



UNIVERSITY
OF TRENTO

Dipartimento di Ingegneria e
Scienza dell'Informazione

Progetto:

Progetto ingegneria del software: farmacia online e gestione dell'assunzione dei farmaci

Titolo del documento:

D4

Gruppo

Document Info

Doc. Name	<i>D4-Farmacia online_SviluppoApi</i>	Doc. Number	SR5
Description	Documento riguardante la parte di Backend, Frontend e Api		

Scopo del documento	3
1. User Flows	4
2. Application Implementation and Documentation	8
2.1. Project Structure	8
2.2. Project Dependencies	9
2.3. Project Data or DB	10
2.4. Project APIs	14
2.4.1. Resources Extraction from the Class Diagram	14
2.4.2. Resources Models	16
2.5. Sviluppo API	18
3. API documentation	31
4. FrontEnd Implementation	35
5. GitHub Repository and Deployment Info	43
6. Testing	44

Scopo del documento

- Introduzione al Progetto: Questo documento fornisce un quadro completo per lo sviluppo di una componente chiave dell'applicazione FarmaciaCGZ. In particolare, si concentra sulla realizzazione dei servizi dedicati alla gestione delle prescrizioni e vendita dei farmaci, e alla prenotazione di visite all'interno dell'ambito dell'applicazione FarmaciaCGZ.
- User Flow e Ruolo delle API per la Gestione degli utenti, delle Farmacie e Medici: Il documento procede con la presentazione delle API necessarie per supportare le operazioni di visualizzazione, inserimento e eliminazione di utenti, farmaci, acquisti, visite, prescrizioni, ecc.. all'interno dell'applicazione FarmaciaCGZ. Queste API sono illustrate tramite il concetto di API Model e il Modello delle risorse, che forniscono una panoramica chiara delle funzionalità offerte.
- Documentazione delle API: Per ciascuna API sviluppata, oltre alla descrizione delle funzionalità offerte, il documento fornisce dettagliate informazioni sulla documentazione relativa.
- Git Repository e Deployment: La sezione finale del documento è dedicata alle informazioni relative al repository Git utilizzato per il progetto, fornendo una chiara traccia di versione e collaborazione nello sviluppo.

1. User Flows

In questa sezione del documento riportiamo gli “user flows” per i ruoli dell'utente, del farmacista e del medico del nostro progetto.

Figura 1 descrive lo user flow dell'utente relativo alla gestione della ricerca dei farmaci, delle farmacie e dei medici, l'acquisto dei farmaci, la prenotazione dei visite medici, la registrazione e l'autenticazione sul sito. L'utente può effettuare navigare sul sito anche senza registrazione ed autenticazione. Però per fare l'acquisto dei farmaci, prenotare la visita medica l'utente deve entrare nel sistema e quindi bisogna essere registrato. L'utente anonimo può iniziare dalla ricerca dei farmaci o dei medici passando all'autenticazione dopo o autenticarsi appena entrato sul sito. Dalla ricerca delle farmacie l'utente può anche passare alla ricerca dei farmaci presenti nel catalogo della farmacia. Dopo la ricerca del farmaco l'utente può aggiungerlo al carrello iniziando la procedura dell'acquisto cioè scegliere la quantità del prodotto, il metodo di spedizione e effettuare il pagamento. L'utente può anche comunicare con la farmacia riguardo l'assistenza e reso ed anche visualizzare la propria scheda dei farmaci (l'orario del consumo dei suoi farmaci prescritti con dosaggi rispettivi). Dopo aver effettuato la ricerca dei medici l'utente autenticato può prenotare la visita o comunicare con il medico nel chat.

Figura 2 descrive lo user flow del farmacista relativo alla registrazione e l'autenticazione sul sito, registrazione della farmacia, aggiunta farmaci, gestione dei ordini. Per effettuare le operazioni sul sito il farmacista deve essere registrato ed autenticato.

Figura 3 descrive lo user flow del medico relativo alla registrazione e l'autenticazione sul sito, gestione delle visite e dell'assunzione dei farmaci dai pazienti, compilazione delle ricette e comunicazione con pazienti. Per effettuare le operazioni sul sito il medico deve essere registrato ed autenticato.

Una legenda che descrive i simboli usati nei user flow è anche presentata in Figura 1.

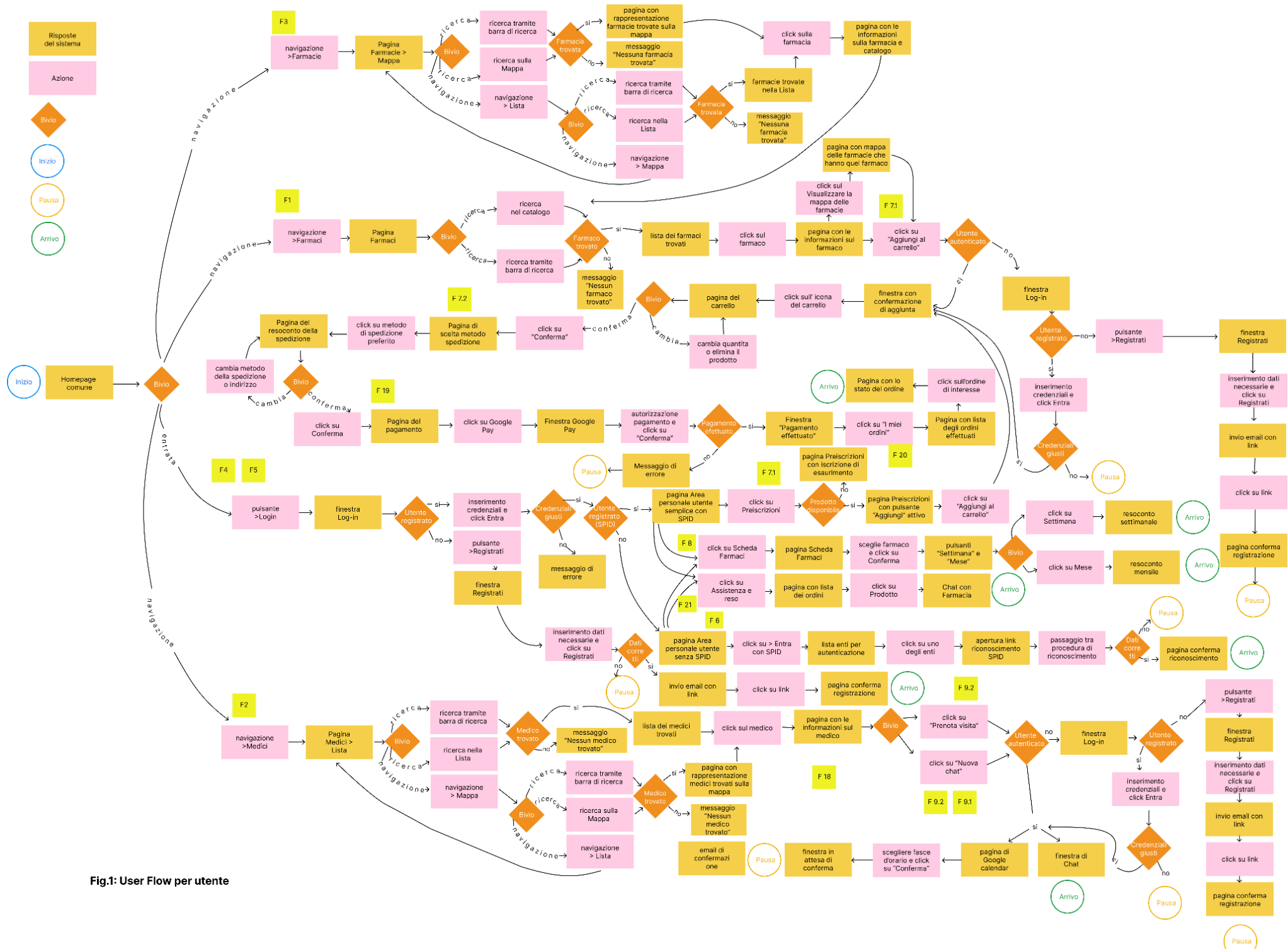


Fig.1: User Flow per utente

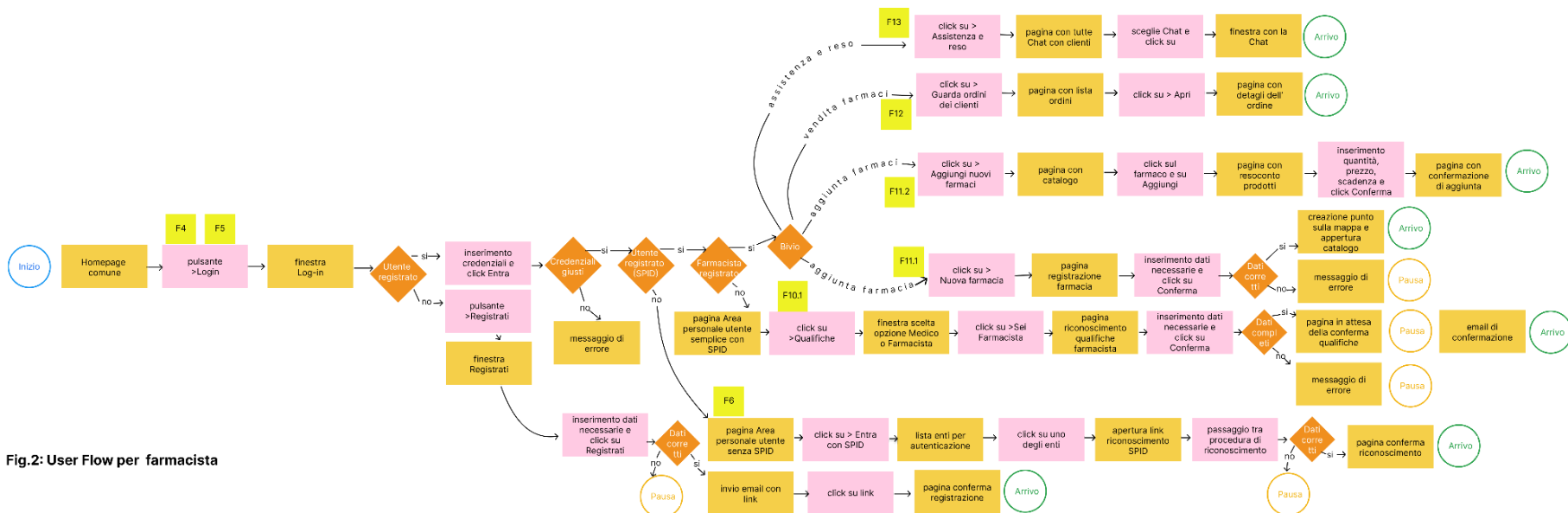


Fig.2: User Flow per farmacista

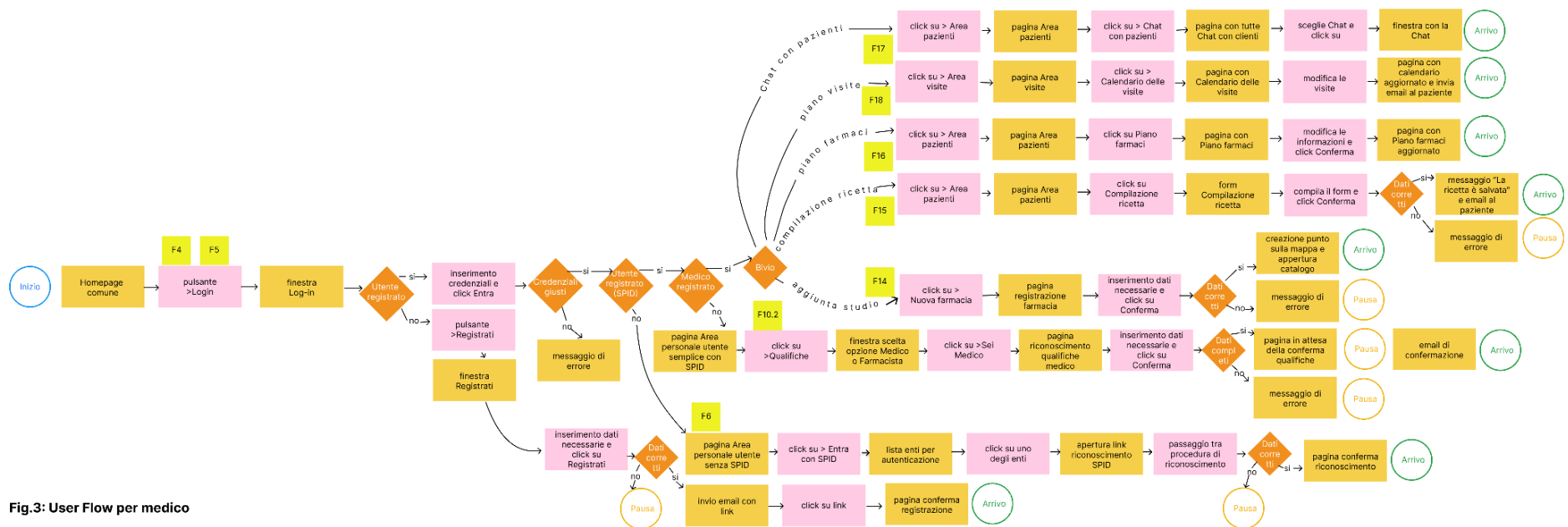


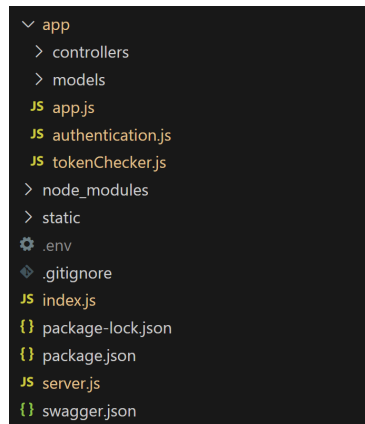
Fig.3: User Flow per medico

2. Application Implementation and Documentation

Nelle sezioni precedenti abbiamo identificato le varie features che devono essere implementate per la nostra applicazione con un'idea di come il nostro utente finale puo' utilizzarle nel suo flusso applicativo. L'applicazione è stata sviluppata utilizzando NodeJS. Per la gestione dei dati abbiamo utilizzato MongoDB.

2.1. Project Structure

La struttura del progetto è composta in questo modo



Nella root del progetto si trovano vari file:

- “index.js” è il file principale, serve a far partire l'intera applicazione
- “server.js” che esegue il server e la connessione al database
- due file package che contengono i package utilizzati
- swagger.json che mostra in formato json il funzionamento delle api
- .env è un file che contiene le credenziali di accesso e altre informazioni utili

Nella cartella principale sono contenute una serie di sottocartelle:

- App che contiene a sua volta delle sottocartelle, ha tutti i file utili alla definizione dei modelli e delle funzioni di base delle varie api (get, post, remove)


```
✓ app
  ✓ controllers
    JS acquisto.js
    JS chat.js
    JS farmaco.js
    JS luogo.js
    JS ricetta.js
    JS utente.js
    JS visita.js
  ✓ models
    JS acquisto.js
    JS chat.js
    JS farmaco.js
    JS luogo.js
    JS ricetta.js
    JS utente.js
    JS visita.js
  JS app.js
  JS authentication.js
  JS tokenChecker.js
```

- E static che contiene i file html delle pagine e i file javascript collegati

```
✓ static
  <> acquisto.html
  <> chat.html
  <> farmacie.html
  <> farmaco.html
  <> index_accesso.html
  <> index.html
  <> luogo.html
  <> medici.html
  <> pazienti.html
  <> ricetta.html
  JS script.js
  <> utente.html
  <> visite.html
```

2.2. Project Dependencies

I seguenti moduli node sono stati aggiunti ed utilizzati nel file package.json

```
"dependencies": {  
  "cors": "^2.8.5",  
  "dotenv": "^16.0.3",  
  "express": "^4.18.2",  
  "fetch": "^1.1.0",  
  "jsonwebtoken": "^9.0.2",  
  "map": "^1.0.1",  
  "mongoose": "^6.7.2",  
  "mongoose-express": "^0.0.1",  
  "source": "^0.0.3",  
  "source-map": "^0.7.4",  
  "swagger-jsdoc": "^5.0.1",  
  "swagger-ui-express": "^4.6.0"  
},  
"devDependencies": {  
  "jest": "^29.6.4",  
  "node-fetch": "^3.3.2",  
  "supertest": "^6.3.3"  
}
```

2.3. Project Data or DB

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
acquistos	1	124B	124B	20KB	1	20KB	20KB
chats	3	320B	107B	36KB	1	36KB	36KB
farmacos	1	249B	249B	36KB	1	36KB	36KB
luogos	2	409B	205B	36KB	1	36KB	36KB
ricettas	1	164B	164B	4KB	1	4KB	4KB
utentes	4	1.12KB	287B	36KB	1	36KB	36KB
visitas	1	84B	84B	36KB	1	36KB	36KB

Utenti e vari tipi:

Medico:

```
_id: ObjectId('64d8be1abb696b8a9f1b1f66')
name: "Michele"
surname: "Ghilardi"
year: 19832-02-05T00:00:00.000+00:00
CF: "hklfghigbn"
email: "michele.ghilardi@studenti.unitn.it"
password: "FarmaciaCGZ"
account_type: "Medico"
indirizzo: "via Caio 15"
SPID: true
titolo_di_studio: "Medicina"
biografia: "E' il migliore in quello che fa"
__v: 0
```

ClientePaziente

```
_id: ObjectId('64f05e8d6b7972ad375cd75d')
name: "Luca"
surname: "Ciullo"
year: 2002-02-11T00:00:00.000+00:00
CF: "hfgiklò"
email: "luca.ciulloi@studenti.unitn.it"
password: "FarmaciaCGZ"
account_type: "ClientePaziente"
indirizzo: "via Qualunque"
SPID: true
titolo_di_studio: ""
biografia: ""
__v: 0
```

Farmacista

```
_id: ObjectId('64f7282df3352bbd48920013')
name: "Anastasia"
surname: "Zyrianova"
year: 2002-11-01T23:00:00.000+00:00
CF: "CLLLC02b84849"
email: "anastasia.zyrianova@studenti.unitn.it"
password: "FarmaciaCGZ"
account_type: "Farmacista"
indirizzo: "via Verde 23"
SPID: true
titolo_di_studio: "Farmacia"
biografia: "Sono farmacista"
__v: 0
```

Due tipi di luoghi:
studio medico:

```
_id: ObjectId('64f728d5f3352bbd48920017')
nome: "Studio Medico"
indirizzo: "Corso 3 Novembre"
type: "Studio Medico"
distanza: 4
descrizione: "Qui si fanno dei controlli"
utenteId: "64d8be1abb696b8a9f1b1f66"
__v: 0
```

Farmacia:

```
_id: ObjectId('64f72869f3352bbd48920015')
nome: "Farmacia Trento"
indirizzo: "Corso 3 Novembre"
type: "Farmacia"
distanza: 4
descrizione: "Qui si vendono farmaci"
utenteId: "64f7282df3352bbd48920013"
__v: 0
```

Farmaco:

```
_id: ObjectId('64f72a03f3352bbd48920023')
name: "sfsfsf"
modalitauso: "Prendere 3 pastiglie al giorno"
foglio_illustrativo: "Il farmaco presenta questo principio attivo"
scadenza: 2025-06-09T00:00:00.000+00:00
prezzo: 15
quantita: 5
luogoId: "64f72869f3352bbd48920015"
__v: 0
```

Visita

```
_id: ObjectId('64f72aa1f3352bbd48920028')
data: 2022-10-31T23:00:00.000+00:00
utenteId: "64f05e8d6b7972ad375cd75d"
farmacoId: "64f72a03f3352bbd48920023"
__v: 0
```

Chat:

```
_id: ObjectId('64f72620f3352bbd48920005')
data: 2022-12-17T00:00:00.000+00:00
text_message: "Buonasera dottore, a che ora è disponibile per farmi una visita?"
__v: 0
```

```
_id: ObjectId('64f72674f3352bbd4892000f')
data: 2022-12-17T00:00:00.000+00:00
text_message: "Le andrebbe bene per il 06-07-2023 alle 10:15?"
__v: 0
```

```
_id: ObjectId('64f7264ef3352bbd4892000a')
data: 2022-12-17T00:00:00.000+00:00
text_message: "Va bene, perfetto"
__v: 0
```

Acquisto:

```
_id: ObjectId('64f726f6f3352bbd48920011')
data: 2023-06-07T00:00:00.000+00:00
utenteId: "64d8be1abb696b8a9f1b1f66"
__v: 0
```

Ricetta:

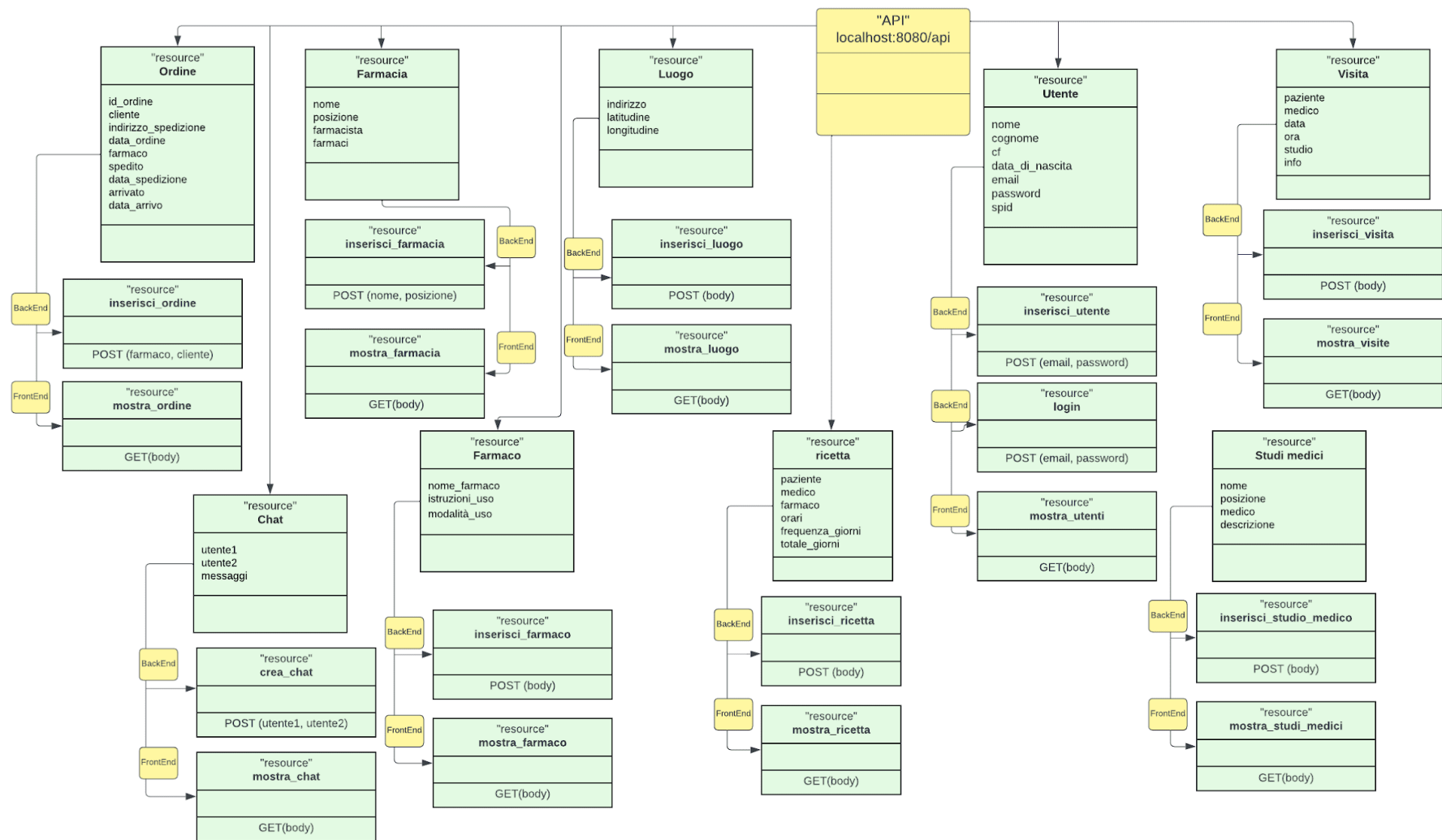
```
_id: ObjectId('64f72d91f1ad5458c6a3bcd6')
quantita: 5
dose: 3
periodo_inizio: 2023-10-31T23:00:00.000+00:00
periodo_fine: 2024-05-06T00:00:00.000+00:00
farmacoNome: "sfsfsf"
visitaId: "64f726f6f3352bbd48920011"
v: 0
```

2.4. Project APIs

2.4.1. Resources Extraction from the Class Diagram

In questa sezione del documento presente Il diagramma delle risorse che rappresenta le risorse estratte dal Diagramma delle Classi con le API che vengono sviluppati per ogni classe, il metodo che viene applicato e la lista dei parametri rispettivi.

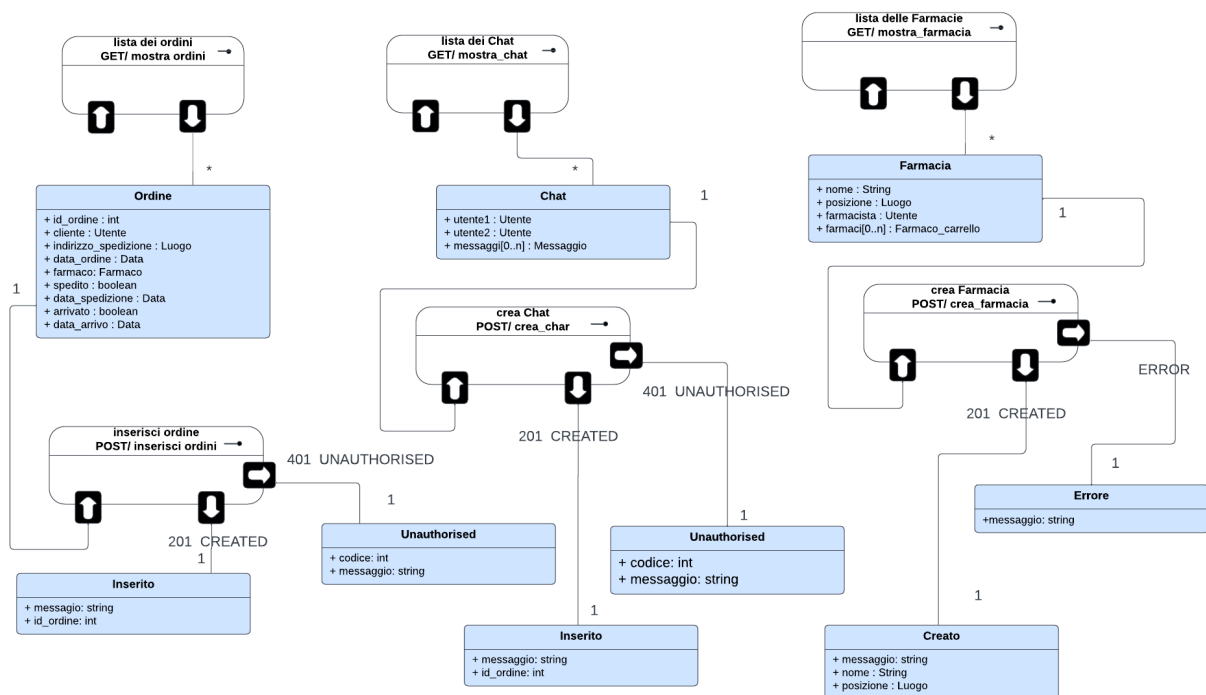
Si può vedere anche il collegamento tra le risorse. Per esempio, la risorsa “Ordine” è collegata con le risorse “inserisci_ordine” dal lato BackEnd e “mostra_ordine” dal lato FrontEnd, la risorsa “Utente” e collegata con le risorse “inserisci_utente”, “login” dal lato BackEnd e “mostra_utenti” dal lato FrontEnd.

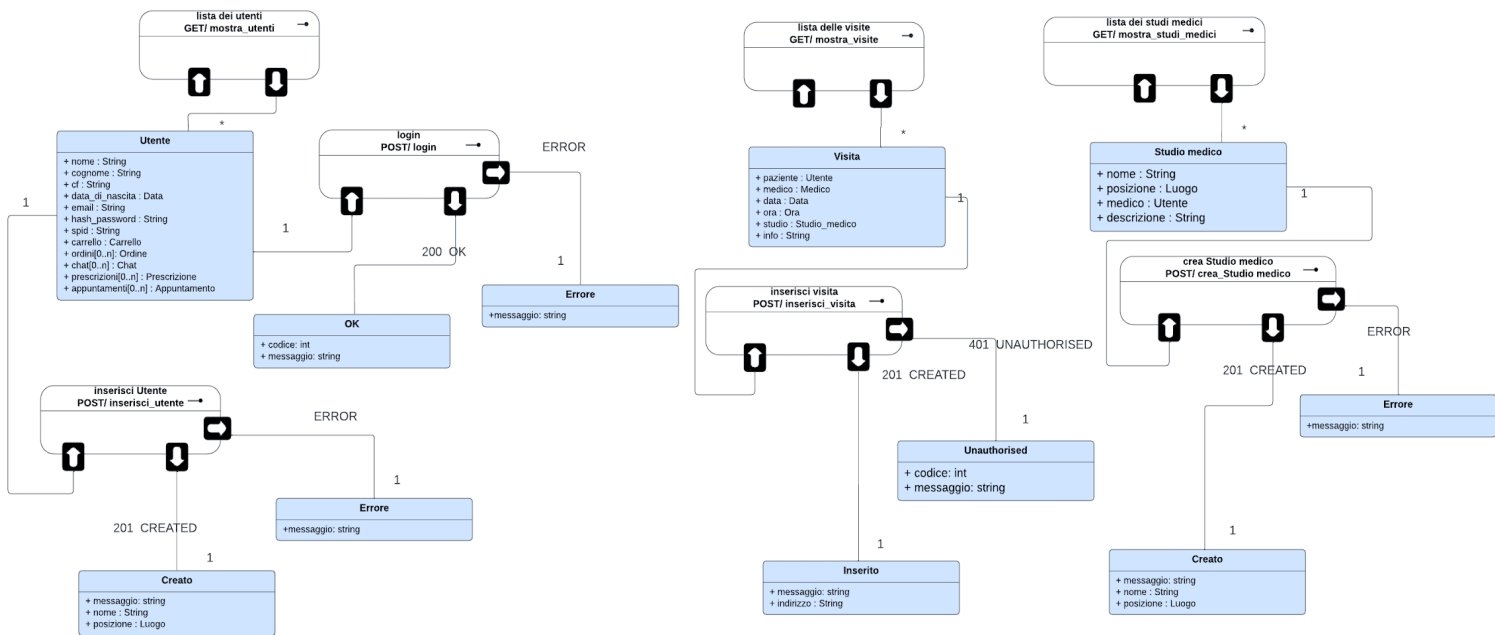
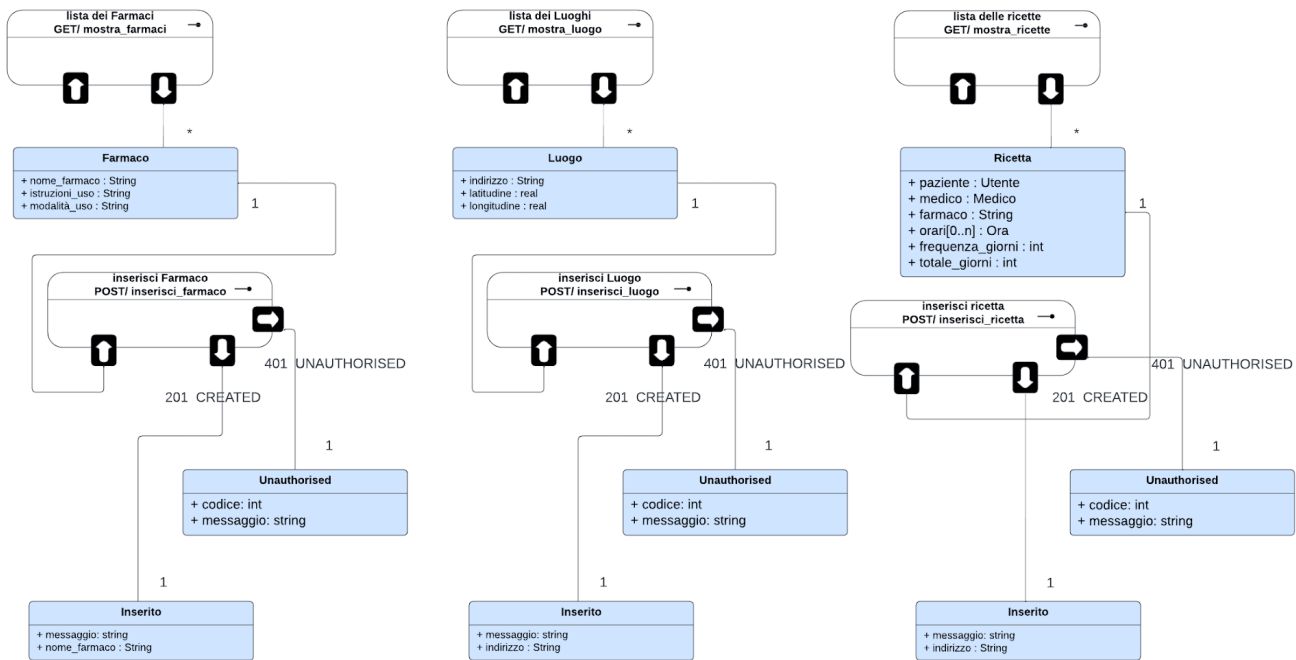


2.4.2. Resources Models

In questa sezione del documento presente Il diagramma delle API nel quale sono specificati i nomi dei risorse, i metodi per la sua gestione (get/post) e URI tramite le quali gli utenti hanno l'accesso per le API dei risorse. Nel request body delle API si può vedere la lista dei parametri che si può mandare. Nel response body si può vedere il valore di ritorno.

Come si può vedere le API sono anche collegate tra di loro. Per esempio il response body delle API “lista degli ordini” che mostra dettagli degli ordini dei clienti è nello stesso tempo il request body delle API “inserisci ordine”. Alla sua volta l’ API “inserisci ordine” ha due response body: per il caso quando l'ordine è stato inserito (201: created) e per il caso quando l’operazione è stata fallita (401: unauthorised). La situazione è simile anche per le altre liste: lista del chat, lista delle farmacie, lista dei farmaci, lista dei luoghi, lista delle ricette, lista dei utenti, lista delle visite e lista dei studi medici. Però request body dell’API “lista dei utenti” è response body per l’ API “inserisci l’utente” e simultaneamente request body per l’API “login”:





2.5. Sviluppo API

Le api vengono configurate all'interno del file "app/app.js" che gestisce una serie di endpoint API e la documentazione Swagger. Ecco cosa fa in dettaglio:

1. Import delle dipendenze: Importa i moduli necessari, tra cui Express, Swagger UI, Cors e altri.
2. Configurazione dell'app Express:
 - 2.6. Esposizione di file statici: Configura Express per servire file statici da una cartella specificata, che potrebbe essere la tua interfaccia utente (frontend).
 - 2.7. Middleware di log: Aggiunge un middleware per registrare alcune informazioni sulle richieste, come il metodo, l'URL e le intestazioni
3. Middleware per l'analisi del corpo delle richieste: Configura Express per analizzare il corpo delle richieste come JSON o dati URL-encoded.
4. Configurazione dei percorsi API:
 - 4.1. Configura i percorsi delle API per le diverse risorse come utente, farmaco, luogo, autenticazione, visita, acquisto, ricetta e chat.
 - 4.2. Aggiunge un middleware di verifica del token JWT per alcune delle rotte (visita, acquisto, chat, ricetta) per garantire l'autenticazione.
5. Configurazione dei controller delle API: Associa i percorsi delle API ai controller corrispondenti. Questi controller sono responsabili di gestire le richieste e le risposte per ciascuna risorsa.
6. Configurazione della documentazione Swagger: Espone la documentazione Swagger attraverso il percorso "/api-docs" utilizzando Swagger UI, consentendo agli sviluppatori di esplorare e testare le API.
7. Gestione delle richieste non trovate (404): Se una richiesta non corrisponde a nessun percorso definito, l'app restituirà una risposta di errore 404.
8. Log delle variabili d'ambiente: Stampa le variabili d'ambiente DB_HOST e DB_USER, che presumibilmente contengono informazioni sulla configurazione del database.

In generale, questo codice rappresenta una base per un'applicazione web con API RESTful e documentazione Swagger. Gestisce il routing delle richieste in base ai percorsi specificati e garantisce l'autenticazione per alcune delle operazioni sensibili tramite token JWT.

La cartella app ha poi due sottocartelle principali: models e controllers.

Andiamo a vedere la cartella models che definisce gli schemi delle varie api

Utente

1. Importazione di Mongoose:

```
javascript
const mongoose = require("mongoose");
```

Questa istruzione importa la libreria Mongoose, che è una libreria JavaScript che semplifica l'interazione con il database MongoDB utilizzando MongoDB Object Data Modeling (ODM).

2. Definizione dello schema utente:

```
javascript
const utenteSchema = new mongoose.Schema({
  name: String,
  surname: String,
  year: Date,
  CF: String,
  email: { type: String, required: true },
  password: { type: String, required: true },
  account_type: String,
  indirizzo: String,
  SPID: Boolean,
  titolo_di_studio: String,
  biografia: String
});
```

Questo blocco di codice definisce uno schema Mongoose chiamato `utenteSchema`. Lo schema definisce la struttura dei documenti che verranno memorizzati nel database. Ad esempio, ogni documento utente avrà campi come "name", "surname", "email", "password", ecc. I tipi di dati dei campi sono specificati (ad esempio, "String" per campi di testo e "Date" per date). Inoltre, ci sono alcune opzioni come "required: true" che indicano che i campi "email" e "password" sono obbligatori.

3. Creazione del modello Mongoose:

```
javascript
const utente = mongoose.model('utente', utenteSchema);
```

Questo codice crea un modello Mongoose chiamato `utente` basato sullo schema `utenteSchema`. Il modello Mongoose rappresenta una collezione del database e consente di eseguire operazioni come l'aggiunta, l'aggiornamento, la ricerca e la rimozione di documenti all'interno di quella collezione.

4. Esportazione del modello:

```
javascript
module.exports = utente;
```

Questa istruzione esporta il modello `utente` in modo che possa essere utilizzato in altri moduli del progetto. Ciò consente ad altre parti del codice di interagire con il modello utente per effettuare query e operazioni sul database MongoDB relative agli utenti.

In sintesi, questo codice definisce uno schema Mongoose e un modello Mongoose per gli utenti del progetto, consentendo di interagire con i dati degli utenti in un database MongoDB in modo più strutturato e semplificato.

Luogo

1. Importazione di Mongoose: esattamente come prima importiamo la libreria Mongoose

2. Definizione dello schema del Luogo:

```
javascript
const luogoSchema = new mongoose.Schema({
  nome: String,
  indirizzo: String,
  type: String,
  distanza: Number,
  descrizione: String,
  utenteld: String
});
```

In questo blocco di codice, definiamo uno schema Mongoose chiamato ``luogoSchema``. Lo schema specifica la struttura dei documenti che saranno memorizzati nel database per la risorsa "Luogo". Questi documenti avranno i seguenti campi:

- ``nome``: Una stringa che rappresenta il nome del luogo.
- ``indirizzo``: Una stringa che rappresenta l'indirizzo del luogo.
- ``type``: Una stringa che rappresenta il tipo di luogo (potrebbe essere ad esempio un ospedale, una farmacia, ecc.).
- ``distanza``: Un numero che rappresenta la distanza del luogo.
- ``descrizione``: Una stringa che rappresenta una descrizione del luogo.
- ``otenteld``: Una stringa che rappresenta l'ID dell'utente associato a questo luogo.

3. Creazione del Modello Mongoose:

```
javascript
const luogo = mongoose.model('luogo', luogoSchema);
```

Qui, creiamo un modello Mongoose chiamato ``luogo`` basato sullo schema ``luogoSchema``. Un modello Mongoose rappresenta una collezione (o tabella) nel database MongoDB e fornisce metodi per eseguire operazioni CRUD (Create, Read, Update, Delete) su quella collezione.

4. Esportazione del Modello:

```
javascript
module.exports = luogo;
```

Alla fine del codice, esportiamo il modello ``luogo`` in modo che possa essere utilizzato in altri moduli del progetto, in particolare dai controller che gestiranno le operazioni relative ai luoghi.

In sintesi, questo codice definisce uno schema Mongoose e un modello Mongoose per la risorsa "Luogo" nel progetto. Questo modello sarà utilizzato per interagire con i dati dei luoghi nel database MongoDB in modo strutturato e semplificato.

Farmaco

1. Importazione di Mongoose:

```
javascript
const mongoose = require("mongoose");
```

Iniziamo importando la libreria Mongoose, che è uno strumento di object data modeling (ODM) per MongoDB. Mongoose semplifica l'interazione con un database MongoDB attraverso un'applicazione Node.js.

2. Definizione dello schema del Farmaco:

```
javascript
const farmacoSchema = new mongoose.Schema({
  name: String,
  modalitauso: String,
  foglio_illustrativo: String,
  scadenza: Date,
  prezzo: Number,
  quantita: Number,
  luogold: String
});
```

In questo blocco di codice, definiamo uno schema Mongoose chiamato ``farmacoSchema``. Lo schema specifica la struttura dei documenti che saranno memorizzati nel database per la risorsa "Farmaco". Questi documenti avranno i seguenti campi:

- ``name``: Una stringa che rappresenta il nome del farmaco.
- ``modalitauso``: Una stringa che rappresenta la modalità d'uso del farmaco.
- ``foglio_illustrativo``: Una stringa che rappresenta il foglio illustrativo del farmaco.
- ``scadenza``: Una data che rappresenta la data di scadenza del farmaco.
- ``prezzo``: Un numero che rappresenta il prezzo del farmaco.
- ``quantita``: Un numero che rappresenta la quantità disponibile del farmaco.
- ``luogold``: Una stringa che rappresenta l'ID del luogo associato a questo farmaco, ossia l'id della farmacia che lo vende..

3. Creazione del Modello Mongoose:

```
javascript
const farmaco = mongoose.model('farmaco', farmacoSchema);
```

Qui, creiamo un modello Mongoose chiamato ``farmaco`` basato sullo schema ``farmacoSchema``. Un modello Mongoose rappresenta una collezione (o tabella) nel database MongoDB e fornisce metodi per eseguire operazioni CRUD (Create, Read, Update, Delete) su quella collezione.

4. Esportazione del Modello:
javascript
module.exports = farmaco;

Alla fine del codice, esportiamo il modello `farmaco` in modo che possa essere utilizzato in altri moduli del progetto, in particolare dai controller che gestiranno le operazioni relative ai farmaci.

In sintesi, questo codice definisce uno schema Mongoose e un modello Mongoose per la risorsa "Farmaco" nel progetto. Questo modello sarà utilizzato per interagire con i dati dei farmaci nel database MongoDB in modo strutturato e semplificato.

Ricetta

1. Importazione di Mongoose:
javascript
const mongoose = require("mongoose");

Iniziamo importando la libreria Mongoose, che è uno strumento di object data modeling (ODM) per MongoDB. Mongoose semplifica l'interazione con un database MongoDB attraverso un'applicazione Node.js.

2. Definizione dello schema della Ricetta:
javascript
const ricettaSchema = new mongoose.Schema({
 data_ricetta: String,
 quantita: Number,
 dose: Number,
 periodo_inizio: Date,
 periodo_fine: Date,
 acquistold: String,
 descrizione: String
});

In questo blocco di codice, definiamo uno schema Mongoose chiamato `ricettaSchema`. Lo schema specifica la struttura dei documenti che saranno memorizzati nel database per la risorsa "Ricetta". Questi documenti avranno i seguenti campi:

- `data_ricetta`: Una stringa che rappresenta la data in cui la ricetta è stata fatta.
- `quantita`: Un numero che rappresenta la quantità di farmaci da acquistare specificata nella ricetta.
- `dose`: Un numero che rappresenta la dose del farmaco da prendere giornalmente o settimanalmente prescritta nella ricetta.
- `periodo_inizio`: Una data che rappresenta la data di inizio dell'assunzione del farmaco.

- ``periodo_fine``: Una data che rappresenta la data di fine dell'assunzione del farmaco.
- ``descrizione``: Una stringa che rappresenta una descrizione o note aggiuntive sulla ricetta.

3. Creazione del Modello Mongoose:

```
javascript  
const ricetta = mongoose.model('ricetta', ricettaSchema);
```

Qui, creiamo un modello Mongoose chiamato ``ricetta`` basato sullo schema ``ricettaSchema``. Un modello Mongoose rappresenta una collezione (o tabella) nel database MongoDB e fornisce metodi per eseguire operazioni CRUD (Create, Read, Update, Delete) su quella collezione.

4. Esportazione del Modello:

```
javascript  
module.exports = ricetta;
```

Alla fine del codice, esportiamo il modello ``ricetta`` in modo che possa essere utilizzato in altri moduli del progetto, in particolare dai controller che gestiranno le operazioni relative alle ricette.

In sintesi, questo codice definisce uno schema Mongoose e un modello Mongoose per la risorsa "Ricetta" nel progetto. Questo modello sarà utilizzato per interagire con i dati delle ricette nel database MongoDB in modo strutturato e semplificato.

Visita

1. Importazione di Mongoose:

```
javascript  
const mongoose = require("mongoose");
```

Iniziamo importando la libreria Mongoose, che è uno strumento di object data modeling (ODM) per MongoDB. Mongoose semplifica l'interazione con un database MongoDB attraverso un'applicazione Node.js.

2. Definizione dello schema della Visita:

```
javascript  
const visitaSchema = new mongoose.Schema({  
  data: Date,  
  utentelid: String,  
  medicoid: String,  
  descrizione: String  
});
```

In questo blocco di codice, definiamo uno schema Mongoose chiamato ``visitaSchema``. Lo schema specifica la struttura dei documenti che saranno

memorizzati nel database per la risorsa "Visita". Questi documenti avranno i seguenti campi:

- `data`: Una data che rappresenta la data della visita medica.
- `utenteld`: Una stringa che rappresenta l'ID dell'utente (paziente) associato a questa visita.
- `medicold`: Una stringa che rappresenta l'ID di un altro utente, il medico, associato a questa visita.
- `descrizione`: Una stringa che rappresenta una descrizione o note aggiuntive sulla visita medica.

3. Creazione del Modello Mongoose:

```
javascript
const visita = mongoose.model('visita', visitaSchema);
```

Qui, creiamo un modello Mongoose chiamato `visita` basato sullo schema `visitaSchema`. Un modello Mongoose rappresenta una collezione (o tabella) nel database MongoDB e fornisce metodi per eseguire operazioni CRUD (Create, Read, Update, Delete) su quella collezione.

4. Esportazione del Modello:

```
javascript
module.exports = visita;
```

Alla fine del codice, esportiamo il modello `visita` in modo che possa essere utilizzato in altri moduli del progetto, in particolare dai controller che gestiranno le operazioni relative alle visite mediche.

In sintesi, questo codice definisce uno schema Mongoose e un modello Mongoose per la risorsa "Visita" nel progetto. Questo modello sarà utilizzato per interagire con i dati delle visite mediche nel database MongoDB in modo strutturato e semplificato.

Acquisto

1. Importazione di Mongoose:

```
javascript
const mongoose = require("mongoose");
```

Questa istruzione importa la libreria Mongoose, che è uno strumento di object data modeling (ODM) per MongoDB. Mongoose semplifica l'interazione con un database MongoDB attraverso un'applicazione Node.js.

2. Definizione dello schema dell'Acquisto:

```
javascript
const acquistoSchema = new mongoose.Schema({
  data: Date,
  utenteld: String,
  farmacold: String,
```



```
});
```

In questo blocco di codice, definiamo uno schema Mongoose chiamato ``acquistoSchema``. Lo schema specifica la struttura dei documenti che saranno memorizzati nel database per la risorsa "Acquisto". Questi documenti avranno i seguenti campi:

- ``data``: Una data che rappresenta la data dell'acquisto del farmaco.
- ``utenteld``: Una stringa che rappresenta l'ID dell'utente che ha effettuato l'acquisto.
- ``farmacold``: Una stringa che rappresenta l'ID del farmaco acquistato.

3. Creazione del Modello Mongoose:

```
javascript  
const acquisto = mongoose.model('acquisto', acquistoSchema);
```

Qui, creiamo un modello Mongoose chiamato ``acquisto`` basato sullo schema ``acquistoSchema``. Un modello Mongoose rappresenta una collezione (o tabella) nel database MongoDB e fornisce metodi per eseguire operazioni CRUD (Create, Read, Update, Delete) su quella collezione.

4. Esportazione del Modello:

```
javascript  
module.exports = acquisto;
```

Alla fine del codice, esportiamo il modello ``acquisto`` in modo che possa essere utilizzato in altri moduli del progetto, in particolare dai controller che gestiranno le operazioni relative agli acquisti di farmaci.

In sintesi, questo codice definisce uno schema Mongoose e un modello Mongoose per la risorsa "Acquisto" nel progetto. Questo modello sarà utilizzato per interagire con i dati degli acquisti di farmaci nel database MongoDB in modo strutturato e semplificato.

Chat

1. Importazione di Mongoose:

```
javascript  
const mongoose = require("mongoose");
```

Questa istruzione importa la libreria Mongoose, che è uno strumento di object data modeling (ODM) per MongoDB. Mongoose semplifica l'interazione con un database MongoDB attraverso un'applicazione Node.js.

2. Definizione dello schema della Chat:

```
javascript  
const chatSchema = new mongoose.Schema({  
  data: Date,  
  mittente: String,
```

```
    destinatario: String,  
    text_message: String  
  });
```

In questo blocco di codice, definiamo uno schema Mongoose chiamato ``chatSchema``. Lo schema specifica la struttura dei documenti che saranno memorizzati nel database per la risorsa "Chat". Questi documenti avranno i seguenti campi:

- ``data``: Una data che rappresenta la data del messaggio.
- ``mittente``: Una stringa che rappresenta l'identificatore del mittente del messaggio.
- ``destinatario``: Una stringa che rappresenta l'identificatore del destinatario del messaggio.
- ``text_message``: Una stringa che contiene il testo del messaggio.

3. Creazione del Modello Mongoose:

```
javascript  
const chat = mongoose.model('chat', chatSchema);
```

Qui, creiamo un modello Mongoose chiamato ``chat`` basato sullo schema ``chatSchema``. Un modello Mongoose rappresenta una collezione (o tabella) nel database MongoDB e fornisce metodi per eseguire operazioni CRUD (Create, Read, Update, Delete) su quella collezione.

4. Esportazione del Modello:

```
javascript  
module.exports = chat;
```

Alla fine del codice, esportiamo il modello ``chat`` in modo che possa essere utilizzato in altri moduli del progetto, in particolare dai controller che gestiranno le operazioni relative alla chat.

In sintesi, questo codice definisce uno schema Mongoose e un modello Mongoose per la risorsa "Chat" nel progetto. Questo modello sarà utilizzato per interagire con i dati delle chat nel database MongoDB in modo strutturato e semplificato.

Passiamo adesso a vedere nel dettaglio i metodi contenuti nella cartella controllers, che sfruttano i modelli visti in precedenza e creano, modificano ed eliminano elementi del database basandosi proprio su quei modelli.

L'utente è rappresentato come una risorsa con diverse informazioni inclusi nome, cognome, email, password, tipo di account e altre proprietà. Di seguito sono descritte le principali funzionalità del modulo:

1. Recupero di un Utente (GET /utente/:id): Questo percorso restituisce i dettagli di un utente specifico in base all'ID fornito come parametro nell'URL. I dettagli vengono mappati in una rappresentazione di risorsa.
2. Recupero degli Utenti per Tipo (GET /utente/type/:tipo): Questo percorso restituisce una lista di utenti che hanno un tipo di account specifico. Il tipo di account viene fornito come parametro nell'URL, come per esempio GET /utente/type/ClientePaziente che restituirà l'elenco di tutti i clienti nel database . I risultati vengono mappati in una rappresentazione di risorsa.
3. Recupero di Tutti gli Utenti (GET /utente): Questo percorso restituisce una lista di tutti gli utenti. Puoi filtrare gli utenti per email specificando il parametro email nell'URL. I risultati vengono mappati in una rappresentazione di risorsa.
4. Creazione di un Utente (POST /utente): Questo percorso consente di creare un nuovo utente. Richiede varie informazioni sull'utente come nome, cognome, email, password, tipo di account, ecc. Verifica se il campo "email" è una stringa non vuota e valida secondo il formato di un indirizzo email.
5. Rimozione di un Utente (DELETE /utente/:id): Questo percorso permette di eliminare un utente esistente. Richiede l'ID dell'utente da eliminare e lo elimina dal database.

Inoltre, il codice definisce una funzione ausiliaria (mapModelloUtente) per generare e mappare i dati dell'utente in una rappresentazione coerente di risorsa.

La funzione checkIfEmailInString è utilizzata per verificare se una stringa corrisponde al formato di un indirizzo email utilizzando espressioni regolari.

In generale, questo modulo gestisce le operazioni CRUD (Create, Read, Update, Delete) per gli utenti, garantendo la validità dei dati e restituendo rappresentazioni adeguate delle risorse.

Un **luogo** è rappresentato come una risorsa con diverse informazioni, tra cui nome, indirizzo, tipo, descrizione, distanza e ID utente associato. Di seguito sono descritte le principali funzionalità del modulo:

1. Elenco dei Luoghi (GET /luogo): Questo percorso restituisce una lista di luoghi. La lista include tutti i luoghi presenti nel database. I risultati vengono mappati in una rappresentazione di risorsa che include collegamenti ipertestuali agli utenti e altre informazioni sul luogo.
2. Elenco dei Luoghi filtrati per Tipo (GET /luogo/tipo/:type): Questo percorso restituisce una lista di luoghi che hanno un tipo specifico (Farmacia o studio medico). Il tipo di luogo viene fornito come parametro nell'URL. Se nessun luogo corrisponde al tipo specificato, viene restituito un messaggio di errore 404.
3. Recupero dei Luoghi per ID Utente (GET /luogo/utente/:idUtente): Questo percorso restituisce una lista di luoghi associati a un utente specifico. L'ID dell'utente viene fornito come parametro nell'URL. Se nessun luogo è associato all'utente specificato, viene restituito un messaggio di errore 404.
4. Creazione di un Luogo (POST /luogo): Questo percorso consente di creare un nuovo luogo. Richiede diverse informazioni sul luogo come nome, indirizzo, tipo, distanza,

descrizione e ID utente associato. Una volta creato il luogo, viene restituito un codice di stato 201 (Created) insieme all'URL del nuovo luogo.

5. Rimozione di un Luogo (DELETE /luogo/:id): Questo percorso permette di eliminare un luogo esistente. Richiede l'ID del luogo da eliminare e lo elimina dal database.

Il codice utilizza il modello Mongoose Luogo per accedere e manipolare i dati dei luoghi nel database. Viene eseguita una gestione degli errori per situazioni in cui il luogo non esiste o non ci sono corrispondenze con il tipo specificato. La risposta include la gestione appropriata dei codici di stato HTTP

Un **farmaco** è rappresentato come una risorsa con diverse informazioni, tra cui nome, modalità d'uso, foglio illustrativo, scadenza, prezzo, quantità e l'ID del luogo associato. Di seguito sono descritte le principali funzionalità del modulo:

1. Recupero di un Farmaco (GET /farmaco/:id): Questo percorso restituisce i dettagli di un farmaco specifico in base all'ID fornito come parametro nell'URL. I dettagli vengono mappati in una rappresentazione di risorsa che include collegamenti ipertestuali al luogo associato e altre informazioni sul farmaco.
2. Elenco dei Farmaci (GET /farmaco): Questo percorso restituisce una lista di farmaci. La lista include tutti i farmaci presenti nel database. I risultati vengono mappati in una rappresentazione di risorsa che include collegamenti ipertestuali al luogo associato e altre informazioni sul farmaco.
3. Creazione di un Farmaco (POST /farmaco): Questo percorso consente di creare un nuovo farmaco. Richiede diverse informazioni sul farmaco come nome, modalità d'uso, foglio illustrativo, ID del luogo associato, scadenza, prezzo e quantità. Verifica se il campo "luogold" fa riferimento a un luogo esistente nel database e se il luogo è già associato a un farmaco.
4. Rimozione di un Farmaco (DELETE /farmaco/:id): Questo percorso permette di eliminare un farmaco esistente. Richiede l'ID del farmaco da eliminare e lo elimina dal database.

Il codice utilizza i modelli Mongoose Farmaco e Luogo per accedere e manipolare i dati dei farmaci e dei luoghi nel database. Viene eseguita una gestione degli errori per situazioni in cui il farmaco o il luogo non esistono o in caso di duplicati per il luogo associato. La risposta include la gestione appropriata dei codici di stato HTTP.

Una **ricetta** medica è rappresentata come una risorsa con diverse informazioni, tra cui l'ID della visita associata, l'ID del farmaco prescritto, la data della ricetta, la quantità, la dose e i periodi di inizio e fine. Ecco una descrizione delle principali funzionalità di questo modulo:

1. Recupero delle Ricette (`GET /ricetta`): Questo percorso restituisce una lista di ricette mediche. La lista può essere filtrata per visita specifica utilizzando il parametro `visitald` nella query dell'URL. I risultati vengono mappati in una rappresentazione di risorsa che include collegamenti ipertestuali alla visita associata e al farmaco prescritto.
2. Creazione di una Ricetta (`POST /ricetta`): Questo percorso consente di creare una nuova ricetta medica. Richiede diverse informazioni sulla ricetta, tra cui l'ID della visita, l'ID del farmaco prescritto, la data della ricetta, la quantità, la dose e i periodi di

inizio e fine. Prima di creare la ricetta, verifica se la visita e il farmaco specificati esistono nel database e se il farmaco è già stato prescritto in una ricetta.

3. Rimozione di una Ricetta (`DELETE /ricetta/:id`): Questo percorso permette di eliminare una ricetta medica esistente in base all'ID fornito come parametro nell'URL.

Il codice utilizza i modelli Mongoose ``Ricetta``, ``Visita`` e ``Farmaco`` per accedere e manipolare i dati delle ricette mediche, delle visite e dei farmaci nel database. Viene eseguita una gestione degli errori per situazioni in cui la visita, il farmaco o la ricetta non esistono o in caso di duplicati per il farmaco prescritto. La risposta include la gestione appropriata dei codici di stato HTTP.

Una **visita** medica è rappresentata come una risorsa con diverse informazioni, tra cui l'ID dell'utente (paziente o medico), l'ID del luogo (struttura medica), la data della visita e una descrizione. Ecco una descrizione delle principali funzionalità di questo modulo:

1. Recupero delle Visite (`GET /visita/id/:idMedico`): Questo percorso restituisce una lista di visite mediche in base all'ID del medico o dell'utente fornito come parametro nella richiesta. La ricerca può essere effettuata sia per medico che per utente, verificando prima se ci sono visite associate a un medico e, se non trovate, cercando tra le visite degli utenti. I risultati vengono mappati in una rappresentazione di risorsa che include collegamenti ipertestuali all'utente, al medico (luogo) e altri dettagli.
2. Creazione di una Visita (`POST /visita`): Questo percorso consente di creare una nuova visita medica. Richiede diverse informazioni sulla visita, tra cui l'ID dell'utente, l'ID del medico (luogo), la data della visita e una descrizione. Dopo la creazione della visita, viene restituito un header di localizzazione con il percorso alla risorsa appena creata.
3. Rimozione di una Visita (`DELETE /visita/:id`): Questo percorso permette di eliminare una visita medica esistente in base all'ID fornito come parametro nell'URL.

Il codice utilizza i modelli Mongoose ``Visita``, ``Utente`` e ``Luogo`` per accedere e manipolare i dati delle visite mediche, degli utenti e dei luoghi nel database. La risposta include la gestione appropriata dei codici di stato HTTP.

Un **acquisto** è rappresentato come una risorsa con diverse informazioni, tra cui l'ID dell'utente che effettua l'acquisto, l'ID del farmaco acquistato e la data dell'acquisto. Ecco una descrizione delle principali funzionalità di questo modulo:

1. Recupero degli Acquisti (`GET /acquisto`): Questo percorso restituisce una lista di acquisti di farmaci. La ricerca può essere effettuata in base all'ID dell'utente che ha effettuato l'acquisto utilizzando il parametro ``utenteld`` nella richiesta. Se il parametro non è presente, vengono restituiti tutti gli acquisti di farmaci. I risultati vengono mappati in una rappresentazione di risorsa che include collegamenti ipertestuali all'utente, al farmaco e altri dettagli.
2. Creazione di un Acquisto (`POST /acquisto`): Questo percorso consente di creare un nuovo acquisto di farmaci. Richiede l'ID dell'utente che effettua l'acquisto, l'ID del farmaco acquistato e la data dell'acquisto. Prima di creare l'acquisto, vengono

eseguite alcune verifiche per garantire che l'utente e il farmaco esistano e che il farmaco non sia già stato acquistato. Dopo la creazione dell'acquisto, viene restituito un header di localizzazione con il percorso alla risorsa appena creata.

3. Rimozione di un Acquisto (`DELETE /acquisto/:id`): Questo percorso permette di eliminare un acquisto di farmaci esistente in base all'ID fornito come parametro nell'URL.

Il codice utilizza i modelli Mongoose `Acquisto`, `Utente` e `Farmaco` per accedere e manipolare i dati degli acquisti, degli utenti e dei farmaci nel database. La risposta include la gestione appropriata dei codici di stato HTTP.

Una **chat** è rappresentata come una risorsa con diverse informazioni, tra cui l'ID dell'utente che invia il messaggio, l'ID del medico destinatario del messaggio, la data del messaggio e il testo del messaggio. Ecco una descrizione delle principali funzionalità di questo modulo:

1. Recupero delle Chat (`GET /chat`): Questo percorso restituisce una lista di chat. La ricerca può essere effettuata in base all'ID dell'utente che invia il messaggio utilizzando il parametro `utenteld` nella richiesta. Se il parametro non è presente, vengono restituite tutte le chat. I risultati vengono mappati in una rappresentazione di risorsa che include collegamenti ipertestuali all'utente, al medico e altri dettagli.
2. Creazione di una Chat (`POST /chat`): Questo percorso consente di creare una nuova chat. Richiede l'ID dell'utente che invia il messaggio, l'ID del medico destinatario del messaggio, la data del messaggio e il testo del messaggio. Prima di creare la chat, vengono eseguite alcune verifiche per garantire che l'utente e il medico esistano e che non esista già una chat tra loro. Dopo la creazione della chat, viene restituito un header di localizzazione con il percorso alla risorsa appena creata.
3. Rimozione di una Chat (`DELETE /chat/:id`): Questo percorso permette di eliminare una chat esistente in base all'ID fornito come parametro nell'URL.

Il codice utilizza i modelli Mongoose `Chat`, `Utente` e `UtenteMedico` per accedere e manipolare i dati delle chat, degli utenti e dei medici nel database. La risposta include la gestione appropriata dei codici di stato HTTP.

3. API documentation

Le API locali dell'applicazione FarmaciaCGZ, come descritto nella sezione precedente, sono state documentate utilizzando il modulo Node.js chiamato Swagger UI Express. Questa scelta ci consente di rendere la documentazione delle API direttamente accessibile a chiunque consulti il codice sorgente dell'applicazione.

Per generare l'endpoint dedicato alla presentazione delle API, abbiamo sfruttato Swagger UI, che crea una pagina web basata sulle specifiche OpenAPI definite. Di seguito mostriamo la pagina web che contiene la documentazione relativa alle 3 principali operazioni API: GET, POST e DELETE, utilizzate per la gestione dei dati all'interno della nostra applicazione.

GET: Questa operazione consente di visualizzare i dati in una pagina HTML.

POST: Permette di inserire nuovi dati nel nostro sistema.

DELETE: Utilizzata per rimuovere dati dal nostro sistema.

L'endpoint da utilizzare per accedere a questa documentazione è il seguente:

`http://localhost:8080/api-docs`

The screenshot shows the Swagger UI interface for an application titled "My user project Application" with version "1.0.0". The base URL is "localhost:8080/". The interface includes a "Schemes" dropdown menu set to "HTTP". Below this, there are two main sections: "Users" (Api for users in the system) and "Utente". The "Utente" section is expanded, showing four API endpoints:

- GET** `/utente` Get all utenti in system
- POST** `/utente` Post utenti in system
- GET** `/utente/{id}` Get all utenti in system
- DELETE** `/utente/{id}` Delete utenti in system

GET

/utente

Get all utenti in system

^

Parameters

Try it out

No parameters

Responses

Response content type

application/json

▼

Code

Description

200

OK

Example Value

Model

```
{
  "name": "string",
  "surname": "string",
  "year": "Unknown Type: date",
  "CF": "string",
  "email": "string",
  "password": "string",
  "account_type": "string",
  "titolo_studio": "string",
  "indirizzo": "string"
}
```

GET

/utente/{id}

Get all utenti in system

^

Parameters

Try it out

Name

Description

id * required

Api for users in the system

(path)

id

Responses

Response content type

application/json

▼

Code

Description

200

OK

Example Value

Model

```
{
  "name": "string",
  "surname": "string",
  "year": "Unknown Type: date",
  "CF": "string",
  "email": "string",
  "password": "string",
  "account_type": "string",
  "titolo_studio": "string",
  "indirizzo": "string"
}
```

POST

/utente

Post utenti in system

^

Parameters

Try it out

Name

Description

body * required

Api for users in the system

(body)

Example Value

Model

```
{
  "name": "string",
  "surname": "string",
  "year": "Unknown Type: date",
  "CF": "string",
  "email": "string",
  "password": "string",
  "account_type": "string",
  "titolo_studio": "string",
  "indirizzo": "string"
}
```

Parameter content type

application/json

▼

Responses

Response content type

application/json

▼

Code

Description

200

OK

Example Value

Model

```
{
  "name": "string",
  "surname": "string",
  "year": "Unknown Type: date",
  "CF": "string",
  "email": "string",
  "password": "string",
  "account_type": "string",
  "titolo_studio": "string",
  "indirizzo": "string"
}
```


DELETE

/utente/{id} Delete utenti in system

^

Parameters

Try it out

Name	Description
id * required (path)	Api for users in the system

id

Responses

Response content type application/json

Code	Description
200	OK

Example Value | Model

```
{  "name": "string",  "surname": "string",  "year": "Unknown type: date",  "cf": "string",  "email": "string",  "password": "string",  "account_type": "string",  "titolo_studio": "string",  "indirizzo": "string"}
```

Acquisto ^

GET

/acquisto

Get all acquisti in system

^

POST

/acquisto

Post Acquisto in system

^

GET

/acquisto/{id}

Get all acquisti in system

^

DELETE

/acquisto/{id}

Delete utenti in system

^

Chat ^

GET

/chat

Get all chat in system

^

POST

/chat

Post chat in system

^

GET

/chat/{id}

Get all utenti in system

^

DELETE

/chat/{id}

Delete utenti in system

^

Farmaco ^

GET

/farmaco

Get all farmaco in system

^

POST

/farmaco

Post farmaco in system

^

GET

/farmaco/{id}

Get all utenti in system

^

DELETE

/farmaco/{id}

Delete utenti in system

^

Luogo

GET	/luogo	Get all luoghi in system	⌵
POST	/luogo	Post luoghi in system	⌵
GET	/luogo/{id}	Get all utenti in system	⌵
DELETE	/luogo/{id}	Delete utenti in system	⌵

Ricetta

GET	/ricetta	Get all ricetta in system	⌵
POST	/ricetta	Post ricetta in system	⌵
GET	/ricetta/{id}	Get all utenti in system	⌵
DELETE	/ricetta/{id}	Delete utenti in system	⌵

Visita

GET	/visita	Get all visite in system	⌵
POST	/visita	Post utenti in system	⌵
GET	/visita/{id}	Get all utenti in system	⌵
DELETE	/visita/{id}	Delete utenti in system	⌵

Models

```
Utente ▾ {  
  name*           string  
  surname*        string  
  year*           date  
  cf*             string  
  email*          string  
  password*       string  
  account_type*   string  
  titolo_studio*  string  
  indirizzo*      string  
}
```

```
Acquisto ▾ {  
  data*           > [...]   
  utenteId*       > [...]   
  farmacoId*      > [...]   
}
```

Chat >

Farmaco >

Luogo >

Ricetta >

Visita >

4. FrontEnd Implementation

Per poter descrivere il frontend è bene descrivere le 4 pagine principali che corrispondono alle pagine visualizzate dai 4 tipi di utenti:

UTENTE ANONIMO

Pagina principale

FarmaciaCGZ

[Farmaci](#) [Farmacie](#) [Medici](#) [Log in](#)

Pulsanti accesso

[Registrati](#) [Login](#)

Effettua il login

Logged User: [Login](#)

L'intestazione della pagina contiene il titolo "FarmaciaCGZ" e una serie di bottoni di navigazione che portano alle diverse sezioni dell'applicazione, tra cui "Farmaci", "Farmacie" e "Medici". In alto a destra c'è anche un pulsante "Log in" per accedere alla pagina di accesso.

Sotto l'intestazione ci sono due pulsanti, "Registrati" e "Login", che consentono all'utente di scegliere se desidera registrarsi o effettuare l'accesso.

Il modulo di registrazione compare quando l'utente clicca su "Registrati".

FarmaciaCGZ

[Farmaci](#) [Farmacie](#) [Medici](#) [Log in](#)

Pulsanti accesso

[Registrati](#) [Login](#)

Registrati

Name:	<input type="text"/>
Cognome:	<input type="text"/>
email:	<input type="text"/>
Password:	<input type="password"/>
Codice fiscale:	<input type="text"/>
Indirizzo:	<input type="text"/>
Numero tel:	<input type="text"/>
Data di nascita:	<input type="text"/> gg/mm/aaaa <input type="button" value="📅"/>
SPID:	<input type="radio"/> Sì <input type="radio"/> No
Tipo di account:	<input type="radio"/> Cliente/Paziente <input checked="" type="radio"/> Medico <input type="radio"/> Farmacista

Titolo di studio:	<input type="text"/>
Bio:	<input type="text"/>

[Registrati](#)

Questo modulo include campi per inserire dettagli come nome, cognome, email, password, codice fiscale, indirizzo, numero di telefono, data di nascita e un'opzione SPID (con scelta tra "Si" e "No"). L'utente può anche selezionare il tipo di account tra "Cliente/Paziente," "Medico," e "Farmacista". A seconda dell'opzione selezionata, potrebbero apparire ulteriori campi come "Titolo di studio" e "Bio". Il pulsante "Registrati" permette di inviare il modulo di registrazione.

Il modulo di accesso (Login) appare quando l'utente clicca su "Login" e richiede l'inserimento di email e password per l'accesso.

In sintesi, questa pagina offre all'utente un'interfaccia per registrarsi o effettuare il login nell'applicazione FarmaciaCGZ, consentendo loro di accedere ai servizi offerti dall'applicazione.

Farmaci

FarmaciaCGZ

[Farmaci](#) [Farmacie](#) [Medici](#) [Log in](#)

Farmaci

Tabella farmaci

Nome:	Modalità uso:	Foglio illustrativo:	Scadenza	Prezzo	Quantità
Rabas	Prendere 3 pastiglie al giorno	Il farmaco presenta questo principio attivo	2025-06-09T00:00:00.000Z	15	5
Tachipirina	Assumere una pastiglia	Puo' avere degli effetti collaterali	2029-03-11T00:00:00.000Z	40	3

Cliccando su “Farmaci” accediamo alla pagina relativa alla lista di farmaci, essendo degli utenti anonimi, l'unica cosa che possiamo fare è visualizzare tutti i farmaci presenti nel sito, senza poter interagire.

Farmacie

FarmaciaCGZ

[Farmaci](#) [Farmacie](#) [Medici](#) [Log in](#)

Farmacie disponibili

[Lista](#) [Mappa](#)

Lista

Nome:	Indirizzo:	Distanza:	Descrizione:
Farmacia Trento	Corso 3 Novembre	4	Qui si vendono farmaci

Premendo su “farmacie” verrà visualizzato l’elenco di farmacie, mentre premendo su “mappa” verrà visualizzata una mappa della città con le farmacie più vicine.

Medici

FarmaciaCGZ

[Farmaci](#) [Farmacie](#) [Medici](#) [Log in](#)

Medici

[Lista](#) [Mappa](#)

Lista

Nome	Cognome	Tipo	Titolo di studio	Nome Luogo	Indirizzo:	Tipo:	Descrizione:	Numero di telefono:	Distanza:	
Michele	Ghilardi	9832-02-05T00:00:00.000Z	hklfghigbn	michele.ghilardi@studenti.unitn.it	via Caio 15	true	Studio Medico	Corso 3 Novembre	4	Qui si fanno dei controlli

Cliccando su “Medici” verrà visualizzato un elenco che raccoglie le informazioni di medici e studi medici, così da avere tutte le informazioni necessarie a portata di mano. Essendo sempre utenti anonimi, possiamo solo visualizzare l’elenco.

CLIENTEPAZIENTE

Pagina principale

FarmaciaCGZ

[Lista farmaci acquistabili](#) [Lista farmaci acquistati](#) [Farmacie](#) [Medici](#) [Lista visite](#) [Scheda assunzione farmaci](#) [Account](#)

Log:

luca.ciuiloi@studenti.unitn.it
ClientePaziente
[Log out](#)

La pagina principale di un utente loggato parte dalla sezione “Account” con le informazioni di base dell’utente: email e tipo di account creato e un pulsante “Log out”. Dopo l’accesso, nella barra di navigazione si può notare come ora abbiamo molte più possibilità di scelta. “Farmacie” e “Medici” le abbiamo già viste, quindi le salteremo, passiamo a “Lista farmaci acquistabili”.

Lista farmaci acquistabili

FarmaciaCGZ

[Lista farmaci acquistabili](#) [Lista farmaci acquistati](#) [Farmacie](#) [Medici](#) [Lista visite](#) [Scheda assunzione farmaci](#) [Account](#)

Farmaci

Tabella farmaci

Nome:	Modalità uso:	Foglio illustrativo:	Scadenza	Prezzo	Quantità	
Rabas	Prendere 3 pastiglie al giorno	Il farmaco presenta questo principio attivo	2025-06-09T00:00:00.000Z	15	5	Acquista
Tachipirina	Assumere una pastiglia	Puo' avere degli effetti collaterali	2029-03-11T00:00:00.000Z	40	3	Acquista

Qui possiamo notare come di fianco ai farmaci ora compare l'opzione "acquista". Questo perché ora siamo utenti loggati come ClientePaziente.

Acquisti

FarmaciaCGZ

[Lista farmaci acquistabili](#) [Lista farmaci acquistati](#) [Farmacie](#) [Medici](#) [Lista visite](#) [Scheda assunzione farmaci](#) [Account](#)

Tabella acquisti

Nome:	Data:	Prezzo	Quantità
2022-10-31T23:00:00.000Z			
Rabas	15	5	

Qui abbiamo l'elenco dei farmaci acquistati

MEDICO

Account

FarmaciaCGZ

[Lista Visite](#) [Informazioni studio medico](#) [Lista pazienti](#) [Account](#)

Log:

michele.ghilardi@studenti.unitn.it
Medico
[Log out](#)

Come nel caso dell'utente ClientePaziente, anche il medico parte visualizzando le informazioni dell'account, ma al posto di ClientePaziente ha medico

Informazioni studio medico

FarmaciaCGZ

[Lista Visite](#) [Informazioni studio medico](#) [Lista pazienti](#) [Account](#)

Informazioni studio medico

Nome	Indirizzo:	Distanza:	Descrizione:
Studio Medico	Corso 3 Novembre	4	Qui si fanno dei controlli

Vediamo invece la pagina “informazioni studio medico” che mostra le informazioni dello studio medico associato a un medico. Se per caso un medico non ha associato ancora il suo studio, la pagina si presenta in questo modo:

FarmaciaCGZ

[Lista Visite](#) [Informazioni studio medico](#) [Lista pazienti](#) [Account](#)

Informazioni studio medico

Registra luogo

Nome luogo:	<input type="text"/>
Indirizzo luogo:	<input type="text"/>
Distanza:	<input type="text"/>
Descrizione luogo:	<input type="text"/>
<input type="button" value="Inserisci utente e luogo"/>	

Dando la possibilità all’utente di registrare il suo studio.

Pazienti

FarmaciaCGZ

[Lista Visite](#) [Informazioni studio medico](#) [Lista pazienti](#) [Account](#)

Pazienti

[Lista pazienti in cura](#) [Lista pazienti completa](#)

Completa

Nome:	Cognome	data di nascita	Codice Fiscale	Email	Indirizzo	SPID	
Luca	Ciullo	2002-02-11T00:00:00.000Z	hfgiklò	luca.ciulloi@studenti.unitn.it	via Qualunque	true	Aggiungi paziente
Luca	Ciullo	2002-02-11T00:00:00.000Z	hfgiklò	luca.ciulloi@studenti.unitn.it	via Qualunque	true	Aggiungi paziente
Anastasia	Zyrianova	2002-11-01T23:00:00.000Z	CLLLC02b84849	anastasias.zyrianova@studenti.unitn.it	via Verde 23	true	Aggiungi paziente

In questa pagina viene visualizzata la lista completa dei pazienti, con l’opzione “Aggiungi paziente”. Una volta premuto il pulsante, l’utente viene aggiunto alla lista dei pazienti in cura, Nell’elenco dei pazienti in cura ci sono più opzioni:: “Visita”, “Chat” e “Crea nuova prescrizione”.

Visita permette di creare nuove visite

Chat fa accedere il medico alla chat tra lui e il paziente

Crea una nuova prescrizione crea una nuova ricetta con tutti i farmaci da prescrivere al paziente.

FARMACISTA

Account

FarmaciaCGZ

Prodotti in vendita

Prodotti venduti

Informazioni farmacia

Account

Log:

anastasia.zyrianova@studenti.unitn.it
Farmacista

Log out

Informazioni farmacia

FarmaciaCGZ

Prodotti in vendita

Prodotti venduti

Informazioni farmacia

Account

Informazioni studio medico

Nome	Indirizzo:	Distanza:	Descrizione:
Farmacia Trento	Corso 3 Novembre	4	Qui si vendono farmaci

Prodotti in vendita

FarmaciaCGZ

Prodotti in vendita

Prodotti venduti

Informazioni farmacia

Account

Farmaci

Lista Farmaci

Inserisci nuovo farmaco

Tabella farmaci

Nome:	Modalità uso:	Foglio illustrativo:	Scadenza	Prezzo	Quantita	
Rabas	Prendere 3 pastiglie al giorno	Il farmaco presenta questo principio attivo	2025-06-09T00:00:00.000Z	15	5	<div>Elimina farmaco</div>
Tachipirina	Assumere una pastiglia	Puo' avere degli effetti collaterali	2029-03-11T00:00:00.000Z	40	3	<div>Elimina farmaco</div>

In questa pagina al posto di “Acquista farnaci” c’è “Elimina farmaco” che da la possibilità al farmacista di eliminare il farmaco. In alto di fianco a “Lista farmaci” abbiamo anche “Inserisci nuovo farmaco”

FarmaciaCGZ

Prodotti in vendita

Prodotti venduti

Informazioni farmacia


Account

Farmaci

Lista Farmaci

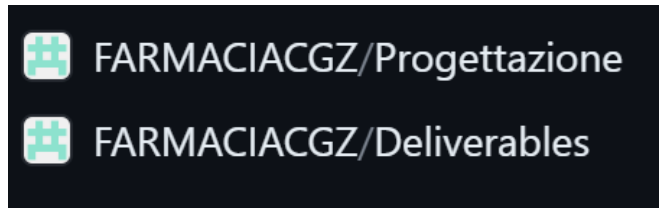
Inserisci nuovo farmaco

Insert new farmaco:

Name:	
Modalità d'uso:	
Foglio illustrativo:	
Scadenza:	gg / mm / aaaa 
Prezzo:	
Quantita:	
	<div>Inserisci Farmaco</div>

Solo i farmacisti possono inserire nuovi farmaci collegati alla loro farmacia.

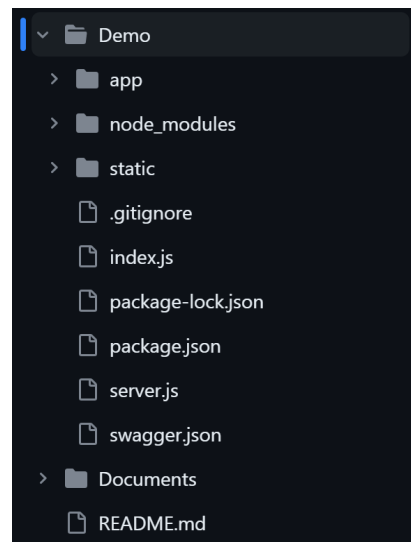
5. GitHub Repository and Deployment Info



All'interno della nostra repository di github sono presenti due macro-cartelle principali: Deliverables e progettazione.

In deliverables, sono presenti tutti i pdf dei documenti riguardanti le varie consegne, quindi i deliverable e i documenti ufficiali.

In progettazione invece:



ci sono due altre sotto-cartelle: Demo che contiene i file delle api e dei vari file javascript e html descritti in precedenza, e Documents che contiene documenti utili alla realizzazione del progetto e dei vari deliverables.

****Manca link per Heroku****

6. Testing

All files

36.22% Statements 346/483 7.05% Branches 6/85 4.25% Functions 2/47 36.22% Lines 346/483

Press n or j to go to the next uncovered block, b, p or k for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
app	<div></div>	72.22%	52/72	33.33%	6/18	40%	2/5	72.22%	52/72
app/controllers	<div></div>	21.78%	66/303	0%	0/67	0%	0/42	21.78%	66/303
app/models	<div></div>	100%	28/28	100%	0/0	100%	0/0	100%	28/28

All files app

72.22% Statements 52/72 33.33% Branches 6/18 40% Functions 2/5 72.22% Lines 52/72

Press n or j to go to the next uncovered block, b, p or k for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
app.js	<div></div>	100%	43/43	85.71%	6/7	100%	2/2	100%	43/43
authentication.js	<div></div>	33.33%	6/18	0%	0/4	0%	0/1	33.33%	6/18
tokenChecker.js	<div></div>	27.27%	3/11	0%	0/7	0%	0/2	27.27%	3/11