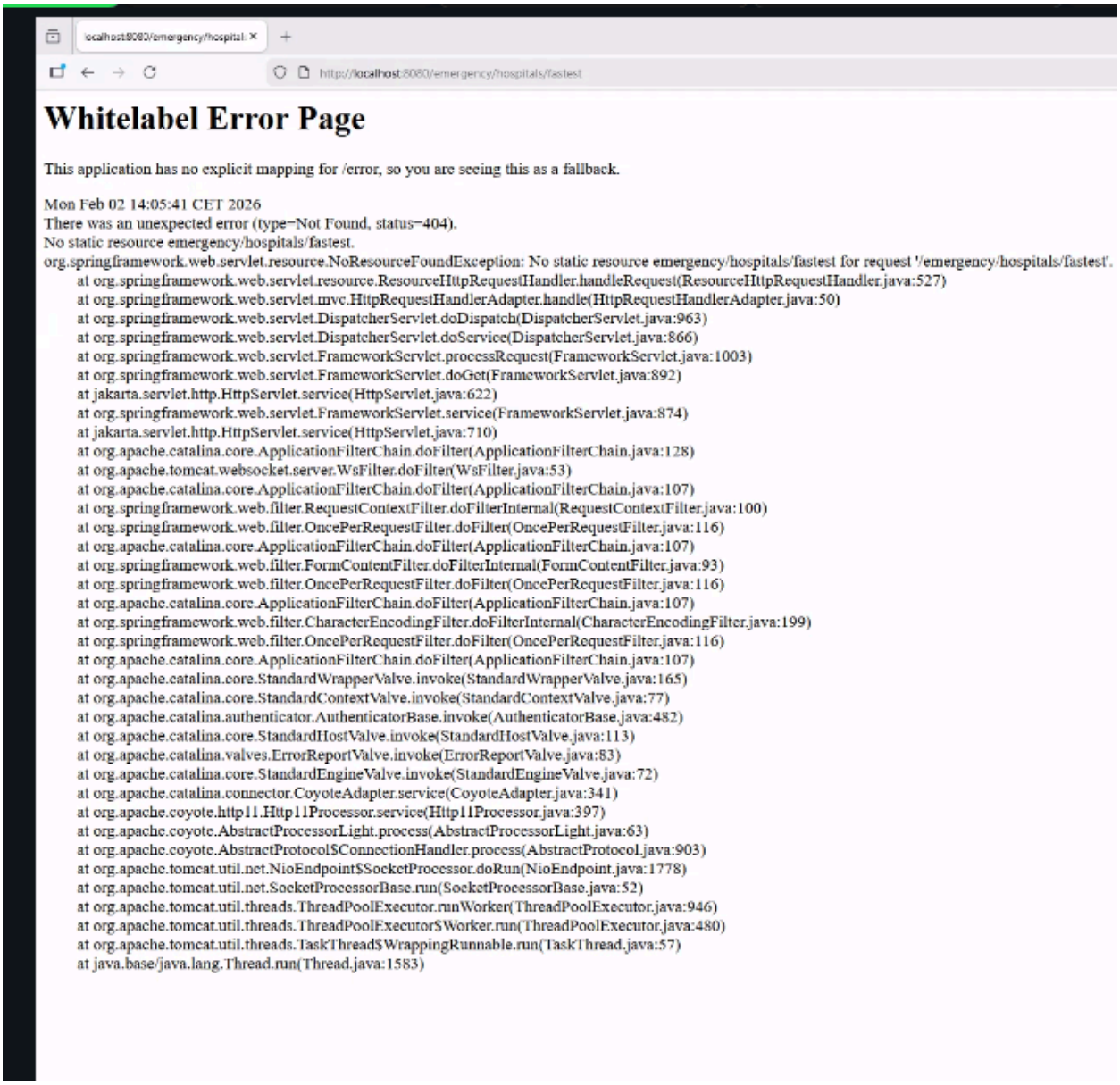
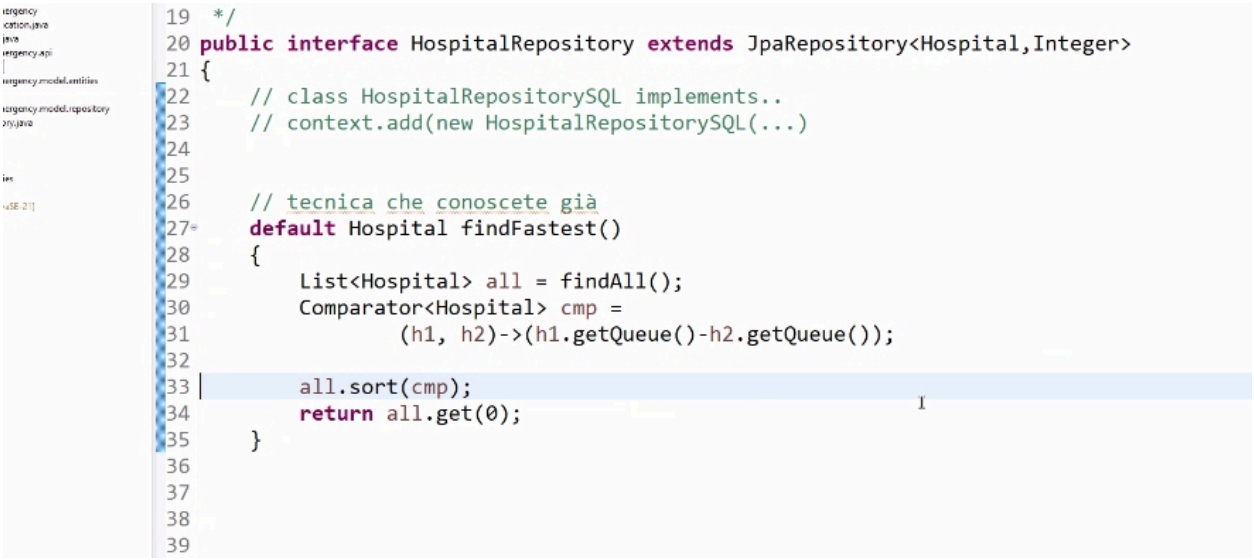
Appunti Pome



Spring Boot sta mostrando una **Whitelabel Error Page**, che è la pagina di errore generica di Spring Boot quando non trova un mapping appropriato per gestire una richiesta.

L'errore principale è **NoResourceFoundException con status 404**, che indica che Spring Boot non riesce a trovare una risorsa statica o un endpoint che corrisponda all'URL richiesto (`/emergency/hospitals/fastest`).



```java
 */
public interface HospitalRepository extends JpaRepository<Hospital,Integer>
{
    // class HospitalRepositorySQL implements..
    // context.add(new HospitalRepositorySQL(...)


    // tecnica che conoscete già
    default Hospital findFastest()
    {
        List<Hospital> all = findAll();
        Comparator<Hospital> cmp =
                (h1, h2)->(h1.getQueue()-h2.getQueue());

        all.sort(cmp);
        return all.get(0);
    }
```

Il metodo `findFastest()` che hai nel repository è un **metodo di business logic** che non è e non deve essere mappato direttamente. In Spring Boot, solo i **controller** hanno metodi mappati agli endpoint HTTP

Repository → Service (opzionale) → Controller → Endpoint HTTP

Mappare significa creare un **collegamento diretto** tra un **indirizzo web (URL)** e un **metodo Java** nel tuo controller

Spring Data JPA naming convention

```
find + First + By + OrderBy + Queue + Asc
  ↓     ↓     ↓     ↓      ↓     ↓
SELECT  LIMIT   (no   ORDER   campo   ASC
       1     filter)  BY     queue
```

Spring analizza il nome `findFirstByOrderByQueueAsc()` parola per parola:
Regole della naming convention

Spring Data JPA segue regole specifiche per interpretare i nomi dei metodi:

1. **Prefisso**: `find`, `get`, `read`, `query`, `search`, `stream` (tutti equivalenti per SELECT)
2. **Limitatori**: `First`, `Top`, `Top3`, `First5` (per limitare il numero di risultati)
3. **Filtri**: dopo `By` puoi aggiungere condizioni come `ByName`, `ByQueueLessThan`, `ByIdAndName`
4. **Ordinamento**: `OrderBy` seguito dal nome del campo e `Asc` o `Desc`

Spring **deduce** automaticamente la query SQL corretta per ciascuno di questi metodi basandosi esclusivamente sul nome.

Keyword	Esempio	JPQL generato
`And`	`findByNameAndAge`	`WHERE x.name = ?1 AND x.age = ?2` java4coding
`Or`	`findByNameOrAge`	`WHERE x.name = ?1 OR x.age = ?2` java4coding
`Is`, `Equals`	`findByNameIs`	`WHERE x.name = ?1` java4coding
`Between`	`findByAgeBetween`	`WHERE x.age BETWEEN ?1 AND ?2` java4coding
`LessThan`	`findByAgeLessThan`	`WHERE x.age < ?1` java4coding
`LessThanEqual`	`findByAgeLessThanEqual`	`WHERE x.age <= ?1` java4coding
`GreaterThan`	`findByAgeGreaterThan`	`WHERE x.age > ?1` java4coding
`GreaterThanEqual`	`findByAgeGreaterThanEqual`	`WHERE x.age >= ?1` java4coding
`Before`	`findByStartDateBefore`	`WHERE x.startDate < ?1` java4coding
`After`	`findByStartDateAfter`	`WHERE x.startDate > ?1` java4coding
`IsNull`, `Null`	`findByAgeIsNull`	`WHERE x.age IS NULL` java4coding
`IsNotNull`, `NotNull`	`findByAgeIsNotNull`	`WHERE x.age IS NOT NULL` java4coding
`Like`	`findByNameLike`	`WHERE x.name LIKE ?1` java4coding
`NotLike`	`findByNameNotLike`	`WHERE x.name NOT LIKE ?1` java4coding
`StartingWith`	`findByNameStartingWith`	`WHERE x.name LIKE ?1%` java4coding
`EndingWith`	`findByNameEndingWith`	`WHERE x.name LIKE %?1` java4coding
`Containing`	`findByNameContaining`	`WHERE x.name LIKE %?1%` java4coding
`Not`	`findByAgeNot`	`WHERE x.age <> ?1` java4coding

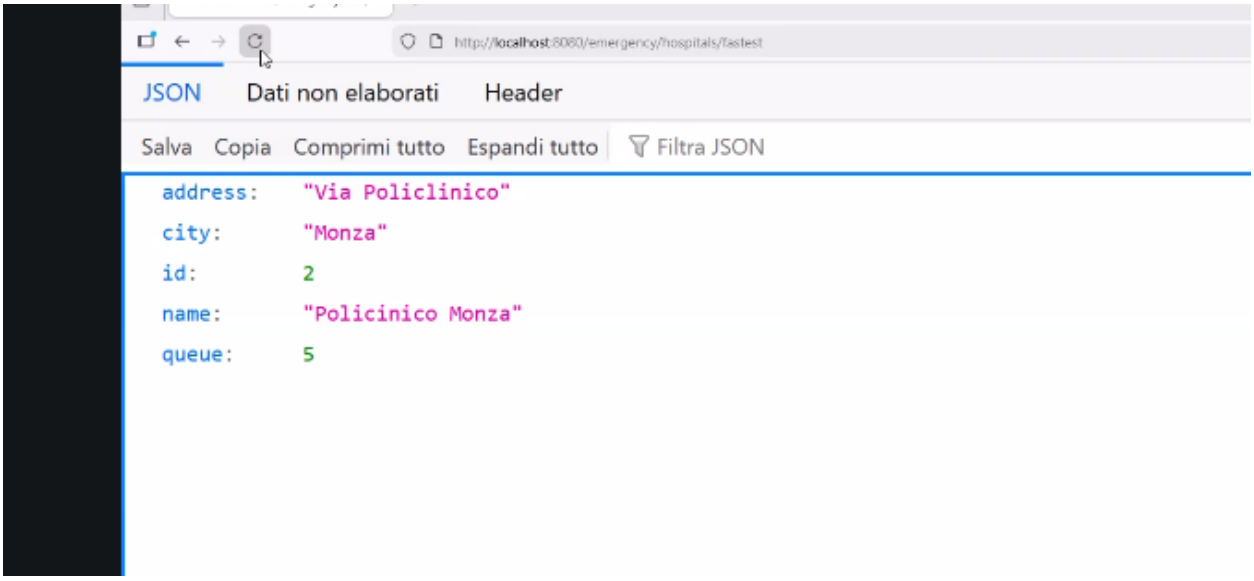
Keyword	Esempio	JPQL generato
`In`	`findByAgeIn(Collection<Age>)`	`WHERE x.age IN ?1` java4coding
`NotIn`	`findByAgeNotIn(Collection<Age>)`	`WHERE x.age NOT IN ?1` java4coding
`True`	`findByActiveTrue`	`WHERE x.active = true` java4coding
`False`	`findByActiveFalse`	`WHERE x.active = false` java4coding
`IgnoreCase`	`findByNameIgnoreCase`	`WHERE UPPER(x.name) = UPPER(?1)` java4coding

Modificatori speciali

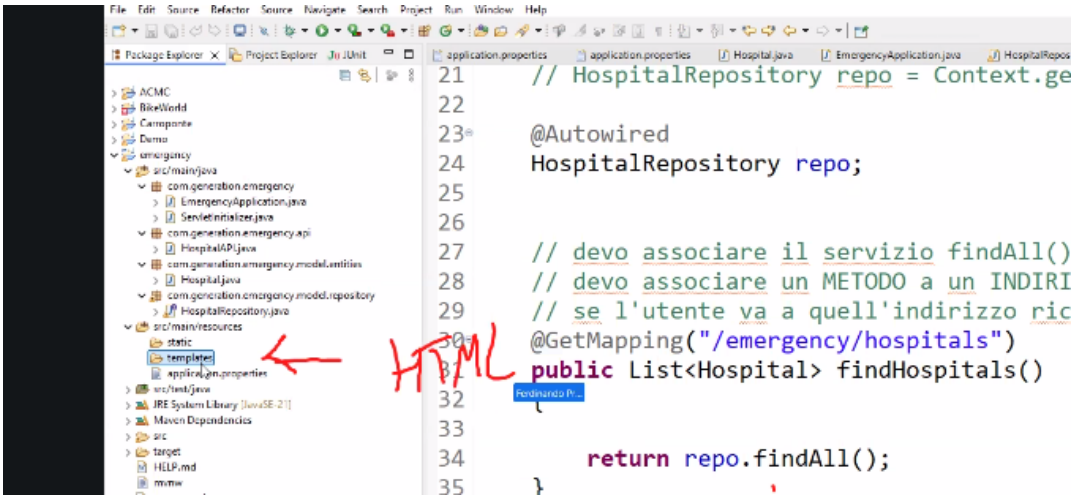
Oltre a `First` e `Top`, ci sono altre opzioni:

- `Distinct`: `findDistinctByName` - rimuove duplicati
- `Count`: `countByAge` - conta i risultati invece di restituirli
- `Delete`, `Remove`: `deleteByAge` - elimina i record
- `Exists`: `existsByEmail` - verifica se esistono record

Andare ad un indirizzo significa eseguire un metodo → quello che vediamo è il ritorno di quel metodo



Le pagine html devono essere nella cartella template



Le pagine HTML ora caricheranno dati da stampare, mangeranno dati. caricheranno dati dalle API.

```
    // devo associare il servizio findAll() a un INDIRIZZO WEB
    // devo associare un METODO a un INDIRIZZO
    // se l'utente va a quell'indirizzo riceve la risposta di QUESTO MET
    @GetMapping("/emergency/api/hospitals")
    public List<Hospital> findHospitals()
    {

        return repo.findAll();
    }

    @GetMapping("/emergency/api/hospitals/fastest")
    public Hospital findFastest()
    {
        // return repo.findFastest();
        return repo.findFirstByOrderByQueueAsc();
    }

```

cambiato indirizzo perche le api devono essere distinte dalle pagine

```java
    // devo associare un METODO a un INDIRIZZO
    // se l'utente va a quell'indirizzo riceve la risposta di QUESTO MET
    @GetMapping("/emergency/api/hospitals")
    public List<Hospital> findHospitals()
    {

        return repo.findAll();
    }

    @GetMapping("/emergency/api/hospitals/fastest")
    public Hospital findFastest()
    {
        // return repo.findFastest();
        return repo.findFirstByOrderByQueueAsc();
    }

```

Creo una pagina HTML che sarà un client per la API → OperatorHome

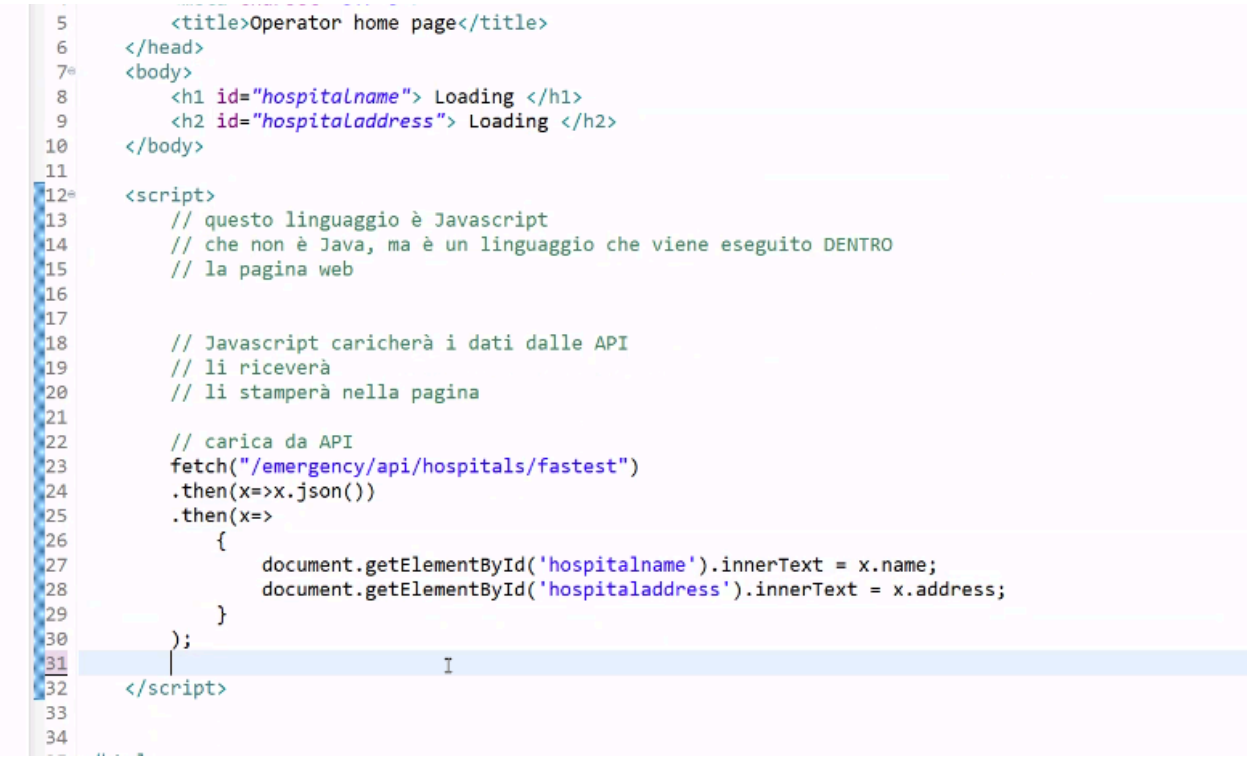
Operatore (angelo che in strada sul posto dell'incidente e deve decidere dove portare il paziente)

```html
<!DOCTYPE html>
    <html>
    <head>
        <meta charset="UTF-8">
        <title>Operator home page</title>
    </head>
    <body>
        <h1 id="hospitalname"> Loading </h1>
        <h2 id="hospitaladdress"> Loading </h2>
    </body>
</html>
```

HTML deve caricre i dati dalla API, ovvero da /emergency/api/hospitals/fastest

Nel body abbiamo due elementi con `id` specifici: `hospitalname` e `hospitaladdress`. Questi `id` sono fondamentali perché funzionano come **etichette identificative** che JavaScript userà per trovare e modificare quegli elementi specifici.

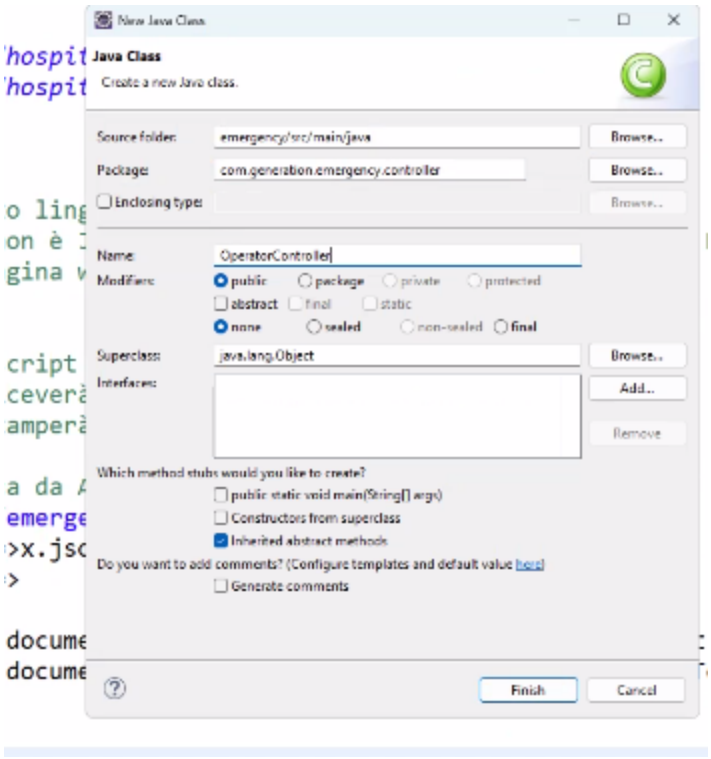
1. **Carica**: `fetch()` fa la richiesta HTTP alla tua API Spring Boot
2. **Riceve**: `.then(x ⇒ x.json())` converte la risposta JSON in oggetto JavaScript
3. **Stampa**: nella pagina usando DOM manipulation per sostituire "Loading..." con i dati reali

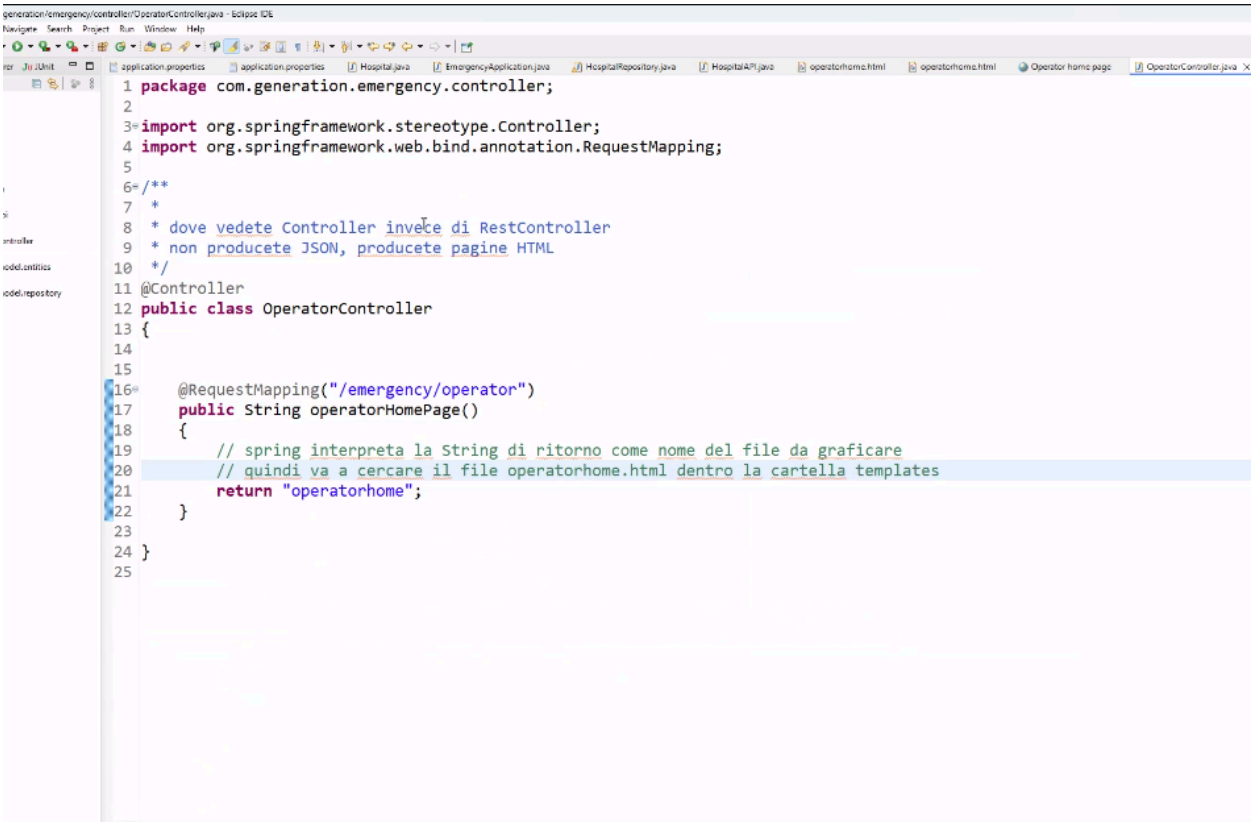


```
5       <title>Operator home page</title>
6   </head>
7   <body>
8       <h1 id="hospitalname"> Loading </h1>
9       <h2 id="hospitaladdress"> Loading </h2>
10  </body>
11
12  <script>
13      // questo linguaggio è Javascript
14      // che non è Java, ma è un linguaggio che viene eseguito DENTRO
15      // la pagina web
16
17
18      // Javascript caricherà i dati dalle API
19      // li riceverà
20      // li stamperà nella pagina
21
22      // carica da API
23      fetch("/emergency/api/hospitals/fastest")
24      .then(x=>x.json())
25      .then(x=>
26          {
27              document.getElementById('hospitalname').innerText = x.name;
28              document.getElementById('hospitaladdress').innerText = x.address;
29          }
30      );
31
32  </script>
33
34
```

Ora devo scrivere un controller per mappare questo file html ad un indirizzo e poi lui si prenderà i dati

Prima c'era un controller che produceva dati grazzi, ora scriveremo un controller che butterà fuori una oagina web



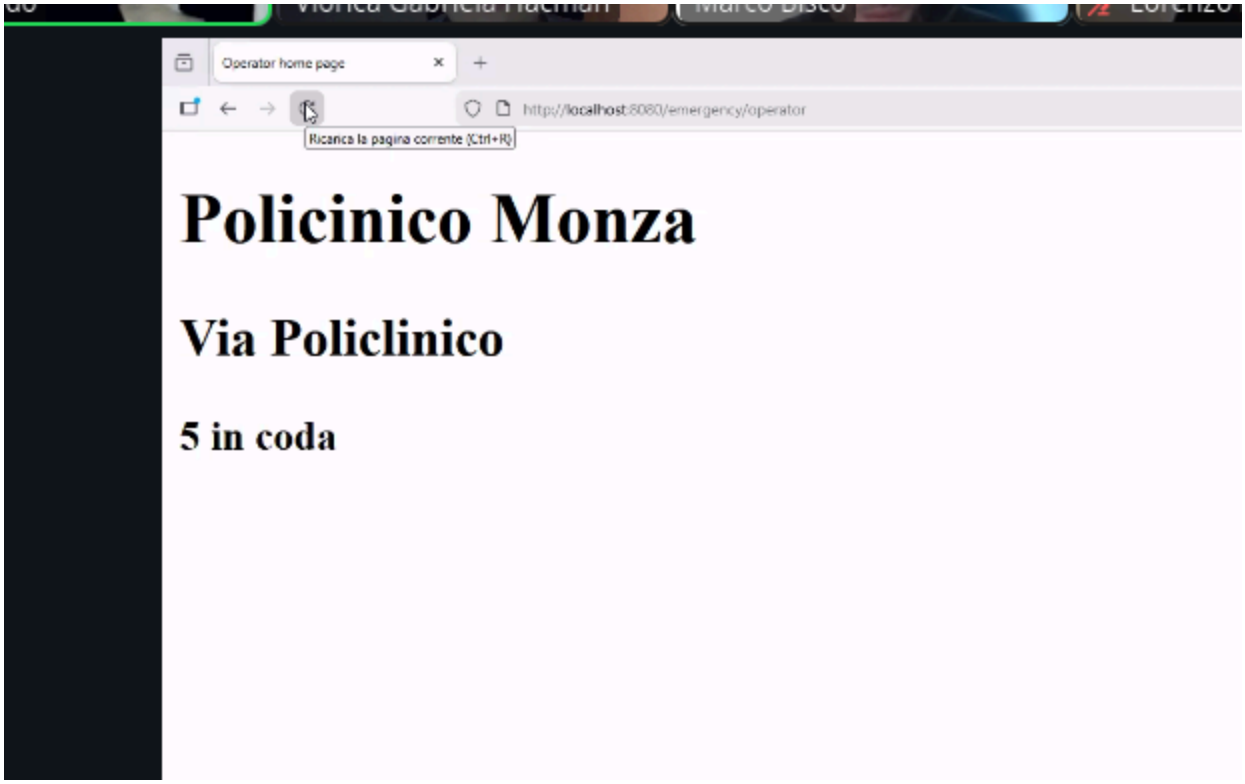


```java
package com.generation.emergency.controller;

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

/**
 *
 * dove vedete Controller invece di RestController
 * non producete JSON, producete pagine HTML
 */
@Controller
public class OperatorController
{


    @RequestMapping("/emergency/operator")
    public String operatorHomePage()
    {
        // spring interpreta la String di ritorno come nome del file da graficare
        // quindi va a cercare il file operatorhome.html dentro la cartella templates
        return "operatorhome";
    }

}
```

@Controller invece di @RestController

La classe è annotata con `@Controller` (non `@RestController`). Questo significa che i metodi **non restituiscono JSON**, ma **nomi di viste** (pagine HTML) che verranno renderizzate dal motore di template di Spring, di solito Thymeleaf. In pratica, questo controller è dedicato a servire l'interfaccia grafica (la pagina `operatorhome.html`), non i dati.

Alcuni controller producono dati grezzi, altri producono pagine web



Mappatura dei controller

Solo i controller con annotazioni come `@Controller` o `@RestController` sono mappati agli endpoint HTTP. La differenza fondamentale è che `@RestController` restituisce dati serializzati in JSON mentre `@Controller` restituisce nomi di viste HTML. Un metodo annotato con `@GetMapping("/emergency/hospitals/fastest")` viene eseguito automaticamente quando arriva una richiesta GET a quell'URL specifico.

Flusso client-server OperatorHome

Il controller `OperatorController` mappato su `/emergency/operator` restituisce la stringa "operatorhome", che Spring Boot interpreta come nome logico di una vista. Il

ViewResolver cerca automaticamente il file `operatorhome.html` nella cartella `src/main/resources/templates/` e lo serve al browser.

Questa pagina HTML contiene elementi con id specifici come `hospitalname` e `hospitaladdress` inizialmente impostati su "Loading". Un blocco JavaScript eseguito nel browser utilizza la Fetch API per chiamare l'endpoint REST `/emergency/api/hospitals/fastest` del controller `@RestController`.

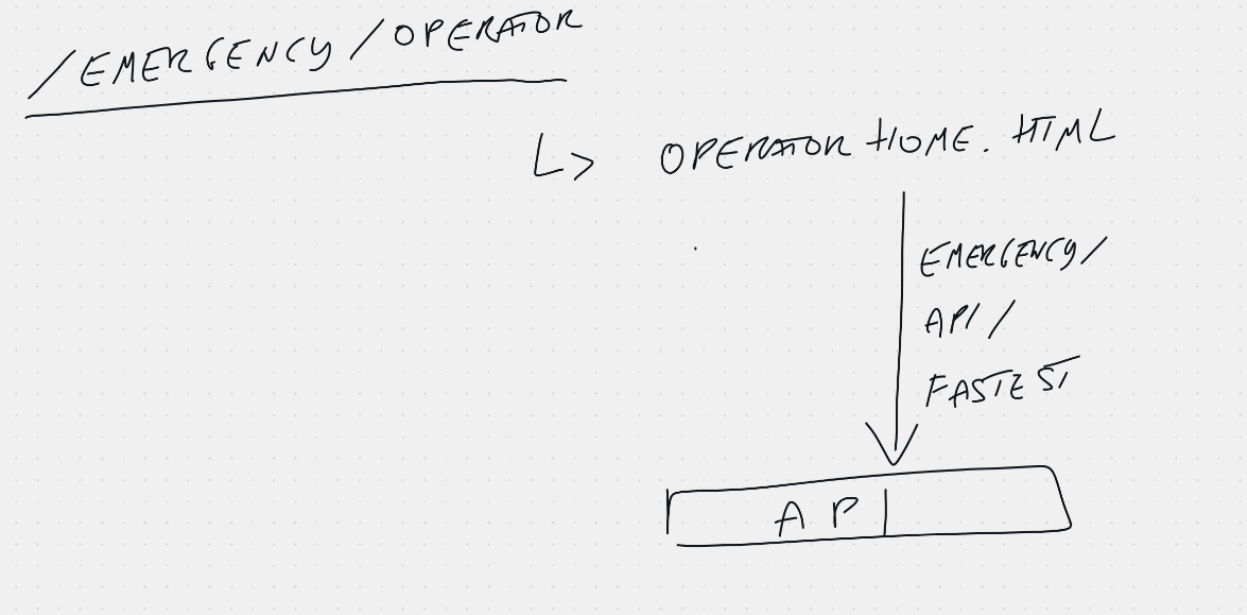
Fetch API e manipolazione DOM

La chiamata `fetch("/emergency/api/hospitals/fastest").then(x => x.json()).then(x => {...})` funziona in tre fasi. Prima `fetch()` invia la richiesta HTTP asincrona e restituisce una Promise. Il primo `.then(x => x.json())` riceve la risposta HTTP grezza e converte il body JSON in un oggetto JavaScript contenente i dati dell'ospedale.

Il secondo `.then(x => {...})` riceve questo oggetto e utilizza `document.getElementById('hospitalname').innerText = x.name` per trovare l'elemento HTML con quell'id e sostituire il testo "Loading" con il nome effettivo dell'ospedale. La stessa operazione viene ripetuta per l'indirizzo. Il browser aggiorna immediatamente la visualizzazione sostituendo i placeholder con i dati reali.

Risultato finale

Quando l'operatore accede all'URL `/emergency/operator`, il controller MVC serve la pagina HTML iniziale che contiene JavaScript. Questo codice si esegue automaticamente caricando i dati dall'API REST e aggiornando dinamicamente la pagina. Il risultato visibile è il nome dell'ospedale consigliato, l'indirizzo e il numero di pazienti in coda, calcolati dal backend e visualizzati dal frontend senza ricaricare la pagina.



Flusso client-server OperatorHome

Il controller `OperatorController` mappato su `/emergency/operator` restituisce la stringa "operatorhome", che Spring Boot interpreta come nome logico di una vista. Il ViewResolver cerca automaticamente il file `operatorhome.html` nella cartella `src/main/resources/templates/` e lo serve al browser.

Questa pagina HTML contiene elementi con id specifici come `hospitalname` e `hospitaladdress` inizialmente impostati su "Loading". Un blocco JavaScript eseguito nel browser utilizza la Fetch API per chiamare l'endpoint REST `/emergency/api/hospitals/fastest` del controller `@RestController`.

Fetch API e manipolazione DOM

La chiamata `fetch("/emergency/api/hospitals/fastest").then(x ⇒ x.json()).then(x ⇒ {...})` funziona in tre fasi. Prima `fetch()` invia la richiesta HTTP asincrona e restituisce una Promise. Il primo `.then(x ⇒ x.json())` riceve la risposta HTTP grezza e converte il body JSON in un oggetto JavaScript contenente i dati dell'ospedale.

Il secondo `.then(x ⇒ {...})` riceve questo oggetto e utilizza `document.getElementById('hospitalname').innerText = x.name` per trovare l'elemento HTML con quell'id e sostituire il testo "Loading" con il nome effettivo dell'ospedale. La stessa

operazione viene ripetuta per l'indirizzo. Il browser aggiorna immediatamente la visualizzazione sostituendo i placeholder con i dati reali.

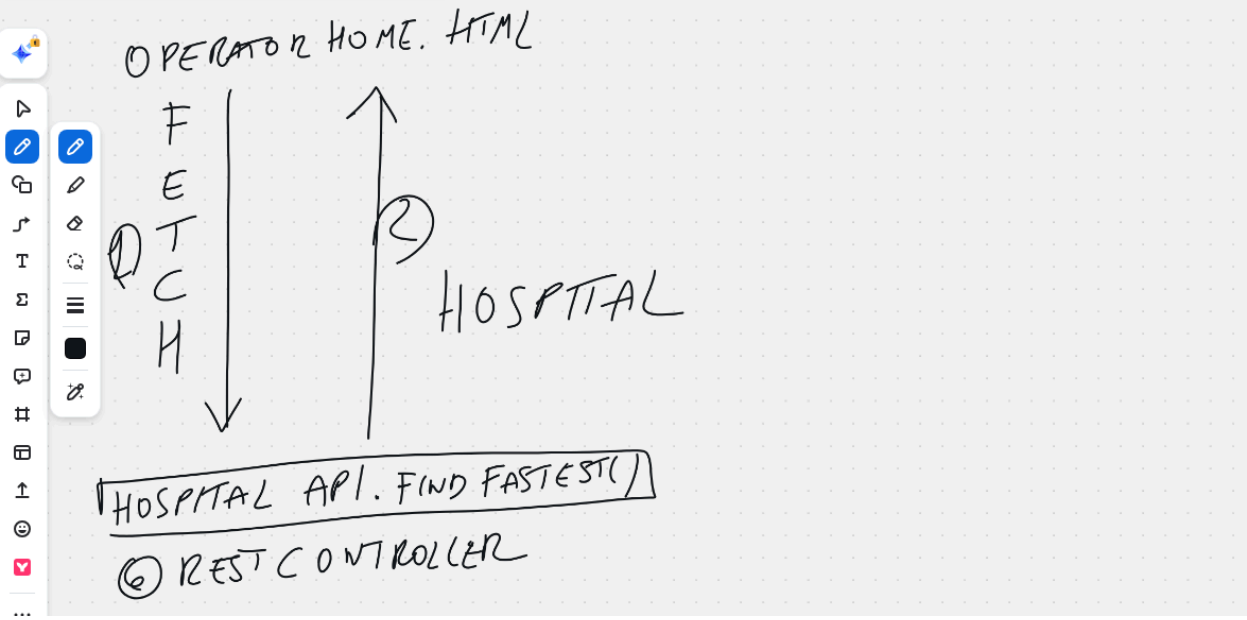
Flusso completo frontend-backend

Quando si carica un sito web si ricevono oppure pagine web complete oppure dati grezzi in formato JSON. La prima richiesta dell'operatore va all'indirizzo `/emergency/operator`, mappato a un controller normale annotato con `@Controller` che restituisce il nome del template `operatorhome.html`. Spring Boot legge questo file dalla cartella templates e lo invia al browser come pagina web completa.

Il file HTML contiene JavaScript che fa una seconda richiesta verso `/emergency/api/hospitals/fastest`, gestita da un controller `@RestController` che fornisce dati JSON puri. JavaScript riceve questi dati e popola dinamicamente la pagina sostituendo i placeholder con le informazioni reali dell'ospedale. L'HTML rappresenta il frontend visibile all'utente mentre le API costituiscono il backend fornitore di servizi e dati.

Risultato finale

Quando l'operatore accede all'URL `/emergency/operator`, il controller MVC serve la pagina HTML iniziale che contiene JavaScript. Questo codice si esegue automaticamente caricando i dati dall'API REST e aggiornando dinamicamente la pagina. Il risultato visibile è il nome dell'ospedale consigliato, l'indirizzo e il numero di pazienti in coda, calcolati dal backend e visualizzati dal frontend senza ricaricare la pagina.



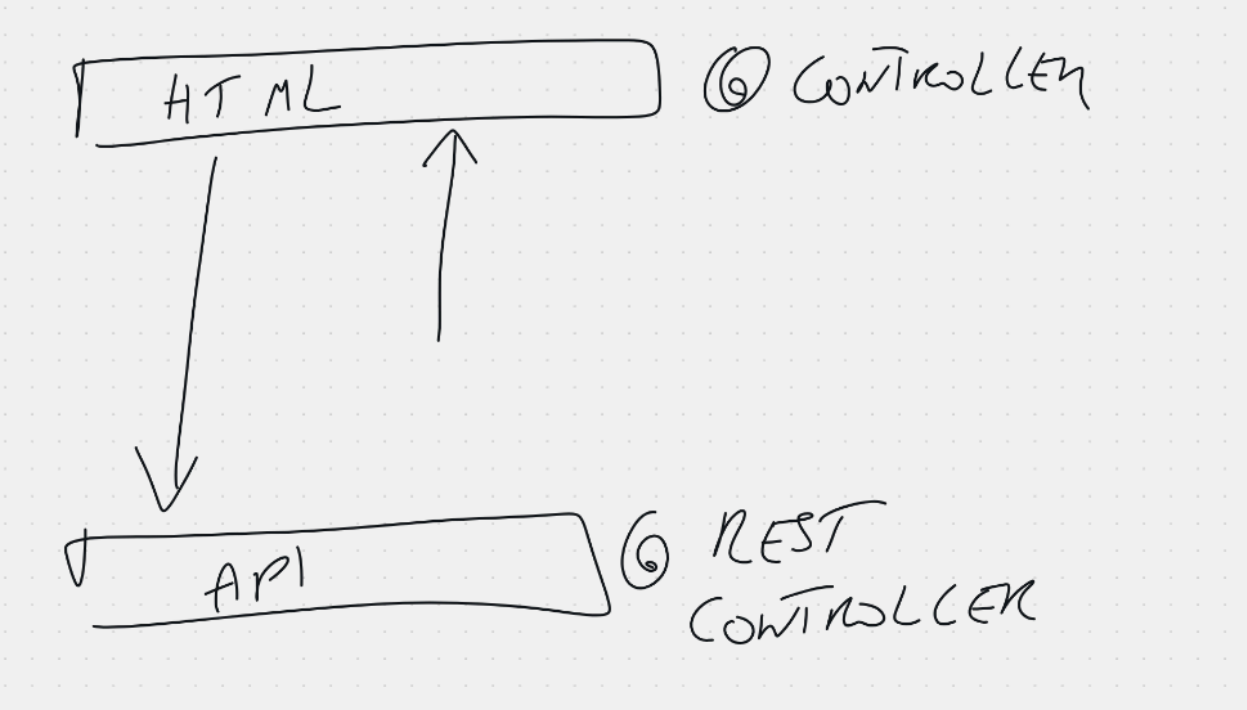
All'interno di questa pagina, il codice JavaScript esegue un'operazione `fetch` che significa "prendere dati" dall'indirizzo API `/emergency/api/hospitals/fastest`.

Questa chiamata fetch va verso il controller `@RestController` che espone l'endpoint REST specifico. Il controller REST chiama il metodo `findFastest()` del repository `HospitalRepository`, riceve l'oggetto Hospital contenente i dati dell'ospedale più veloce e lo serializza automaticamente in formato JSON.

HTML e JavaScript costituiscono il frontend, ovvero la parte dell'applicazione che gira nel browser dell'utente e consuma i servizi. Rappresentano l'interfaccia che l'operatore vede e con cui interagisce, richiedendo dati tramite chiamate fetch.

Le API rappresentano il backend, ovvero la parte del sistema che gira sul server e fornisce i servizi. È il fornitore di dati e logica di business, accessibile tramite gli endpoint REST gestiti dai controller annotati con `@RestController`.

Le frecce bidirezionali mostrano il flusso di comunicazione: il frontend invia richieste HTTP verso il backend e riceve in risposta dati JSON strutturati. Questa separazione tra frontend consumatore e backend fornitore è il pattern standard delle applicazioni web contemporanee.

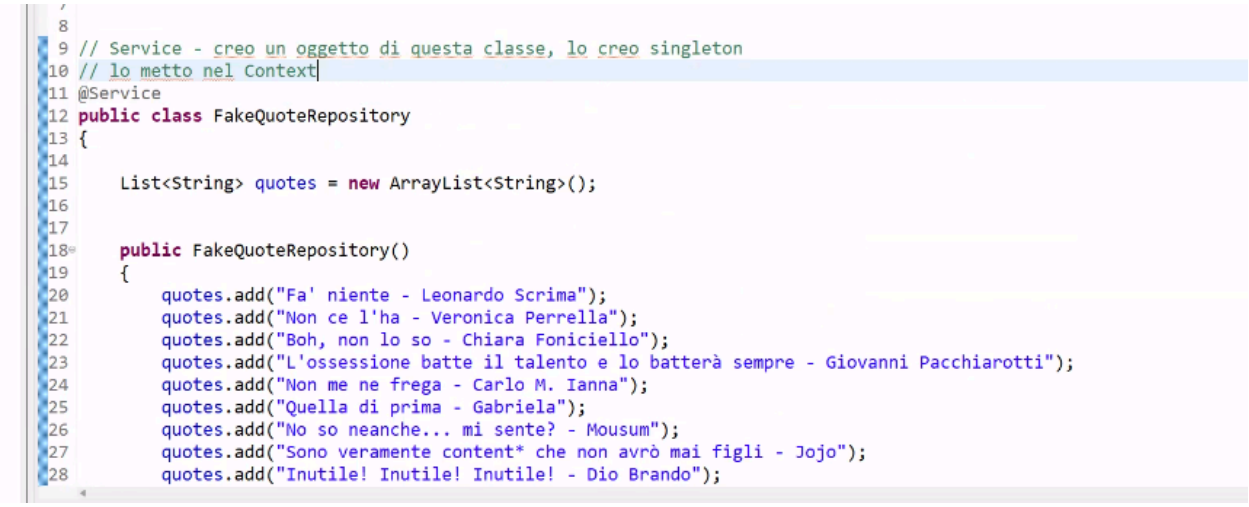


Una pagina web è composta da molteplici file HTML forniti dai controller annotati con `@Controller`, che mappano gli URL alle viste nella cartella templates.

La freccia che sale dal blocco HTML verso il Controller mostra la prima richiesta HTTP: quando l'utente digita `/emergency/operator`, il Controller restituisce il nome del template HTML che Spring Boot serve al browser.

Successivamente il JavaScript contenuto nell'HTML fa una seconda richiesta verso il blocco API, indicato dalla freccia discendente. Questa richiesta va al RestController che gestisce gli endpoint REST e fornisce dati strutturati in formato JSON.

I due tipi di controller hanno responsabilità distinte ma complementari: il Controller gestisce la presentazione (HTML frontend), mentre il RestController gestisce i dati (API backend). Insieme formano l'architettura completa dell'applicazione web.



```java
// Service - creo un oggetto di questa classe, lo creo singleton
// lo metto nel Context
@Service
public class FakeQuoteRepository
{

    List<String> quotes = new ArrayList<String>();


    public FakeQuoteRepository()
    {
        quotes.add("Fa' niente - Leonardo Scrima");
        quotes.add("Non ce l'ha - Veronica Perrella");
        quotes.add("Boh, non lo so - Chiara Foniciello");
        quotes.add("L'ossessione batte il talento e lo batterà sempre - Giovanni Pacchiarotti");
        quotes.add("Non me ne frega - Carlo M. Ianna");
        quotes.add("Quella di prima - Gabriela");
        quotes.add("No so neanche... mi sente? - Mousum");
        quotes.add("Sono veramente content* che non avrò mai figli - Jojo");
        quotes.add("Inutile! Inutile! Inutile! - Dio Brando");
```

L'annotazione `@Service` serve a indicare che una classe appartiene allo strato di servizio dell'applicazione, ovvero contiene la logica di business del dominio specifico. Tecnicamente è una specializzazione dell'annotazione `@Component`, quindi viene rilevata automaticamente dalla scansione del classpath di Spring durante l'avvio dell'applicazione.

Spring crea un'istanza singleton di questa classe e la registra come bean gestito nel container IoC (Inversion of Control), rendendola disponibile per l'iniezione delle dipendenze tramite `@Autowired` in altri componenti come controller o repository. Nel caso della classe `QueueRepository`, l'annotazione `@Service` permette ai controller REST di iniettare questa classe e utilizzare il metodo `list()` per ottenere le informazioni sulle code degli ospedali.

L'annotazione `@Autowired` è il meccanismo di **Dependency Injection** di Spring che permette di iniettare automaticamente le dipendenze necessarie in un controller, servizio o qualsiasi altro componente gestito dal container IoC.