = pazienteService.findAllDimessi();
63             return pazienti.stream().map(PazienteDTO::new).collect(Collectors.toList());
64         }
65     }
```

DTO

Il package “dto” contiene le classi che definiscono gli standard per i passaggi di dati tra il back-end e il front-end, utilizzando come tipo il JSON.

Tutti i metodi del controller che devono ritornare dati strutturati facendo uso dei DTO.

Ecco un esempio:



```
package com.ospedale.project.dto;

import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.annotation.JsonInclude.Include.NON_NULL;

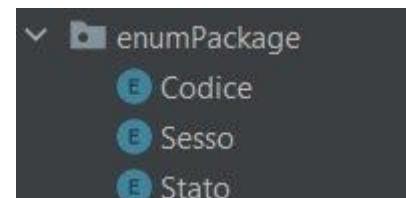
@JsonInclude(JsonInclude.Include.NON_NULL)
public class LoginDTO {
    public String username;
    public String password;

    public LoginDTO(String username, String password) {
        this.username = username;
        this.password = password;
    }
}
```

Enum

Il package “enum” contiene le classi Enum utilizzate all’interno delle altre come attributi.

Ecco un esempio:



```
package com.ospedale.project.enumPackage;

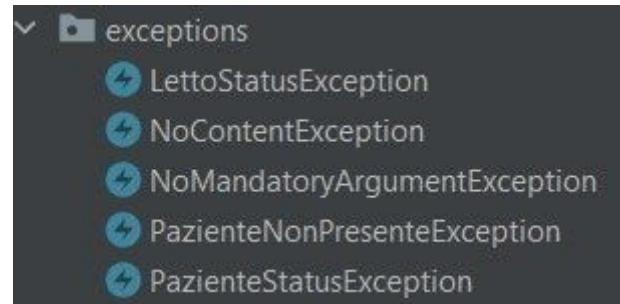
import javax.annotation.Nullable;

@Nullable
public enum Codice {
    ROSSO,
    GIALLO,
    VERDE,
    BIANCO
}
```

Exceptions

Il package “exceptions” contiene le classi che estendono la classe *ResponseStatusException* (lanciate in risposta a richieste HTTP). Queste classi rappresentano eccezioni personalizzate create per mandare dettagli più precisi alla view (front-end), sia per quanto riguarda lo status HTTP dell'errore generato, sia per quanto riguarda un messaggio che descriva brevemente il problema.

Ecco un esempio:



```
package com.ospedale.project.exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.server.ResponseStatusException;

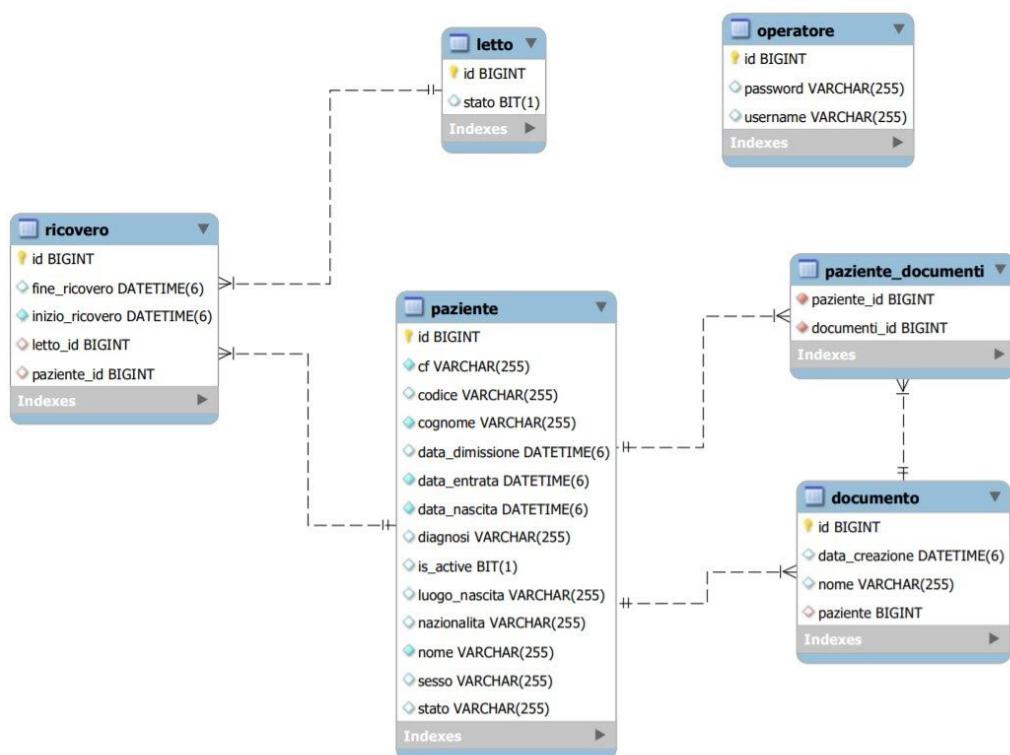
public class NoMandatoryArgumentException extends ResponseStatusException {

    public NoMandatoryArgumentException(HttpStatus status) { super(status, "Mancano dei campi obbligatori"); }
}
```

Diagramma ER e struttura del Database

Il Database, come spiegato in precedenza, è strutturato sulla base delle classi presenti all'interno del package “model”, contenente le Entity.

Il diagramma Entità-Relazioni del Database è il seguente:

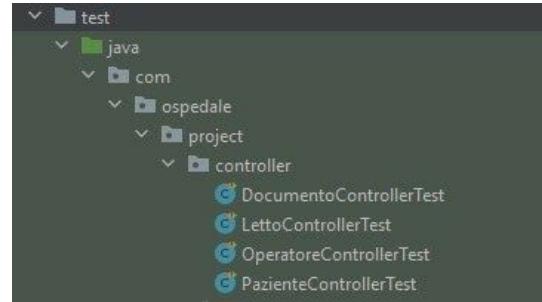


Testing

Classi di Test con SpringBootTest e JUnit 5

I Test d'integrazione per l'applicazione sono contenuti all'interno del package “test”.

I test sono stati realizzati con SpringBootTest, una libreria Maven che permette di effettuare un setup automatico dell'ambiente di test.
Questa si richiama tramite l'annotazione @SpringBootTest all'inizio di ogni classe.



```
@SpringBootTest  
public class PazienteControllerTest {
```

In questo progetto sono stati realizzati dei test d'integrazione per ogni classe annotata con @Controller, su ogni singola chiamata ai servizi in maniera separata.

Il metodo testato va annotato con @Test e al suo interno si richiamano i metodi della classe Assertions (facente parte della libreria JUnit 5 prevista da SpringBootTest) per verificare la coerenza dei dati.

Ecco un esempio del metodo testGetPazienti() all'interno del controller del paziente che ha l'obiettivo di testare correttamente il metodo getPazienti() :

```
@Test  
public void testGetPazienti() {  
    //Lista pazienti nel Database e Lista pazienti ottenuta con la chiamata all'API  
    List<PazienteDTO> pazientiDTO = pazienteController.getPazienti();  
    List<Paziente> pazienti = pazienteRepository.findAll();  
  
    //Verifico gli Assert  
    assertNotNull(pazienti);  
    assertFalse(pazienti.isEmpty());  
    assertNotNull(pazientiDTO);  
    assertFalse(pazientiDTO.isEmpty());  
    assertEquals(pazientiDTO.size(), pazienti.size());  
  
    //Convertendo PazienteDTO e Paziente in due liste di stringhe con i codici fiscali  
    List<String> pazientiDTOcf = pazientiDTO.stream().map(PazienteDTO::getCf).collect(Collectors.toList());  
    List<String> pazientiCf = pazienti.stream().map(Paziente::getCf).collect(Collectors.toList());  
  
    //Verifico confrontando il cf se tutti i pazienti sono stati inseriti correttamente  
    pazientiDTOcf.forEach(cf -> assertTrue(pazientiCf.contains(cf)));  
}
```

La logica seguita nel test in figura è la seguente:

- Si ottengono due liste di pazienti, una con i dati presenti nel Database e la seconda con i dati ottenuti chiamando l'API dal controller del paziente.
- Successivamente si verifica che le due liste non siano vuote e non siano oggetti *null*.
- Si controlla che tutti i codici fiscali presenti nella lista generata dalla chiamata al servizio siano tutti contenuti nella lista di pazienti presenti sul Database.

I dati utilizzati per eseguire i test sono aggiunti in un Database in memory chiamato *H2*, incapsulato all'interno dell'applicazione.

I metodi *inizializeDb()* e *deleteDb()*, richiamati rispettivamente nei metodi *init()* e *destroy()*, permettono di riempire e svuotare il Db con i dati necessari per i test, prima e dopo ogni metodo.

```
public void init() {  
    //Annotations  
    @BeforeEach  
    void setUp() {  
        Avviando paziente = new Paziente();  
        Avviando paziente.setNome("Paziente");  
        Avviando paziente.setCognome("Avviando");  
        Avviando paziente.setCf("123456789012345678");  
        pazienteRepository.insert(paziente);  
    }  
  
    //Test  
    paziente = new Paziente();  
    paziente.setNome("Paziente");  
    paziente.setCognome("Paziente");  
    paziente.setCf("123456789012345678");  
    pazienteRepository.insert(paziente);  
}  
  
@AfterEach  
void tearDown() {  
    pazienteRepository.deleteAll();  
}
```

```
public void deleteDb() {  
    //Elimino tutti gli elementi dal  
    //ricoveroRepository.deleteAll();  
  
    lettoRepository.deleteAll();  
  
    pazienteRepository.deleteAll();  
}
```

Le annotazioni *@BeforeEach* e *@AfterEach* prima dei metodi *init()* e *destroy()* servono proprio a questo scopo.

```
@BeforeEach  
public void init() { initializeDb(); }  
  
@AfterEach  
public void destroy () { deleteDb(); }
```

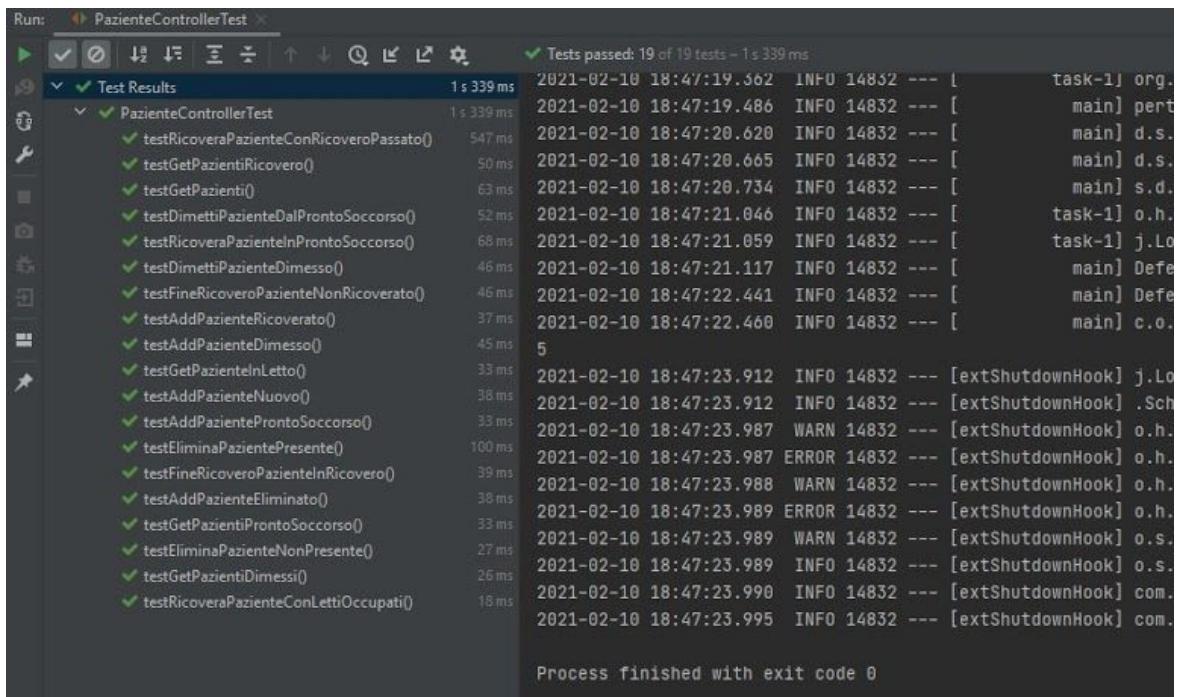
Utilizzare un database in memory permette di non intaccare il database originale e di avere un ambiente di test più efficiente.

I metodi della classe Assertions usati più di frequente sono:

- **assertTrue()** verifica che la condizione passata nelle parentesi tonde sia vera;
- **assertFalse()** verifica che la condizione passata nelle parentesi tonde sia falsa;
- **assertNull()** verifica che la condizione passata nelle parentesi tonde ritorni null;
- **assertNotNull()** verifica che la condizione passata nelle parentesi tonde non ritorni null;
- **assertSame()** verifica che due oggetti o variabili siano uguali;
- **assertThrows()** verifica che una data Exception, passata come parametro, venga lanciata da un preciso statement eseguibile.

Rapporto sull'andamento dei test

Tutti i test scritti per le classi controller hanno dato esito positivo.
In figura vengono mostrati i test eseguiti con successo della classe *PazienteControllerTest* associata al controller *PazienteController*.



The screenshot shows the IntelliJ IDEA interface with the 'Run' tool window open. The title bar says 'Run: PazienteControllerTest'. The left pane displays a tree view of test results under 'Test Results'. The root node is 'PazienteControllerTest' with a duration of 1s 339 ms. It contains 19 child nodes, each representing a test method that has passed. The right pane shows the log output for these tests. The log starts with 'Tests passed: 19 of 19 tests - 1s 339 ms' and lists 19 entries, one for each test method. Each entry includes the timestamp, log level (INFO), thread ID (14832), duration (e.g., 547 ms), and the name of the class and method. The log ends with 'Process finished with exit code 0'.

Method	Duration	Timestamp	Log Level	Thread	Class	Method
testRicercaPazienteConRicoveroPassato()	547 ms	2021-02-10 18:47:19.362	INFO	14832 ---	[task-1] org.	pert
testGetPazientiRicovero()	50 ms	2021-02-10 18:47:20.620	INFO	14832 ---	[main] d.s.	d.s.
testGetPazienti()	63 ms	2021-02-10 18:47:20.665	INFO	14832 ---	[main] d.s.	s.d.
testDimettiPazienteDalProntoSoccorso()	52 ms	2021-02-10 18:47:20.734	INFO	14832 ---	[main] o.h.	o.h.
testRicercaPazienteInProntoSoccorso()	68 ms	2021-02-10 18:47:21.046	INFO	14832 ---	[task-1] j.lo	j.lo
testDimettiPazienteDimesso()	46 ms	2021-02-10 18:47:21.059	INFO	14832 ---	[main] Defe	Defe
testFineRicoveroPazienteNonRicoverato()	46 ms	2021-02-10 18:47:21.117	INFO	14832 ---	[main] Defe	Defe
testAddPazienteRicoverato()	37 ms	2021-02-10 18:47:22.441	INFO	14832 ---	[main] c.o.	c.o.
testAddPazienteDimesso()	45 ms	2021-02-10 18:47:22.460	INFO	14832 ---	[main] c.o.	c.o.
testGetPazienteInLetto()	33 ms	2021-02-10 18:47:23.912	INFO	14832 ---	[extShutdownHook] j.lo	j.lo
testAddPazienteNuovo()	38 ms	2021-02-10 18:47:23.912	INFO	14832 ---	[extShutdownHook] .sch	.sch
testAddPazienteProntoSoccorso()	33 ms	2021-02-10 18:47:23.987	WARN	14832 ---	[extShutdownHook] o.h.	o.h.
testEliminaPazientePresente()	100 ms	2021-02-10 18:47:23.987	ERROR	14832 ---	[extShutdownHook] o.h.	o.h.
testFineRicoveroPazienteInRicovero()	39 ms	2021-02-10 18:47:23.988	WARN	14832 ---	[extShutdownHook] o.h.	o.h.
testAddPazienteEliminato()	38 ms	2021-02-10 18:47:23.988	WARN	14832 ---	[extShutdownHook] o.h.	o.h.
testGetPazientiProntoSoccorso()	33 ms	2021-02-10 18:47:23.989	ERROR	14832 ---	[extShutdownHook] o.h.	o.h.
testEliminaPazienteNonPresente()	27 ms	2021-02-10 18:47:23.989	WARN	14832 ---	[extShutdownHook] o.s.	o.s.
testGetPazientiDimessi()	26 ms	2021-02-10 18:47:23.989	INFO	14832 ---	[extShutdownHook] com.	com.
testRicercaPazienteConLettiOccupati()	18 ms	2021-02-10 18:47:23.990	INFO	14832 ---	[extShutdownHook] com.	com.
		2021-02-10 18:47:23.995	INFO	14832 ---	[extShutdownHook] com.	com.

Istruzioni per l'avvio dell'applicazione

Prerequisiti

Per il corretto funzionamento dell'applicazione sono **consigliati** degli applicativi che elencheremo in seguito:

- [Java](#) (versione 8)
- Java JDK (**versione 1.8**)
- [IntelliJ IDEA Ultimate](#) (versione 2020.3.1 o superiore)
- [MySQL server](#) (versione 8.0.0 o superiore)
- [MySQL Workbench](#) (versione 8.0.0 o superiore)
- [Visual Studio Code](#)
- [Node.js](#) (versione 14.15.5 LTS o superiore)
- Angular CLI (**versione 9.1.13**)
- Browser web (è consigliato l'utilizzo di [Chrome](#))

Alcuni di questi applicativi verranno installati in seguito, altri tramite linea di comando, è indispensabile però rispettare la versione indicata per riuscire ad arrivare al corretto funzionamento dell'applicazione.

Configurare il DB

Una volta scaricato l'installer di MySQL server si può procedere selezionando il profilo "developer", accettando i "Termini e condizioni" e terminare l'installazione. Durante l'installazione potrebbe richiedere di dover scaricare delle librerie per python e visual studio, ma nel nostro caso non sono necessarie.

Una volta scaricato il tutto aprire il cmd di windows con i permessi da amministratore e digitare `mysqld`

Su Mac/Linux basterà digitare sul terminale la stringa `sudo service mysql start`

A questo punto si aprirà MySQL server dove sarà necessario solamente impostare una password per il profilo (nel nostro caso noi abbiamo utilizzato la pass: `root`) ed accendere il server.

Per quanto riguarda MySql Workbench, una volta installato dal sito ed aperto, clicchiamo sul + per creare una nuova connessione, diamogli il nome che più desideriamo, impostiamo una password cliccando su “store in vault”, e testiamo la connessione al server.

Una volta fatto ciò all'interno dell'applicazione clicchiamo su “data import” per importare la struttura del database. Selezioniamo “import from self contained file” e selezioniamo il file presente a questo [link](#)

Aggiungiamo un nuovo schema e digitiamo “gestione_ospedale”, una volta fatto ciò confermiamo ed importiamo il tutto.

Se tutto è andato a buon fine sarà possibile visualizzare la struttura del database ed eseguire delle query di prova.

Impostare il Back-end

Una volta scaricato Java dal sito ufficiale, possiamo continuare con il download della repository del progetto, essa è situata a questo [link](#)

Una volta clonata da Git o scaricata in formato zip (scelta consigliata) possiamo procedere con l'installazione di IntelliJ IDEA.

Nel nostro caso abbiamo utilizzato la versione ULTIMATE in quanto gratuita grazie alla partnership con il Politecnico di Milano.

Impostiamo il launcher a 64 bit e selezioniamo l'opzione “add launcher DIR to the path”, clicchiamo next e finiamo l'installazione.

Apriamo la cartella del progetto tramite il launcher e aspettiamo che carichi tutte le dipendenze. Ora dovrebbe comparire l'opzione di scaricare Java JDK versione 1.8 (rilevata in automatico da IntelliJ IDEA), la scarichiamo e procediamo con l'avvio del server.

Se necessario impostare il DB, recarsi presso **src/main/resources/application.properties** all'interno del progetto, ed impostare username e password del DB MySql ed eventualmente cambiare l'URL allo schema creato precedentemente.

A questo punto rechiamoci presso
src/main/java/com/ospedale/project/GestioneOspedaleApplication.java e clicchiamo sull'icona verde sulla sinistra del codice per avviare il server.

Se tutto è partito correttamente sarà possibile eseguire una chiamata di prova al server copiando ed incollando <http://localhost:8080/testHealth> nel browser vedendo se compare *true*.

Impostare il Front-end

Una volta scaricato Visual Studio Code dal sito ufficiale, possiamo continuare con il download della repository del progetto, essa è situata a questo [link](#)

Una volta clonata da Git o scaricata in formato zip (scelta consigliata) possiamo procedere con l'installazione di Node.js

Potrebbe chiederci di dover installare **chocolatey** (package manager), noi acconsentiamo e procediamo.

Per vedere se node è stato correttamente installato digitiamo sul cmd o sul terminale questo comando `node -v`, se ci restituirà la versione di node.js lo avremmo installato.

Passiamo all'installazione di Angular 9, essa deve essere esattamente la 9.1.13 perchè il progetto è stato sviluppato con delle dipendenze specifiche per questa versione.

Apriamo il cmd o il terminale in modalità amministratore e digitiamo:

```
npm install -g @angular/cli@9.1.13
```

Una volta installato correttamente Angular 9, è possibile visualizzarne la corretta installazione digitando nel terminale `ng version` e vedere se ci restituisce la versione di Angular corretta.

Per finire apriamo la nostra directory di progetto tramite Visual Studio Code, apriamo un terminale o un cmd direttamente **dentro questa directory**, e digitiamo il comando `npm install` per scaricare tutte le librerie necessarie ad Angular.

Una volta terminato possiamo procedere e far partire l'applicazione digitando: `ng serve` il quale farà partire il server del progetto Angular il quale molto probabilmente sarà situato presso <http://localhost:4200>, se non risulta tale indirizzo, controllare il terminale e visualizzare l'indirizzo da lui suggerito.

Eventualmente potrebbe dare un errore a causa del file “package.json” in questo caso è possibile risolvere il problema digitando il comando:

```
npm install --save-dev @angular-devkit/build-angular
```

Se è necessario cambiare la porta e l'indirizzo di collegamento tra back-end e front-end, all'interno del progetto recarsi sul file **src/app/ajax.service.ts** e cambiare l'URL o la porta tramite le variabili “url” e “port”.

Se tutto è andato a buon fine, aprendo l'applicazione web dal browser tramite l'indirizzo consigliato sul terminale ci ritroveremo sulla pagina di login.

Credenziali di accesso

Le credenziali standard per entrare nell'applicazione sono:

- Username: **admin**
- Password: **admin**

Link utili

- **Git front-end:**
<https://gitlab.com/lorenzocastorina/gestione-ospedaliera-fe.git>
- **Git back-end:**
https://gitlab.com/obster991/gestione_ospedale_be.git
- **Qr Code per le scannerizzazioni del paziente:**
[https://drive.google.com/drive/folders/1WC3YLb5L05fd2MjYt0JI4xItjxVEdJrx
?usp=sharing](https://drive.google.com/drive/folders/1WC3YLb5L05fd2MjYt0JI4xItjxVEdJrx?usp=sharing)
- **Dump del DB:**
[https://drive.google.com/drive/folders/1vIrk5ZiSDN42EKEMdeFkdLzrJSST64
Yc](https://drive.google.com/drive/folders/1vIrk5ZiSDN42EKEMdeFkdLzrJSST64Yc)
- **API:**
[https://www.notion.so/Gestione-Ospedale-081293c87ec44a788d2122aed06
c3c0f](https://www.notion.so/Gestione-Ospedale-081293c87ec44a788d2122aed06c3c0f)

Lorenzo Castorina & Andrea Potì

Roma - 12/02/2021