

Progetto di

Testing e verifica del software

Basato sul progetto di Informatica III (B) «Gestione Farmacie»

Samuele Ferri

a.a. 2019-2020



Università degli studi di Bergamo
Scuola di Ingegneria
Corso di laurea in Ingegneria Informatica
v 0.0.1

Indice

I. Fase 0 - Fase 1	1
1. Requisiti	3
1.1. Introduzione	3
1.2. Toolchain	4
2. Specifiche	7
2.1. Introduzione	7
2.2. Funzionalità richieste	7
2.3. Analisi casi d'uso	8
2.3.1. U-UC1: Registrazione cliente	8
2.3.2. U-UC2: Login cliente/farmacista	9
2.3.3. U-UC3: Login manager	9
2.3.4. A-UC1: Gestione carrello dei farmaci	10
2.3.5. A-UC2: Prenotazioni dei farmaci	10
2.3.6. A-UC3: Acquisto dei farmaci	11
2.3.7. O-UC1: Calcolo automatico degli orari settimanali delle farmacie	12
2.3.8. O-UC2: Modifica manuale degli orari settimanali delle farmacie	12
2.3.9. O-UC3: Visualizzazione orario settimanale di tutte le farmacie	13
2.3.10. O-UC4: Visualizzazione orario settimanale della singola farmacia	13
2.3.11. C-UC1: Inserimento nuova farmacia	14
2.3.12. C-UC2: Eliminazione farmacia	14
2.3.13. C-UC3: Gestione farmacia ferie/indisponibilità temporanea . .	15
2.3.14. F-UC1: Registrazione del farmaco in ingresso nel magazzino .	15
2.3.15. F-UC2: Modifica manuale della disponibilità dei farmaci . . .	16
2.3.16. F-UC3: Generazione report delle vendite	16
2.3.17. F-UC4: Visualizzazione farmacista della disponibilità dei farmaci	17
2.3.18. F-UC5: Vendita nuovo prodotto	17
2.3.19. F-UC5: Trasferimento medicinali	18
2.4. Use cases diagram	19
3. Architettura	21
3.1. Architettura hardware	21

3.2. Architettura software	21
3.2.1. Deployment diagram	24
3.2.2. Component diagram	25
3.3. Analisi dei componenti	27
3.3.1. Database	28
3.3.2. Component Shop	32
 II. Fase 2	 33
 4. Implementazione	 35
4.1. Selezione funzioni da implementare	35
4.2. Apps	36
 5. Authentication	 37
5.1. Tokens	37
5.2. Permission	37
 6. Shop	 39
6.1. Home	40
6.2. About e Contact	40
6.3. Pharmacies	41
6.4. Categories, Products e Search	41
6.5. Cart e Payment	42
6.6. Timetable	43
6.7. Admin	44
 7. REST API	 45
 III. Fase 3	 49
 8. Timetable	 51
8.1. Descrizione	51
8.2. Pseudocodice	53
8.3. Analisi di complessità	54
8.4. Activity diagram	55
8.5. Testing dell'algoritmo	56
8.6. Struttura di un caso di test	56

IV. Fase 4	59
9. Transfer	61
9.1. Descrizione	61
9.2. Criterio scelta	62
9.3. Pseudocodice	63
9.4. Analisi di complessità	64
9.5. Activity diagram	65
9.6. Testing dell'algoritmo	65
9.7. Struttura di un caso di test	66
V. Testing e verifica del software	69
10.Code Testing	71
10.1. Test (unittest)	71
10.1.1. Models tests	71
10.1.2. Forms tests	72
10.1.3. Views tests	73
10.2. Test parametrici	74
10.3. Criteri di copertura	76
10.4. Criteri avanzati di copertura	78
10.4.1. Mutation Testing	78
10.4.2. TODO	79
10.5. Continuous Integration (CI)	79
10.6. Capture and Replay	80
10.7. Mock	80
11.Code Verification	83
11.1. Assertions	83
11.2. Design by Contract	84
11.3. Program Verification	86
11.4. Analisi statica	86
11.4.1. Pylint	86
11.4.2. Flake8	87
11.4.3. Bandit	88
11.4.4. Pyreverse e GraphViz	89
11.5. Code Refactoring	91
11.6. Integrazione con CI	93
11.7. Code Inspection	93
12.Model Verification	97
13.Model Based Testing	99

VI. Manuale utente	101
Bibliografia	107

Parte I.

Fase 0 - Fase 1

1. Requisiti

1.1. Introduzione

Il cliente, in possesso di una catena di farmacie private localizzate in Lombardia, richiede e commissiona la creazione di un'applicazione web per migliorare la gestione delle vendite, la gestione degli orari e l'interazione con i propri clienti.

Le distanze da una farmacia ad un'altra sono variabili, in genere sono distanziate da un minimo di circa 1 km ad un massimo di circa 100 km coprendo una regione spaziale quadrata di lato 100 km.

Le farmacie sono strutturate tutte secondo lo stesso modello: ognuna è gestita da 1 manager della farmacia e conta 5 dipendenti; ogni farmacia ha 3 casse per la vendita con un lettore QR code/barcode in dotazione per ogni cassa.

Le farmacie, attualmente, hanno orari differenti di apertura e chiusura secondo decisioni locali prese dal manager di ogni singola farmacia; inoltre, ogni farmacia deve essere aperta almeno X ore nell'arco della settimana.

Dal cliente emerge la volontà di automatizzare/ottimizzare diversi ambiti, sia relativi all'interazione con i propri clienti che per la gestione delle proprie farmacie.

Per quanto riguarda la gestione dell'interazione con i propri clienti vengono esposti i seguenti requisiti:

1. Creare un applicativo dove i clienti potranno registrarsi e loggare con le proprie credenziali e che metta a disposizione le seguenti funzionalità:
 - a) Accesso e registrazione all'applicazione cliente.
 - b) Visualizzazione degli orari di apertura: permettere ai clienti, anche non registrati, di visionare gli orari settimanali di apertura delle singole farmacie.
 - c) Visualizzazione della disponibilità dei farmaci: dovrà essere possibile visionare dai potenziali clienti, per ogni farmaco, la disponibilità o meno, in ogni farmacia con la relativa quantità a disposizione.
 - d) Acquisto farmaci: per gli utenti loggati sarà possibile acquistare online i farmaci per i quali non è richiesta la ricetta del medico e scegliere dove ritirarli: presso la farmacia più vicina oppure in un'altra farmacia a scelta del cliente.

Per quanto riguarda la gestione delle farmacie, il cliente espone i seguenti requisiti:

1. Gestione degli orari: volontà di gestire gli orari delle proprie farmacie per poter avere sempre almeno una farmacia aperta a disposizione per i clienti, sapendo che ogni farmacia deve essere aperta almeno X ore nell'arco della settimana.
2. Gestione delle vendite e del magazzino:
 - a) Registrazione in modo automatico della vendita del farmaco tramite lettore QR code/barcode.
 - b) Registrazione in modo automatico dei farmaci in ingresso al magazzino (acquistati tramite altri applicativi già esistenti).
 - c) Modifica manuale dei dati relativi alle quantità disponibili dei farmaci di ogni farmacia (ad esempio nel caso in cui una o più confezioni scada, deve essere decrementata la disponibilità in magazzino).
 - d) Trasferimento dei farmaci: si potrà dare la possibilità al farmacista, in caso di necessità (disponibilità bassa) di fare arrivare nella sua farmacia X il farmaco da farmacie limitrofe che ne hanno disponibilità (ad esempio Y e Z). La scelta di far arrivare il farmaco da Y o Z dovrà dipendere da due parametri: dalla quantità disponibile in Y e in Z (ridistribuzione della quantità) e dalla distanza Y-X e Z-X.
 - e) Generazione automatica giornaliera dei report sulle vendite; questi report vengono poi utilizzati per migliorare la scelta della quantità di farmaci da acquistare in ogni farmacia.

1.2. Toolchain

Per la realizzazione del presente software verranno utilizzati i seguenti strumenti:

- Modellazione:
 - **Use case diagram, deployment diagram, component diagram:** *UMLet*, strumento open source, molto intuitivo ed utile per la realizzazione di diagrammi (principalmente UML) (<http://www.umlet.com>)
 - **Use case diagram, deployment diagram, component diagram:** *Draw.io*, strumento online per la realizzazione di diagrammi (<https://www.draw.io/>)
 - **Class diagram, sequence diagram:** *pyreverse*, UML Diagrams for Python, analizza codice Python ed estrarre UML class diagrams e package dependencies (<https://pypi.org/project/pyreverse/>)
 - **Database diagram:** *GraphViz*, libreria usata da Pyreverse per creare un database diagram (<https://www.graphviz.org/>)
- Implementazione software:

- **Linguaggio di programmazione:** *Python*, linguaggio di programmazione (<https://www.python.org/>)
- **Package manager:** *pip*, sistema di gestione dei pacchetti standard utilizzato per installare e gestire i pacchetti software scritti in Python (<https://pypi.org/project/pip/>)
- **IDE:** *JetBrains PyCharm*, ambiente di sviluppo integrato per il linguaggio Python (<https://www.jetbrains.com/pycharm/>)
- **Web framework:** *Django*, web framework con licenza open source per lo sviluppo di applicazioni web, scritto in linguaggio Python, seguendo il paradigma "Model-Template-View" (<https://www.djangoproject.com/>)
- **API:** *Django REST Framework*, toolkit potente e flessibile per costruire Web APIs in Django (<https://www.django-rest-framework.org/>)
- **API Development:** *Swagger UI*, strumento che permette di visualizzare e interagire con *OpenAPI* (*Redoc* incluso) (<https://swagger.io/>)
- **DBMS:** *SQLite3*, una libreria software scritta in linguaggio C che implementa un DBMS SQL di tipo ACID incorporabile all'interno di applicazioni (<https://docs.python.org/3/library/sqlite3.html>)
- **DBMS Access:** *DB-API 2.0 interface*, Python ha integrato il supporto per *SQLite3* (<https://docs.python.org/3/library/sqlite3.html>)
- **HTML (Stili):** *Bootstrap*, raccolta di strumenti liberi per la creazione di siti e applicazioni per il Web (<https://getbootstrap.com/>)
- **HTML (Stili):** *jQuery*, libreria JavaScript veloce, piccola e piena di funzionalità (<https://jquery.com/>)
- Code Testing:
 - **Test:** *unittest*, framework per il testing ispirato da JUnit (<https://docs.python.org/3/library/>)
 - **Test parametrici:** *parameterized*, package per il testing parametrico usando *unittest* (<https://pypi.org/project/parameterized/>)
 - **Criteri di copertura:** *coverage*, strumento per misurare la copertura del codice Python, output anche in HTML/XML (<https://coverage.readthedocs.io/>)
 - **Criteri di copertura:** *Codecov*, sito per l'analisi approfondita della copertura del codice basata sulla comparazione tra reports precedenti (<https://codecov.io/>)
 - **Criteri di copertura avanzati:** *django-mutpy*, framework per il mutation testing in Python (<https://pypi.org/project/django-mutpy/>)
 - **Continuous Integration:** *GitHub Actions*, permette di compilare il codice ed eseguire i test ad ogni push direttamente dal repository di *GitHub* (<https://help.github.com/en/actions>)

- **Capture and Replay:** *Selenium*, framework portatile per testare le applicazioni web, fornisce uno strumento di riproduzione (*Firefox* necessario) (<https://selenium.dev/>)
- **Mock:** *unittest.mock*, libreria per il testing in Python che permette di sostituire parti del sistema con oggetti mock e fare assertions sul loro funzionamento (<https://docs.python.org/3/library/unittest.mock.html>)
- Code Verification:
 - **Design by Contract:** *icontract*, pacchetto che fornisce Design by Contract in Python (<https://pypi.org/project/icontract/>)
 - **Program Verificaion:** TODO
 - **Analisi statica:** *pylint*, strumento d'analisi del codice sorgente (bug finder) che fa anche controllo di qualità per codice Python con configurazione altamente personalizzabile (*.pylintrc*) (<https://www.pylint.org/>)
 - **Analisi statica:** *flake8*, strumento per l'analisi del codice (<https://pypi.org/project/flake8>)
 - **Analisi statica:** *bandit*, framework per eseguire analisi di sicurezza del codice sorgente Python (<https://pypi.org/project/bandit/>)
 - **Code Refactoring:** *black*, formattatore di codice per Python (<https://pypi.org/project/black>)
- Model Verification
 - TODO (ASMETA)
- Model Based Testing
 - TODO
- Documentazione e Versioning:
 - **Documentazione:** *LaTeX*, con scrittura tramite l'editor *LyX* (<https://www.lyx.org/>)
 - **Versioning:** *GitHub*, servizio di hosting per progetti software (<https://github.com/>)

2. Specifiche

2.1. Introduzione

Le funzioni richieste dal committente non sono tutte importanti allo stesso livello, alcune richiedono di essere implementate prima di altre. Nelle successive sezioni descriveremo tutte le funzionalità richieste e le classificheremo a seconda dell'importanza:

- Funzioni con alta priorità: sono le funzioni sulle quali si basa il funzionamento "base" del programma che, quindi, richiedono di essere realizzate prima delle altre.
- Funzioni con media priorità: sono le funzioni che richiedono di essere realizzate prima di quelle con bassa priorità. Date le caratteristiche di queste funzioni è possibile implementarle parallelamente alle altre funzioni.
- Funzioni con bassa priorità: sono le funzioni "di contorno", dalle quali non dipende alcuna altra funzione.

2.2. Funzionalità richieste

Le funzioni richieste sono divise in gruppi per contesto; ogni funzione ha associato un codice e una priorità:

Users [USER]

- U-UC1 Registrazione cliente (Alta)
- U-UC2 Login cliente/farmacista (Alta)
- U-UC3 Login manager (Alta)

Gestione acquisiti, carrello cliente [ACQUISTI]

- A-UC1 Gestione carrello dei farmaci (Alta)
- A-UC2 Prenotazione dei farmaci (Media)
- A-UC3 Acquisto dei farmaci (Media)

Gestione orari [ORARI]

- O-UC1 Calcolo automatico degli orari settimanali delle farmacie (Alta)
- O-UC2 Modifica manuale degli orari settimanali delle farmacie (Bassa)
- O-UC3 Visualizzazione orario settimanale di tutte le farmacie (Alta)
- O-UC4 Visualizzazione orario settimanale della singola farmacia (Media)

Gestione catena farmacie [CATENA]

- C-UC1 Inserimento nuova farmacia (Alta)
- C-UC2 Eliminazione farmacia (Alta)
- C-UC3 Gestione farmacia ferie/indisponibilità temporanea (Bassa)

Gestione farmaci, vendite e magazzino [FARMACI]

- F-UC1 Registrazione del farmaco in ingresso nel magazzino (Bassa)
- F-UC2 Modifica manuale della disponibilità dei farmaci (Media)
- F-UC3 Generazione report delle vendite (Bassa)
- F-UC4 Visualizzazione farmacista della disponibilità dei farmaci (Alta)
- F-UC5 Vendita nuovo prodotto (Alta)
- F-UC6 Trasferimento medicinali (Alta)

2.3. Analisi casi d'uso

Vengono effettuate delle analisi su alcuni casi d'uso. Gli attori presenti sono: il cliente, il gestore della farmacia e il manager delle farmacie (admin) che gestisce tutto l'applicativo e calcola i turni tramite un pannello admin.

2.3.1. U-UC1: Registrazione cliente

Descrizione: Il cliente, tramite l'applicativo, potrà creare un account personale con il quale potrà usufruire di tutti i servizi che gli spettano. Con l'atto di registrazione i dati del cliente verranno inseriti nel database.

Attori coinvolti: Cliente

Precondizioni: Il cliente deve connettersi all'applicativo web.

Postcondizioni: Il cliente avrà un account e attraverso l'accesso ad esso potrà usufruire dei servizi.

Processo:

1. Il cliente accede all'applicativo web
2. Dalla tendina seleziona la sezione "Sign up"
3. Inserisce tutti i dati, tra cui username e password del proprio account
4. Preme il bottone "Sign up"

Alternative:

- Il campo "password" e "conferma password" non coincidono. In questo caso esce un messaggio d'errore e si devono ricompilare questi due campi.
- Username non valido perché già utilizzato. In questo caso esce un messaggio di errore e si deve ricompilare il campo.

2.3.2. U-UC2: Login cliente/farmacista

Descrizione: Il cliente, tramite l'applicativo, potrà loggarsi con le proprie credenziali (username e password) scelte in fase di registrazione.

Attori coinvolti: Cliente

Precondizioni: Il cliente dev'essere in possesso di un account (registrazione).

Postcondizioni: Il cliente è loggato e potrà effettuare tutte le operazioni dedicate a coloro che sono loggati.

Processo:

1. Il cliente apre l'applicativo (accede all'applicativo web)
2. Dalla tendina seleziona la sezione "Login"
3. Inserisce username e password del proprio account
4. Preme il bottone "Login"

Alternative:

- Siccome pure i farmacisti effettuano il login dalla stessa piattaforma, essi verranno indirizzati nella sezione a loro dedicata in cui potranno effettuare le loro operazioni.
- Errore nell'inserimento dei campi richiesti. Un messaggio di errore apparirà sullo schermo richiedendo la ri-compilazione dei campi.

2.3.3. U-UC3: Login manager

Descrizione: Il manager, tramite l'applicativo, potrà loggarsi al pannello admin con le proprie credenziali (username e password).

Attori coinvolti: Manager

Precondizioni: Il manager dev'essere in possesso di un account admin.

Postcondizioni: Il manager è loggato e potrà effettuare tutte le operazioni dedicate a coloro che sono manager.

Processo:

1. Il manager apre l'applicativo (accede all'applicativo web)
2. Accede alla sezione admin
3. Inserisce username e password del proprio account
4. Preme il bottone "Login"

2.3.4. A-UC1: Gestione carrello dei farmaci

Descrizione: Il cliente tramite una sezione dell'applicativo potrà accedere alla sezione "Carrello" e modificare quali prodotti aggiungere/togliere e in quale quantità.

Attori coinvolti: Cliente

Precondizioni: Il cliente deve possedere un account ed essersi loggato.

Postcondizioni: Verranno aggiornati i farmaci e le relative quantità di acquisto desiderate.

Processo:

1. Il cliente apre l'applicativo ed effettua il login
2. Dal menu principale seleziona la sezione "Cart"
3. Apparirà una sezione dalla quale al cliente sarà possibile visualizzare e modificare/aggiungere/togliere i prodotti selezionati
4. Possibilità di continuare:
 - a) Procedere alla prenotazione dei farmaci (A-UC2)
 - b) Procedere all'acquisto dei farmaci (A-UC3)

Alternative:

- Inserimento errato dei dati: Il cliente al passo 1 del processo può fallire l'autenticazione con il server e quindi non poter accedere al proprio account.

2.3.5. A-UC2: Prenotazioni dei farmaci

Descrizione: Gli utenti possono prenotare online i farmaci e scegliere dove ritirarli.

Attori coinvolti: Cliente

Precondizioni: Il cliente ha riempito il carrello con i farmaci che vuole prenotare.

Postcondizioni: Il cliente avrà una ricevuta di prenotazione per potersi recare nella farmacia scelta per il ritiro e pagare in cassa.

Processo:

1. Il cliente apre l'applicativo ed effettua il login
2. Clicca la sezione "Cart"
3. Seleziona il farmaco che vuole prenotare
4. Clicca sulla sezione "Book now"
5. Sceglie la farmacia in cui poter ritirare i farmaci
6. Cliccando su "Book now" avviene l'effettiva prenotazione
7. Viene emessa una ricevuta di prenotazione da presentare al farmacista per il ritiro dei farmaci entro sette giorni

Alternative:

- Inserimento errato dei dati: Il cliente al passo 1 del processo può fallire l'autenticazione con il server e quindi non poter accedere al proprio account.
- Scadenza dei 30 minuti di prelazione.

2.3.6. A-UC3: Acquisto dei farmaci

Descrizione: Gli utenti possono acquistare online i farmaci e scegliere dove ritirarli.

Attori coinvolti: Cliente

Precondizioni: Il cliente ha riempito il carrello con i farmaci che vuole acquistare. Deve avere aggiunto una carta di credito con saldo disponibile maggiore alla spesa da effettuare se vorrà procedere all'acquisto.

Postcondizioni: Il cliente avrà una ricevuta di acquisto per potersi recare nella farmacia scelta per il ritiro.

Processo:

1. Il cliente apre l'applicativo ed effettua il login
2. Clicca la sezione "Cart"
3. Clicca sulla sezione "Book now"
4. Seleziona il farmaco che vuole acquistare
5. Inserisce i dati della propria carta di credito
6. Cliccando su "Buy" avviene l'effettivo pagamento

Alternative:

- Inserimento errato dei dati: Il cliente al passo 1 del processo può fallire l'autenticazione con il server e quindi non poter accedere al proprio account.
- Scadenza dei 30 minuti di prelazione sul farmaco: viene visualizzato un messaggio di errore con scritto "Tempo massimo di prelazione scaduto".
- Carta di credito non accettata: viene visualizzato un messaggio di errore con la richiesta di ricompilare i campi richiesti.
- Saldo disponibile sulla carta non sufficiente per l'acquisto (pagamento rifiutato).

2.3.7. O-UC1: Calcolo automatico degli orari settimanali delle farmacie

Descrizione: Il manager attraverso il pannello admin effettua il calcolo automatico degli orari settimanali delle farmacie.

Attori coinvolti: Manager

Precondizioni: Il manager dev'essere loggato nell'area admin.

Postcondizioni: Verranno calcolati e inseriti nel database gli orari settimanali di tutte le farmacie.

Processo:

1. Il manager apre l'applicativo ed effettua il login nell'area admin
2. Clicca il bottone "Calculate timetables"
3. Dopo aver visualizzato il risultato del processo, conferma attraverso "Confirm" e i dati verranno inseriti nel database

Alternative:

- Inserimento errato dei dati: il manager al passo 1 del processo può fallire l'autenticazione con il server e quindi non poter accedere alla sezione dedicata.

2.3.8. O-UC2: Modifica manuale degli orari settimanali delle farmacie

Descrizione: Il manager attraverso l'applicativo può effettuare delle modifiche manuali agli orari calcolati in maniera automatica dall'applicativo (O-UC1).

Attori coinvolti: Manager

Precondizioni: Il manager dev'essere loggato nell'area admin.

Postcondizioni: Verranno modificati e aggiornati nel database gli orari settimanali di tutte le farmacie soggetti alla modifica manuale.

Processo:

1. Il manager apre l'applicativo ed effettua il login nell'area admin
2. Clicca il bottone "Modify timetables"
3. Seleziona la settimana in cui effettuare le modifiche
4. Effettua le modifiche desiderate
5. Attraverso il bottone "Save" effettua il salvataggio e l'aggiornamento relativo nel database

Alternative:

- Inserimento errato dei dati: il manager al passo 1 del processo può fallire l'autenticazione con il server e quindi non poter accedere alla sezione dedicata.

2.3.9. O-UC3: Visualizzazione orario settimanale di tutte le farmacie

Descrizione: Il cliente attraverso l'applicativo può visualizzare gli orari settimanali di tutte le farmacie.

Attori coinvolti: Cliente

Precondizioni: Il cliente deve accedere all'applicativo web (non è necessario il login).

Postcondizioni: Verranno visualizzati sottoforma di tabella gli orari settimanali di tutte le farmacie.

Processo:

1. Il cliente accede all'applicativo web
2. Clicca il bottone "Timetables"
3. Verranno visualizzati sottoforma di tabella gli orari settimanali di tutte le farmacie nel periodo scelto

2.3.10. O-UC4: Visualizzazione orario settimanale della singola farmacia

Descrizione: Il cliente attraverso l'applicativo può visualizzare l'orario settimanale di una specifica farmacia.

Attori coinvolti: Cliente

Precondizioni: Il cliente deve accedere all'applicativo web (non è necessario il login).

Postcondizioni: Verranno visualizzati sottoforma di tabella gli orari settimanali della farmacia selezionata.

Processo:

1. Il cliente accede all'applicativo web
2. Attraverso la tendina "Find pharmacy", seleziona la farmacia di cui vuole visualizzare l'orario
3. Verranno visualizzati sottoforma di tabella gli orari settimanali della farmacia selezionata

Alternative:

- Invece di selezionare la farmacia dalla lista può selezionarla dalla mappa; può anche essere scelta come default la farmacia aperta più vicina calcolata tramite GPS.

2.3.11. C-UC1: Inserimento nuova farmacia

Descrizione: Il manager attraverso l'applicativo può inserire una nuova farmacia nel database.

Attori coinvolti: Manager

Precondizioni: Il manager dev'essere loggato nell'area admin.

Postcondizioni: Verranno aggiunti al database i dati della nuova farmacia.

Processo:

1. Il manager apre l'applicativo ed effettua il login nell'area admin
2. Clicca il bottone "Add pharmacy"
3. Compila il form con tutti i dati
4. Attraverso il bottone "Add", effettua il salvataggio e l'inserimento nel database

Alternative:

- Inserimento errato dei dati: il manager al passo 1 del processo può fallire l'autenticazione con il server e quindi non poter accedere alla sezione dedicata.
- I dati nel form possono non essere accettati se non consoni ai requisiti con conseguente messaggio di errore e richiesta di reinserire i campi.

2.3.12. C-UC2: Eliminazione farmacia

Descrizione: Il manager attraverso l'applicativo può eliminare una farmacia, la quale verrà tolta dal database.

Attori coinvolti: Manager

Precondizioni: Il manager dev'essere loggato nell'area admin.

Postcondizioni: Verranno eliminati dal database i dati della farmacia.

Processo:

1. Il manager apre l'applicativo ed effettua il login nell'area admin
2. Seleziona dalla tendina la farmacia che vuole eliminare
3. Clicca il bottone "Delete pharmacy"
4. Conferma l'effettiva eliminazione dopo l'uscita del messaggio di avvertimento

Alternative:

- Inserimento errato dei dati: il manager al passo 1 del processo può fallire l'autenticazione con il server e quindi non poter accedere alla sezione dedicata.

2.3.13. C-UC3: Gestione farmacia ferie/indisponibilità temporanea

Descrizione: Il manager attraverso l'applicativo inserisce se una farmacia per un determinato periodo è indisponibile e quindi non viene conteggiata nel calcolo dell'orario settimanale.

Attori coinvolti: Manager

Precondizioni: Il manager dev'essere loggato nell'area admin.

Postcondizioni: Verranno aggiunti nel database i dati relativi ai giorni e orari di indisponibilità di una determinata farmacia.

Processo:

1. Il manager apre l'applicativo ed effettua il login nell'area admin
2. Seleziona dalla tendina la farmacia in questione
3. Compila la tabella manualmente inserendo i giorni e il motivo della indisponibilità
4. Conferma cliccando "Confirm"

Alternative:

- Inserimento errato dei dati: il manager al passo 1 del processo può fallire l'autenticazione con il server e quindi non poter accedere alla sezione dedicata.

2.3.14. F-UC1: Registrazione del farmaco in ingresso nel magazzino

Descrizione: Il farmacista legge con la pistola QR code/barcode dedicato al magazzino il pacco dei farmaci. Il sistema in modo automatico estrae le informazioni dal file JSON ed effettua le query sul database (update) così da aggiornare le disponibilità

Attori coinvolti: Farmacista

Precondizioni: Il manager delle farmacie ha effettuato gli acquisti.

Postcondizioni: La disponibilità del farmaco appena acquistato verranno incrementate per la farmacia in cui è stato acquistato tale farmaco.

Processo:

1. Il farmacista legge con la pistola le confezioni di un determinato farmaco acquistato
2. Viene creato un file JSON in cui sono visualizzate le informazioni di acquisto farmaco

2.3.15. F-UC2: Modifica manuale della disponibilità dei farmaci

Descrizione: Il farmacista modifica manualmente la disponibilità di un farmaco (ad esempio causa scadenza o danneggiamento della confezione).

Attori coinvolti: Farmacista

Precondizioni: Il farmacista deve essersi loggato.

Postcondizioni: La disponibilità del farmaco verranno modificate (per la farmacia in cui il farmacista lavora).

Processo:

1. Il farmacista effettua il login nella sua area di competenza
2. Il farmacista attraverso la sezione “Products” può accedere alla lista dei farmaci presenti in quella farmacia
3. Attraverso il pulsante “Modify quantity available” può scegliere la disponibilità e aggiornarla premendo il tasto “Modify”

2.3.16. F-UC3: Generazione report delle vendite

Descrizione: Il farmacista, potrà decidere di generare il report (di vendita) della relativa farmacia.

Attori coinvolti: Farmacista

Precondizioni: Ci sono delle vendite nel database.

Postcondizioni: Il farmacista genera il report di vendita.

Processo:

1. Il farmacista accede all'applicativo web e si logga
2. Seleziona dalla tendina una farmacia
3. Seleziona la voce “Generate report”

4. Seleziona l'orizzonte temporale su cui effettuare le statistiche
5. Preme il bottone "Generate"
6. Viene visualizzato il report di vendita relativo alla farmacia e all'orizzonte temporale e le possibilità di esportazione del documento

2.3.17. F-UC4: Visualizzazione farmacista della disponibilità dei farmaci

Descrizione: Il farmacista, tramite l'applicativo, potrà cercare la disponibilità di un determinato farmaco.

Attori coinvolti: Farmacista

Precondizioni: Il farmacista deve accedere all'applicativo web.

Postcondizioni: Il farmacista visualizza le quantità di un determinato farmaco.

Processo:

1. Il farmacista accede all'applicativo web
2. Cerca il farmaco con il tasto "Find product"
3. Inserisce il nome del farmaco
4. Vengono visualizzate le informazioni generali del farmaco con la disponibilità nella farmacia

2.3.18. F-UC5: Vendita nuovo prodotto

Descrizione: Il farmacista, tramite l'applicativo, potrà inserire la vendita di un nuovo farmaco nella propria farmacia.

Attori coinvolti: Farmacista

Precondizioni: Il farmacista deve accedere all'applicativo web.

Postcondizioni: Il farmacista ha aggiunto un nuovo farmaco da vendere nella propria farmacia.

Processo:

1. Il farmacista accede all'applicativo web
2. Clicca sulla sezione "Sell a product"
3. Inserisce i dati nel form i dati per specificare le caratteristiche e dettagli del nuovo farmaco da aggiungere all'elenco dei farmaci in vendita

2.3.19. F-UC5: Trasferimento medicinali

Descrizione: Il farmacista, tramite l'applicativo, potrà richiedere la quantità che è interessato a ricevere di una determinata categoria di farmaco.

Attori coinvolti: Farmacista

Precondizioni: Il farmacista deve accedere all'applicativo web.

Postcondizioni: Il farmacista ottiene il percorso da fare per andare a ritirare la quantità di farmaci richiesta. Nel percorso gli saranno segnalate in ordine le farmacie in cui deve spostarsi e la quantità di farmaci che potrà ricevere in ognuna di esse.

Processo:

1. Il farmacista accede all'applicativo web
2. Clicca sulla sezione "Medicine transfer"
3. Inserisce i dati nel form i dati per specificare la categoria del prodotto richiesto e la relativa quantità
4. Visualizzerà il percorso da effettuare

2.4. Use cases diagram

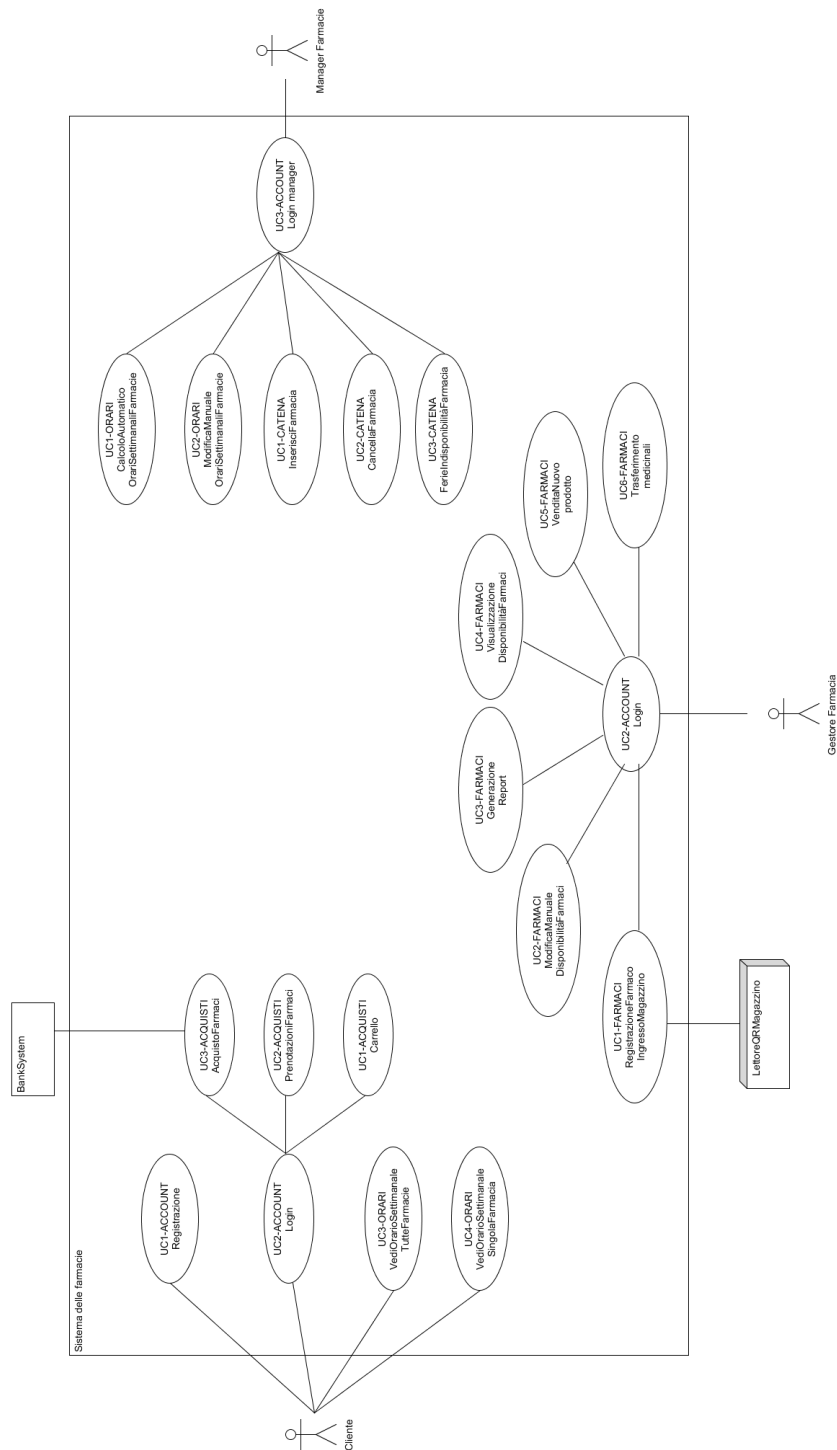


Figura 2.1.: Use cases diagram

3. Architettura

3.1. Architettura hardware

L'architettura hardware sarà formata da un singolo server che si interfacerà a un database per la gestione dei dati. I vari client usati dai vari attori si interfaceranno con il server.

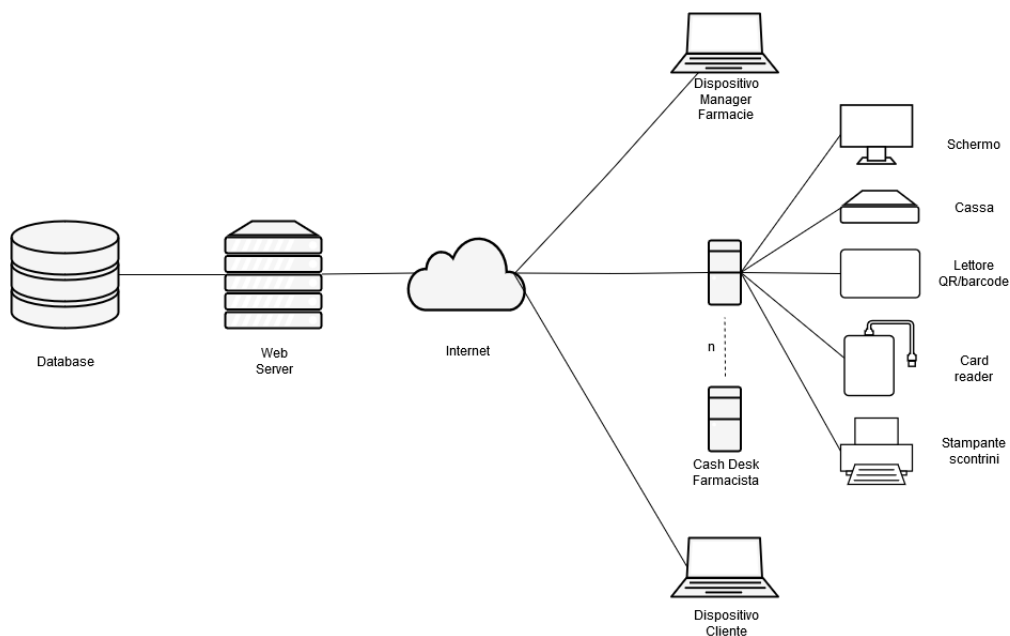


Figura 3.1.: Topology system

3.2. Architettura software

L'architettura software scelta è basata sul *Django Framework*. È stato scelto di optare per questo web framework per vari fattori tra cui:

- Time to market: dover sviluppare ogni singola riga di codice aumenta drasticamente il tuo time-to-market, ossia il tempo necessario a fare sì che il tuo progetto passi dall'essere una semplice idea a un progetto funzionante vero e

proprio; utilizzando un framework, si ha già tutta una serie di strumenti pronti all'uso che permettono di risparmiare tempo e concentrarti sullo sviluppo dell'idea.

- Sicurezza: contiene all'interno componenti riguardanti la sicurezza e l'autenticazione già pronti, bisogna solo tenerli aggiornati.
- Community: lavorare con un framework, ancora meglio se open source, riduce drasticamente il carico di lavoro potendo riutilizzare codice o template già presenti.

Inoltre con Django è possibile creare un sito web accessibile da qualsiasi tipo di terminale (computer, tablet, smartphone...) usato dai vari attori (clienti, farmacisti, manager) come si vede dall'architettura hardware del progetto.

Django, pur influenzato fortemente dalla filosofia di sviluppo Model-View-Controller, adotta un pattern Model-View-Template formato da:

- Model: è il data access layer che gestisce i dati.
- View: è il layer che esegue la business logic e interagisce con il model per gestire i dati per presentarli al template.
- Template: è il presentation layer che gestisce l'interfaccia utente completamente.

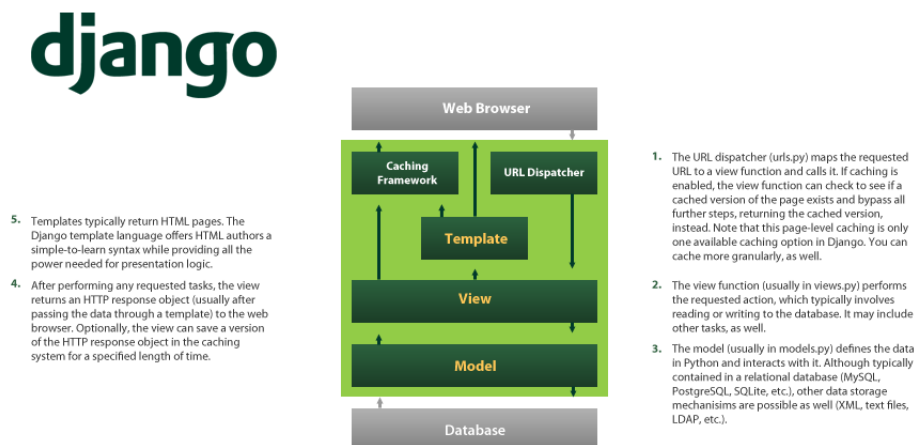


Figura 3.2.: Django

Quindi viene a mancare un controller separato tipico di un MVC pattern: infatti questa funzione viene gestita dal framework stesso.

Rispetto al framework MVC classico in Django ciò che sarebbe chiamato "controller" è chiamato "view" mentre ciò che dovrebbe essere chiamato "view" è chiamato "template".

Nella figura 3.3 è rappresentato il ciclo di vita di Django.

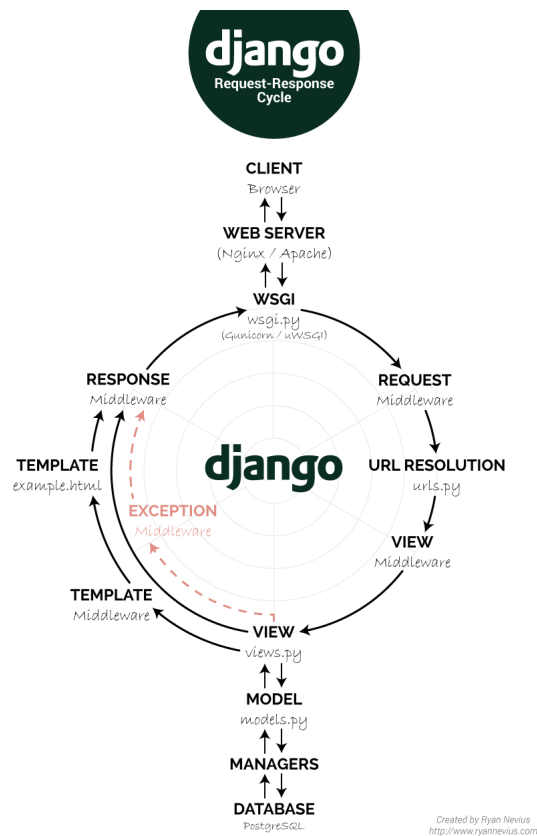


Figura 3.3.: Django nel dettaglio

3.2.1. Deployment diagram

Nella figura 3.4 sono riportati il deployment diagram della soluzione proposta per soddisfare la richiesta del committente. In particolare, si è scelto di accorpare molte componenti nel web server in modo da rendere scalabile l'applicazione su qualsiasi client avente a disposizione un semplice web browser, in aggiunta ai dispositivi di cassa nel caso del farmacista. In questo modo i clienti da qualsiasi dispositivo potranno accedere al sito e pure il manager per gestire il server.

Il database è sulla stessa workstation del web server in modo da facilitarne la comunicazione. Si è scelto di usare *SQLite3* come database visto che ha il supporto nativo di Python e per semplicità. In caso di futura mole di dati eccessiva e prestazioni ridotte si potrà passare facilmente ad alternative tipo *MySQL*.

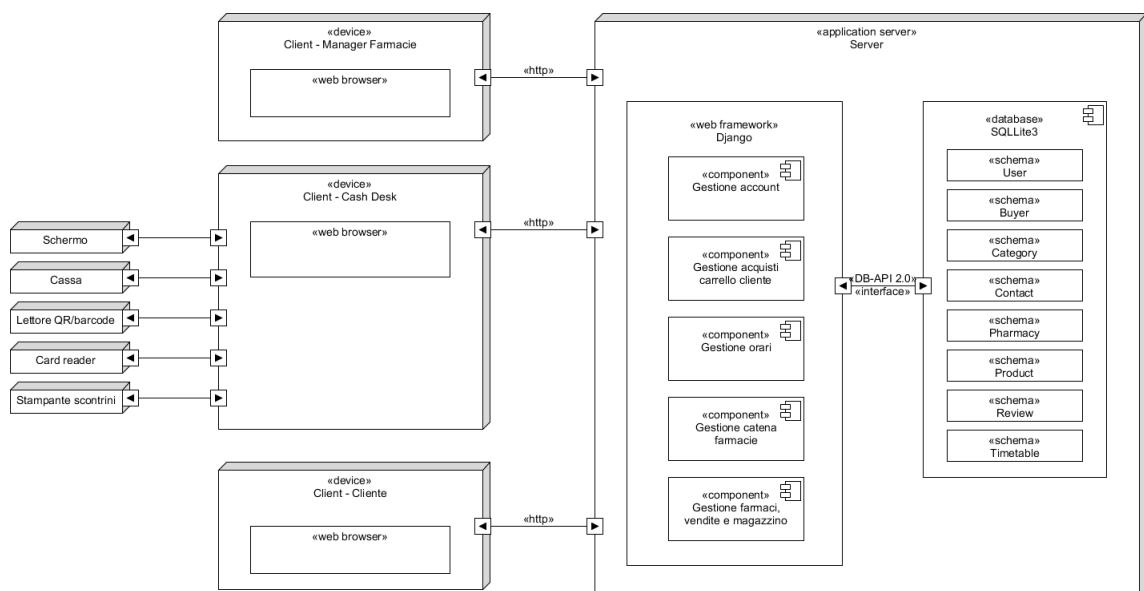


Figura 3.4.: Deployment diagram

Successivamente è stato deciso ridisegnare le componenti principali dell'applicazione, passando dal deployment con raggruppamento delle funzioni richieste dai casi d'uso al deployment diviso per macro funzionalità che l'applicazione dovrà implementare; le nuove componenti sono: *authentication*, *shop*, *timetable*, *transfer*.

Nella figura 3.5 è rappresentato il deployment diagram aggiornato con la nuova divisione in componenti.

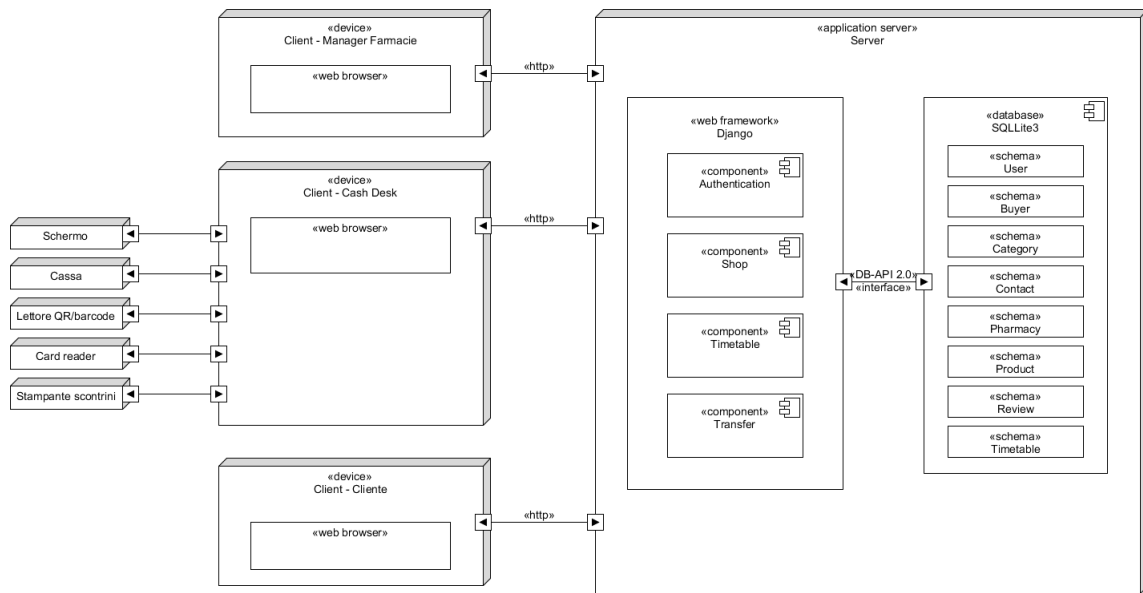


Figura 3.5.: Deployment diagram aggiornato

3.2.2. Component diagram

La figura 3.6 rappresenta il component diagram in cui ogni singolo componente ha lo stesso pattern, ossia è formato da tre blocchi come nel paradigma di Django:

- Model: permette l'interfaccia con il database.
- View: ha il compito di implementare la logica di funzionamento di ogni singolo componente ed elabora i dati del database.
- Template: si occupa di svolgere il ruolo del presenter, ovvero dell'interfaccia grafica.

L'analisi nel dettaglio e le modalità di comunicazione tra i vari componenti verranno spiegata nelle sezioni successive analizzando anche nel dettaglio la componente *shop*.

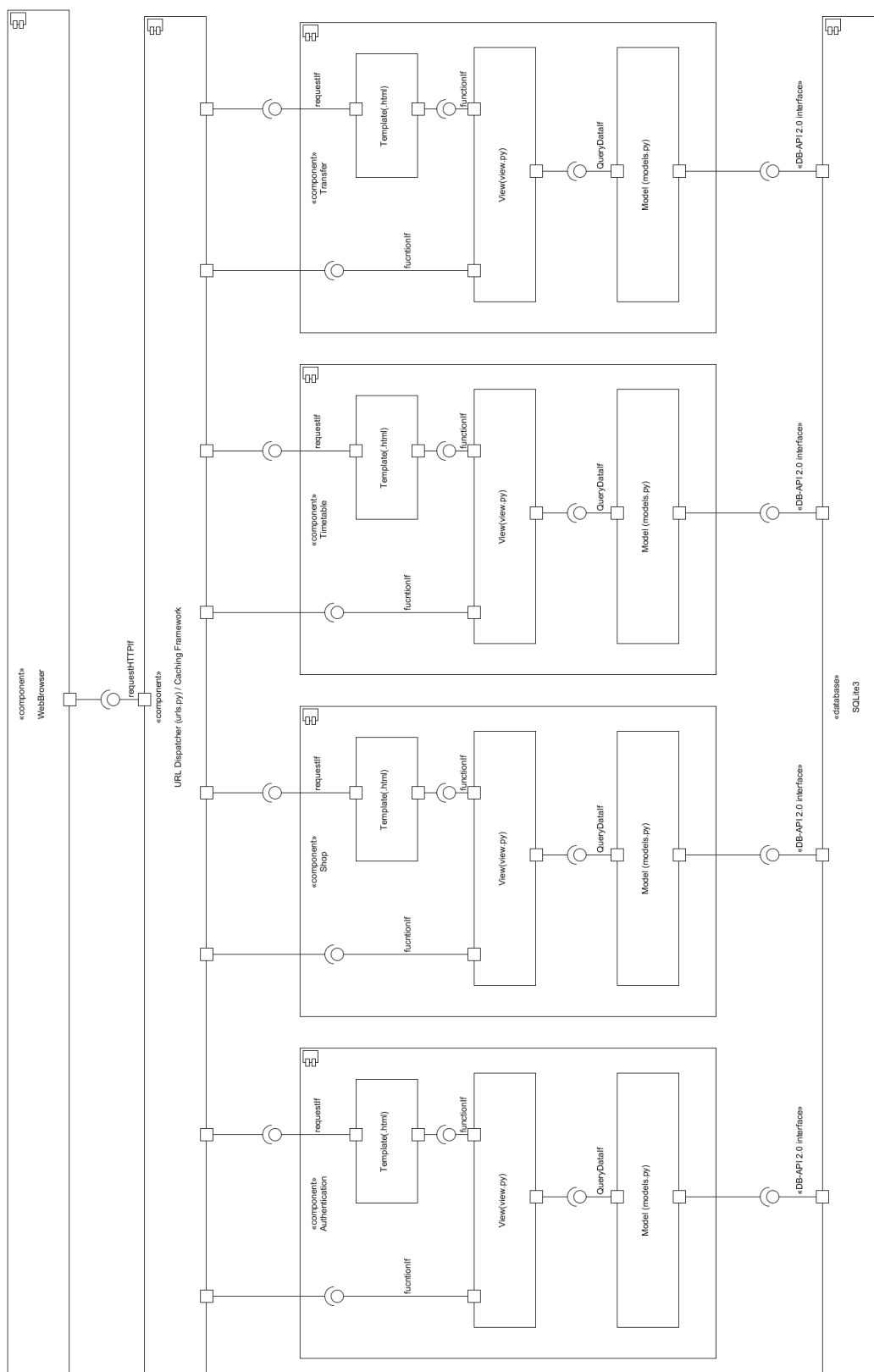


Figura 3.6.: Component diagram

3.3. Analisi dei componenti

Introduciamo il class diagram rappresentato nella figura 3.7. Sono state individuate le seguenti interfacce che interagiscono tra di loro fornendo e ricevendo servizi. I servizi forniti da ogni interfaccia sono esemplificati nei metodi che possiede.

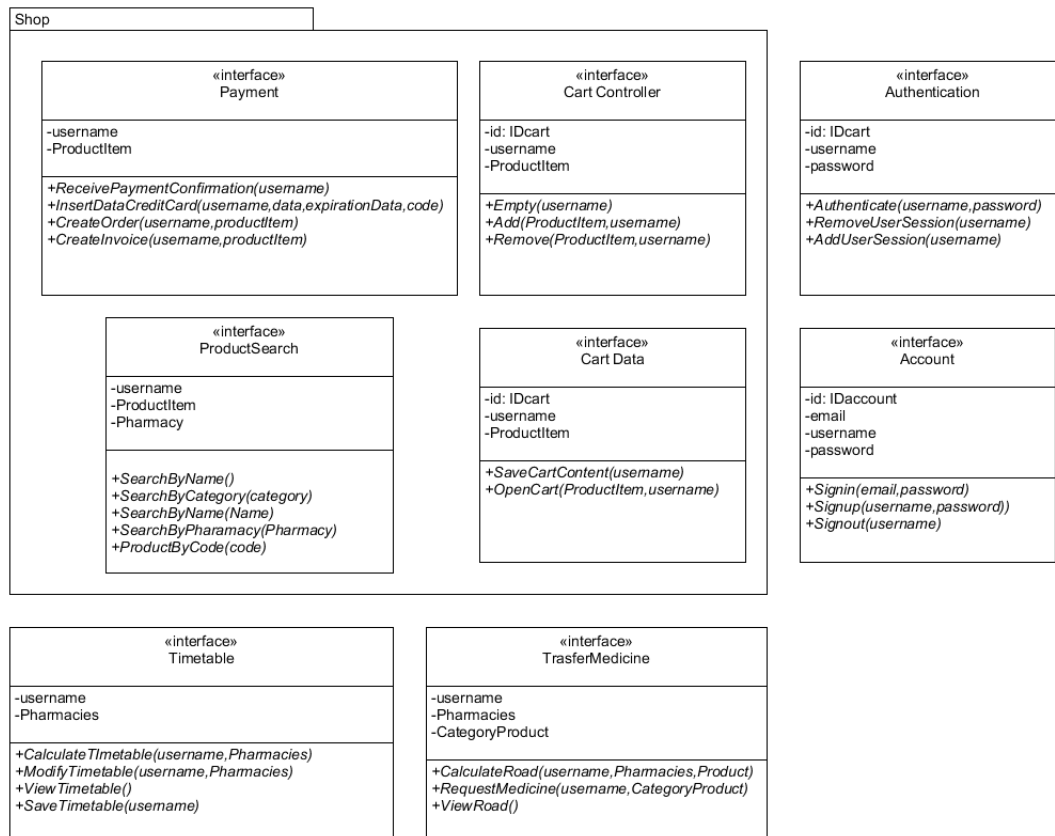


Figura 3.7.: Class diagram

Ci sono due interfacce *CartData* e *CartController*. Sono due interfacce separate che interagiscono tra loro. *CartController* si occupa di fornire tutte le azioni disponibili al cliente con il carrello: aggiungere prodotti, togliere prodotti e svuotare il carrello. *CartData* si occupa di salvare per l'account il relativo carrello in modo che rimanga anche ai prossimi login (in sessioni successive) quindi cart data fornisce funzionalità per salvare i dati del carrello di un account e recuperare i dati in sessioni successive.

3.3.1. Database

Nella figura 3.8 è rappresentato il diagramma entità relazione del database. Tuttavia, invece di adottare *MySQL*, considerando la mole di dati non elevata e per facilità di utilizzo si è scelto di adottare *SQLite3* come storage di dati. La figura è stata ottenuta automaticamente da riga di comando grazie ai tools *Pyreverse* e *GraphViz*.

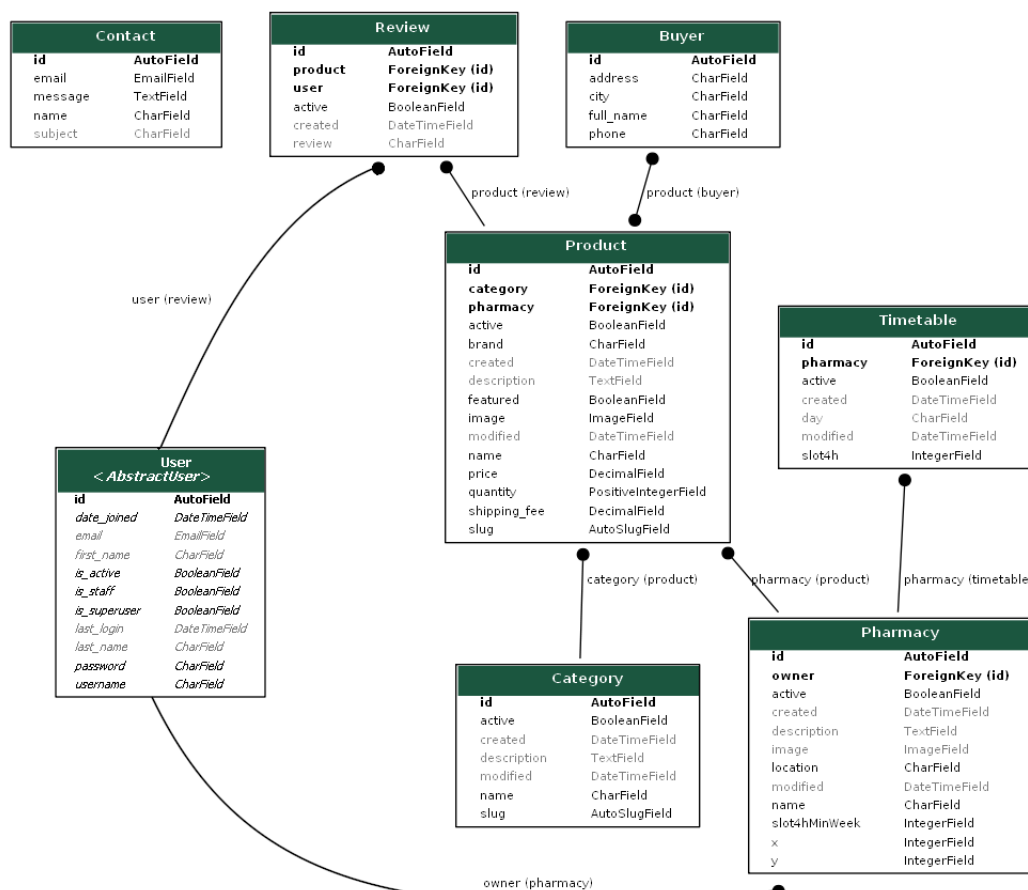


Figura 3.8.: Database ER diagram

Si è scelto che ogni tabella abbia un identificatore univoco come chiave primaria per evitare conflitti tra nomi identici. Analizziamo ora i campi principali delle varie tabelle:

Tabella User:

Modello già presente in Django contenente i vari campi riferiti ai vari utenti presenti sulla piattaforma.

- *id*: identificatore univoco

- *data_joined*: data di registrazione
- *is_active*, *is_staff*, *is_superuser*: indicano i permessi che ha l'utente, in particolare ogni account di ogni farmacia sarà *staff* e il gestore di tutte le farmacie sarà *superuser*
- *password*: password dell'account (hash)
- *username*: nome utente dell'account

Tabella Pharmacy:

Modello delle farmacie.

- *id*: identificatore univoco
- *owner*: foreign key riferita all'*id* delle tabella User
- *active*: booleano indicante se la farmacia è attiva o nascosta
- *location*: posizione geografica della farmacia
- *name*: nome della farmacia
- *slot4hMinWeek*: numero di slot minimi di 4h in cui la farmacia rimane aperta settimanalmente
- *x*: coordinata della posizione
- *y*: coordinata della posizione

Tabella Category:

Modello delle categorie dei farmaci.

- *id*: identificatore univoco
- *active*: booleano indicante se la categoria è attiva o nascosta
- *name*: nome della categoria di farmaci
- *slug*: campo automatico per identificare la categoria nella barra indirizzi

Tabella Product:

Modello dei farmaci e dei prodotti venduti.

- *id*: identificatore univoco
- *category*: foreign key riferita all'*id* delle tabella Category
- *pharmacy*: foreign key riferita all'*id* delle tabella Pharmacy
- *active*: booleano indicante se il prodotto è attivo o nascosto

- *brand*: marchio del prodotto
- *featured*: booleano indicante se il prodotto è in risalto
- *image*: immagine del prodotto
- *name*: nome del prodotto
- *price*: prezzo del prodotto
- *quantity*: quantità del prodotto
- *shipping_fee*: spese di spedizione del prodotto
- *slug*: campo automatico per identificare il prodotto nella barra indirizzi

Tabella Buyer:

Modello degli acquirenti.

- *id*: identificatore univoco
- *address*: indirizzo dell'acquirente
- *city*: città dell'acquirente
- *full_name*: nome completo dell'acquirente
- *phone*: numero di cellulare dell'acquirente

Tabella Review:

Modello delle recensioni.

- *id*: identificatore univoco
- *product*: foreign key riferita all'*id* delle tabella Product
- *user*: foreign key riferita all'*id* delle tabella User
- *review*: contenuto della recensione del prodotto

Tabella Contact:

Modello dei messaggi di contatto.

- *id*: identificatore univoco
- *email*: mail del richiedente
- *messaggio*: messaggio di richiesta
- *name*: nome del richiedente

Tabella Timetable:

Modello delle tabelle degli orari delle farmacie.

- *id*: identificatore univoco
- *pharmacy*: foreign key riferita all'*id* delle tabella Pharmacy
- *day*: giorno della settimana coperto
- *slot4h*: intero indicante quale slot della giornata copre

3.3.2. Component Shop

Introduciamo il component diagram di *shop* rappresentato nella figura 3.9.

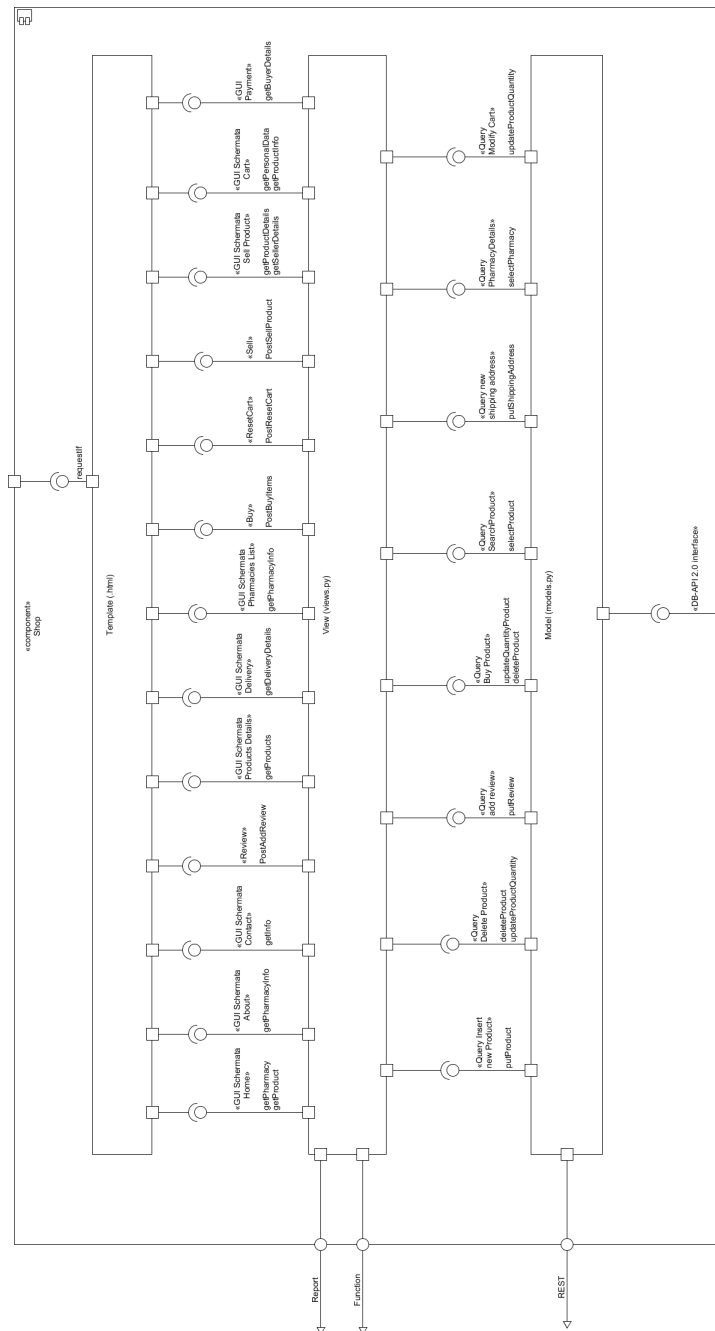


Figura 3.9.: Component diagram di shop

Nella fase successiva andremo a implementare questa componente come base dell'applicazione.

Parte II.

Fase 2

4. Implementazione

In questa fase viene implementata l'applicazione e le funzionalità di base.

4.1. Selezione funzioni da implementare

È stato scelto di implementare le funzioni con priorità maggiore richieste dal committente e rendere mock alcune a priorità più bassa. Inoltre sono state riorganizzate le categorie che raggruppavano le funzioni nel seguente modo mantenendo i codici dei casi d'uso precedenti:

Authentication

- U-UC1 Registrazione cliente (Alta)
- U-UC2 Login cliente/farmacista (Alta)
- U-UC3 Login manager (Alta)

Shop

- A-UC1 Gestione carrello dei farmaci (Alta)
- A-UC2 Prenotazione dei farmaci (Media) -> *Mock*
- A-UC3 Acquisto dei farmaci (Media) -> *Mock (Soltanto il pagamento)*
- F-UC1 Registrazione del farmaco in ingresso nel magazzino (Bassa) -> *Mock*
- F-UC2 Modifica manuale della disponibilità dei farmaci (Media)
- F-UC3 Generazione report delle vendite (Bassa) -> *Mock*
- F-UC4 Visualizzazione farmacista della disponibilità dei farmaci (Alta)
- F-UC5 Vendita nuovo prodotto (Alta)
- C-UC1 Inserimento nuova farmacia (Alta)
- C-UC2 Eliminazione farmacia (Alta)
- C-UC3 Gestione farmacia ferie/indisponibilità temporanea (Bassa) -> *Mock*

Timetable

- O-UC1 Calcolo automatico degli orari settimanali delle farmacie (Alta) -> *Algoritmo implementato nelle fasi successive*
- O-UC2 Modifica manuale degli orari settimanali delle farmacie (Bassa) -> *Mock*
- O-UC3 Visualizzazione orario settimanale di tutte le farmacie (Alta)
- O-UC4 Visualizzazione orario settimanale della singola farmacia (Media) -> *Mock*

Transfer

- F-UC6 Trasferimento medicinali (Alta) -> *Algoritmo implementato nelle fasi successive (Greedy)*

Come si può notare è stata creata una grossa componente centrale *shop*, mentre *authentication* si occupa dell'autenticazione nell'applicazione. Le altre componenti *timetable* e *transfer* verranno trattate nelle fasi successive in cui implementeremo due algoritmi.

4.2. Apps

Si vuole creare quindi un'applicazione web *pharmacies* che gestisca i vari moduli e funzionalità aggiuntive: in particolare avrà il ruolo di unire tutti gli urls e i vari models definiti nelle varie applicazioni.

L'applicazione sarà divisa in quattro apps:

- *authentication*: modulo che si occupa di tutta la parte di autenticazione e registrazione degli utenti e dei permessi relativi.
- *shop*: modulo principale che si occupa di gestire tutta la parte di presentazione, acquisto e vendita dei farmaci e della gestione delle farmacie.
- *timetable*: modulo che si occupa della presentazione e del calcolo degli orari settimanali delle farmacie (*Fase 3*).
- *transfer*: modulo che si occupa del trasferimento dei farmaci tra la farmacia (*Fase 4*).

Iniziamo sviluppare le prime due componenti in questa fase.

5. Authentication

L'app *authentication* si occupa di tutta la parte di autenticazione e registrazione degli utenti e dei permessi relativi.

In particolare gestisce tutti i form relativi alle funzioni quali: *register*, *activate*, *login*, *logout*, *password-reset*, *password-reset-confirm*, *password-reset-complete*.

È necessario configurare il file *settings.py* per abilitare la ricezione e l'invio delle email di conferma.

5.1. Tokens

La password è salvata non in chiaro ma grazie a *tokens.py* viene effettuato l'hash della password.

```
class TokenGenerate(PasswordResetTokenGenerator):
    def _make_hash_value(self, user, timestamp):
        return (
            six.text_type(user.id) + six.text_type(timestamp) + six
            ↪ .text_type(user.is_active)
        )
activation_token = TokenGenerate()
```

5.2. Permission

Nella cartella *pharmacies* vi è il file *permission.py* che si occupa di gestire alcune funzioni che permettono di attribuire permessi relativi al tipo di account dell'utente.

Ad esempio *IsStaffOrReadOnly* permette l'accesso a una determinata funzione se e solo se l'utente che richiede l'azione è membro dello staff (farmacista) oppure l'azione è solo di lettura. In particolare per limitare l'accesso a tutte le API fornite dall'applicazione vengono usate questi permessi. In *SAFE_METHODS* sono presenti i metodi di lettura quali *GET*, *HEAD*, *OPTION*.

```
class IsStaffOrReadOnly(permissions.BasePermission):
    """
        The request is authenticated as a member of a staff
        ↪ , or is a read-only request
    """
```

```
def has_permission(self, request, view):
    if request.method in SAFE_METHODS:
        return True
    elif request.user.is_staff:
        return True
    else:
        return False
```

Le possibili classi sono: *IsAdmin*, *IsStaff*, *IsAuthenticated*, *IsAdminOrReadOnly*, *IsStaffOrReadOnly*, *IsAuthenticatedOrReadOnly*.

6. Shop

L'app *shop* è il modulo principale che si occupa di gestire tutta la parte di presentazione, acquisto e vendita dei farmaci e della gestione delle farmacie.

È la parte più corposa che gestisce anche il template base per la visualizzazione del sito web.

Nella figura sono rappresentate tutte le relazioni tra i vari package presenti nell'app *shop*.

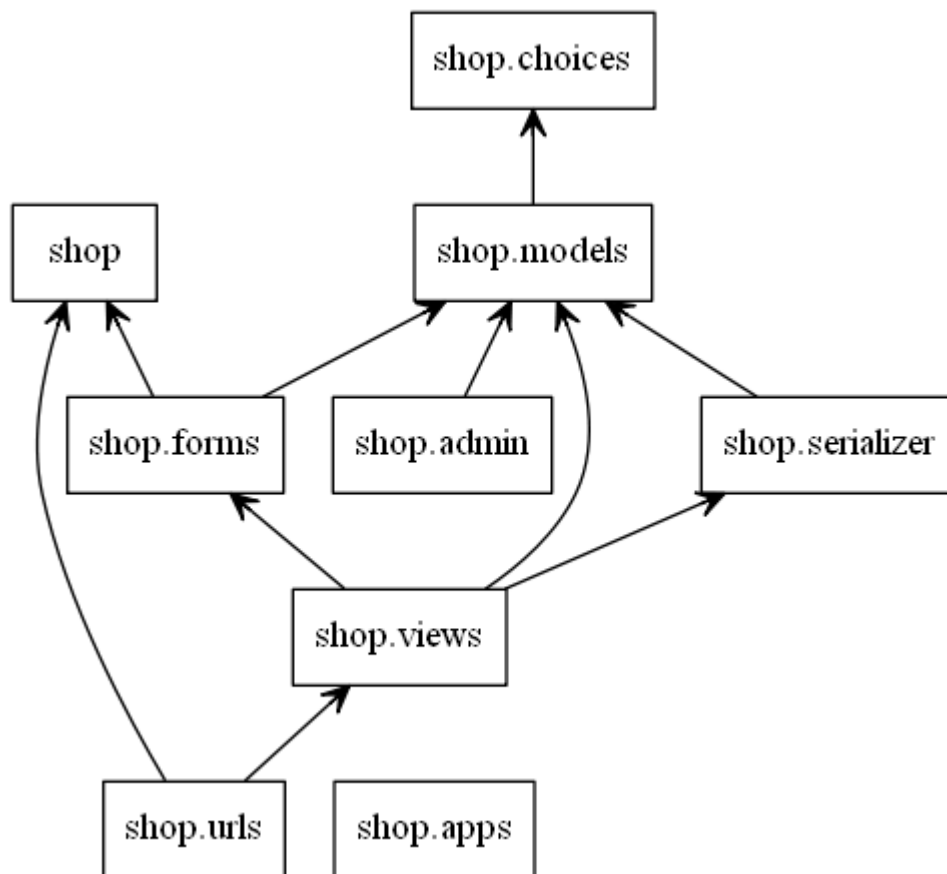


Figura 6.1.: Packages di shop

6.1. Home

In questo modulo viene definita la funzione *homepage* e il template *base.html* che si occupa di visualizzare il corpo principale del sito web.

Grazie a *Bootstrap* e *jQuery* è possibile adottare dei modelli predefiniti di costruzione per l'applicazione web.

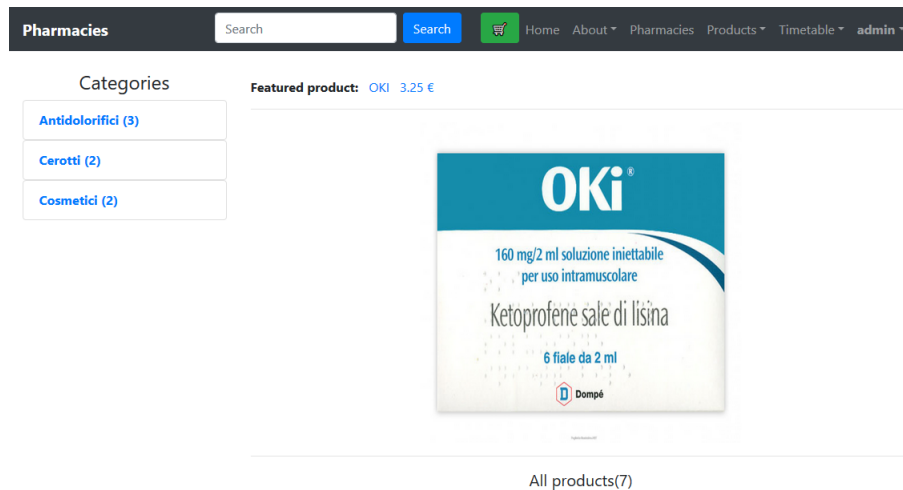


Figura 6.2.: Homepage

Nella pagina principale è presente una barra superiore che contiene lo strumento di ricerca dei prodotti e tutti i collegamenti alle varie funzionalità del sito web. In alto a destra vi è la sezione relativa all'autenticazione dell'account. Nel corpo principale della pagina sono presenti uno slider dei prodotti in evidenza, l'elenco di tutti i prodotti e l'elenco delle categorie dei prodotti a sinistra.

Procediamo ad analizzare le funzionalità presenti nella barra superiore e quale tipo di utente potrà accedervi. Per ognuna di queste sono state create specifiche funzioni e variazioni del template di base per la presentazione grafica.

6.2. About e Contact

Sono sezioni contenenti informazioni sulla catena delle farmacie e un form per la richiesta di supporto.

Tutte queste sezioni sono accessibili anche da utenti non registrati, tuttavia per inviare un messaggio è necessario inserire i propri dati.

6.3. Pharmacies

È una sezione dove vengono listate tutte le farmacie attive presenti nel database. È possibile anche visualizzare nel dettaglio il nome, la descrizione e la posizione di ogni singola farmacia.

Tutte queste sezioni sono accessibili anche da utenti non registrati.

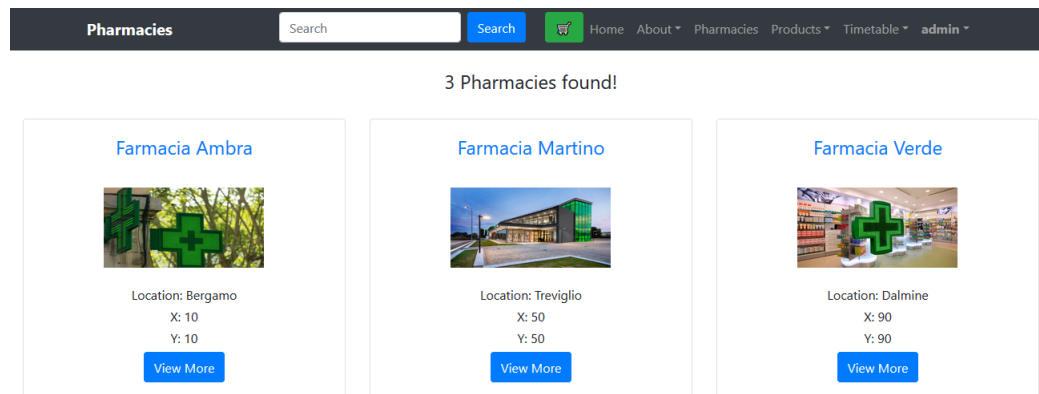


Figura 6.3.: Farmacie

6.4. Categories, Products e Search

Products è una sezione dove vengono listate tutti i farmaci presenti in tutte le farmacie attive presenti nel database. È possibile anche visualizzare nel dettaglio il nome, la descrizione, il prezzo e in quali farmacie è presente il prodotto. Oltre a ciò è possibile aggiungere al carrello il prodotto, acquistarlo direttamente o scrivere una recensione sul prodotto.

Inoltre dalla homepage è possibile listare i prodotti appartenenti a una sola farmacia oppure procedere alla ricerca di uno specifico farmaco usando la barra di ricerca presente nella barra superiore.

Tutte queste sezioni sono accessibili anche da utenti non registrati.

Nel caso in cui l'utente è parte dello staff (farmacista) potrà accedere alle seguenti sottosezioni:

- **Products:** permette di vedere la lista di tutti i prodotti presenti nelle farmacie.
- **Sell a product:** permette al farmacista di inserire velocemente un farmaco nel database compilando tutti i campi un apposito form e indicando la propria farmacia.

- **Medicine transfers:** permette al farmacista di poter richiedere il trasferimento di farmaci da altre farmacie (*Algoritmo greedy implementato nelle fasi successive*).

The screenshot shows a web application interface for selling products. At the top, there is a navigation bar with 'Pharmacies' and a search bar. Below this, the 'Sell your product' form is displayed. The form includes fields for Pharmacy, Name, Image, Category, Description, Brand, Quantity, Price, and Shipping fee, each with a corresponding input type (dropdown, text, or number). A green 'Submit' button is located at the bottom of the form.

Figura 6.4.: Sell a product form

6.5. Cart e Payment

Cart è una sezione per vedere tutti i farmaci messi nel carrello in attesa di procedere all'acquisto. Il carrello presenta un riepilogo dei prodotti aggiunti, il prezzo totale, i costi di spedizione e due bottoni: *reset cart* (svuotare il carrello) e *proceede to checkout* (procedere all'acquisto).

Procedendo all'acquisto il cliente dovrà compilare un form per indicare il proprio indirizzo di spedizione e successivamente quale metodo di pagamento scegliere tra quelli proposti.

È un componente mock, il pagamento ha una componente random che ne determina l'esito positivo o negativo della transazione.

```
def calculate_amount():
```



```
    return randint(1, 100) # Random

class Payment():
    def __init__(self, invoice_id, credit_card):
        assert isinstance(credit_card, FakeCreditCard), "
            ↪ credit_card is not a FakeCreditCard instance"
        self.credit_card = credit_card

    def process(self, request):
        amount = calculate_amount()
        assert amount >= 0, "amount should be positive"
        if self.credit_card.has_enough_credit(amount):
            self.credit_card.withdraw(amount)
            self.status = 'processed'
        else:
            self.status = 'cancelled'
            return self.status

class FakeCreditCard:
    def __init__(self, balance=50):
        assert balance >= 0, "balance should be positive"
        self.balance = balance

    def has_enough_credit(self, amount):
        return self.balance > amount

    def withdraw(self, amount):
        self.balance = self.balance - amount
        assert self.balance >= 0, "balance should be positive"
```

Questa sezione è accessibile a da chiunque ma il pagamento può essere concluso solo da utenti registrati.

6.6. Timetable

Timetable è una sezione visibile a chiunque dove è possibile visualizzare la tabella degli orari settimanali delle farmacie divise in slot di 4 ore. Solo il manager delle farmacie però potrà calcolare questi orari settimanali cliccando un bottone apposito nella tendina sulla barra superiore (*Algoritmo di calcolo implementato nelle fasi successive*).

6.7. Admin

Il manager di tutte le farmacie (avrà permessi superuser) collegandosi alla sezione */admin* già presente in Django potrà gestire tutti i dati presenti nel database e i permessi relativi agli utenti.

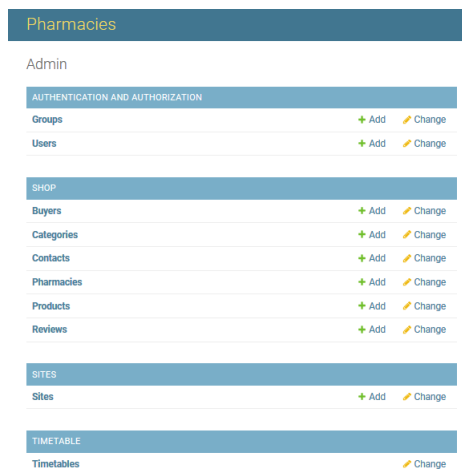


Figura 6.5.: Sezione admin

7. REST API

L'applicazione mette a disposizione una serie di *API (Application Programming Interfaces)* grazie a *Django REST Framework*.

Per poter implementare il framework si è dovuto serializzare ogni modello presente nelle varie apps e definirne i permessi per limitare l'accesso a dati sensibili.

Ad esempio il modello *User* è stato serializzato in modo da nascondere i dati sensibili quali email e password degli utenti nelle chiamate *GET*.

```
# serializer.py
class UserSerializer(serializers.ModelSerializer):
    email = serializers.CharField(write_only=True) # Hide in
    ↪ GET
    password = serializers.CharField(write_only=True) # Hide
    ↪ in GET

    class Meta:
        model = User
        fields = "__all__"
```

Inoltre si sono limitati i permessi delle varie classi a utenti solo admin in questo caso.

```
# views.py
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = [IsAdmin]
```

Tutte le *permission_classes* sono state definite nel file *permission.py* descritto in precedenza; le possibili classi sono: *IsAdmin*, *IsStaff*, *IsAuthenticated*, *IsAdminOrReadOnly*, *IsStaffOrReadOnly*, *IsAuthenticatedOrReadOnly*.

Il tool *Swagger UI*¹ permette la visualizzazione grafica delle API accedendo all'url */swagger* o */redoc* dove è possibile interagire e provare le varie richieste definite. Inoltre è possibile vedere una descrizione dettagliata dei vari modelli presenti nel database.

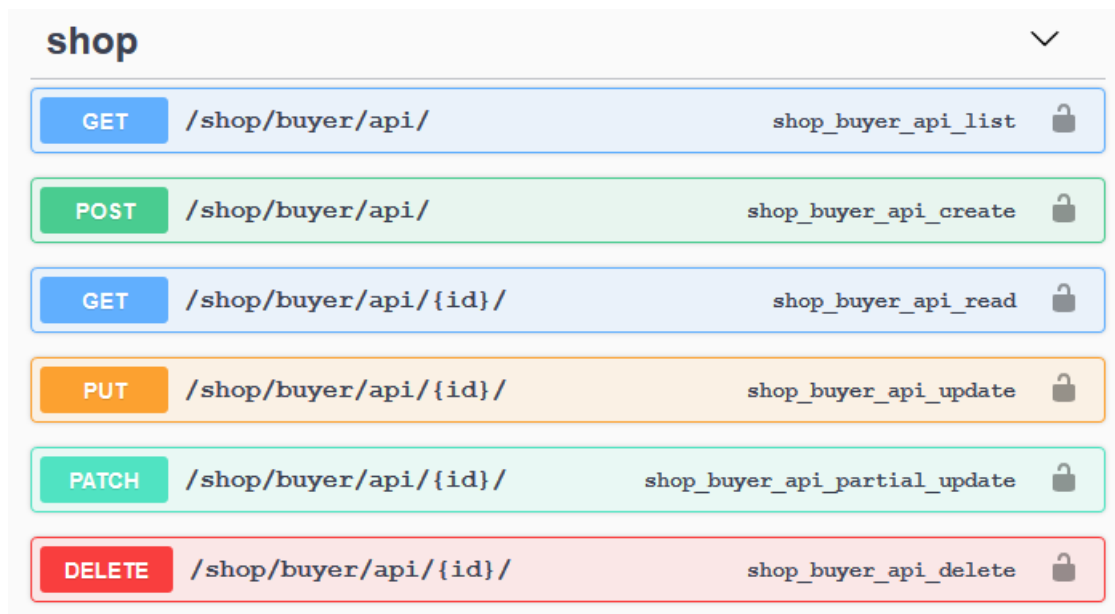


Figura 7.1.: Swagger

¹Swagger UI si basa sullo standard OpenAPI.

Nella figura 7.2 è presente un'interazione con con il GET di un prodotto dato l'identificativo univoco. Quest'operazione di sola lettura sui prodotto può essere effettuata da qualsiasi utente anche non registrato.

The screenshot displays a REST client interface with the following details:

- Request URL:** `http://127.0.0.1:8000/shop/product/api/1/`
- Server response:**
 - Code:** 200
 - Details:**
 - Response body:** A JSON object representing a product and its associated pharmacy and category. The product is named "OKI" with a price of 7.25 and is located in Bergamo. It is associated with a pharmacy named "Farmacia" and a category named "antinfiammatorio".
 - Response headers:** Includes headers for allow, content-length, content-type, date, server, vary, x-content-type-options, and x-frame-options.

```
{
  "id": 1,
  "pharmacy": {
    "id": 2,
    "owner": {
      "id": 2,
      "last_login": null,
      "is_superuser": false,
      "username": "TestUser",
      "first_name": "",
      "last_name": "",
      "is_staff": false,
      "is_active": true,
      "date_joined": "2020-01-29T19:42:20.710058Z",
      "groups": [],
      "user_permissions": []
    },
    "name": "Farmacia",
    "image": "http://127.0.0.1:8000/media/farmacia.png",
    "x": 50,
    "y": 50,
    "slot4hMinWeek": 5,
    "location": "Bergamo",
    "description": "Text",
    "active": true,
    "created": "2020-01-29T19:42:20.737982Z",
    "modified": "2020-01-29T19:42:20.737982Z"
  },
  "category": {
    "id": 2,
    "slug": "antinfiammatorio",
    "name": "Antinfiammatorio",
    "description": "Text",
    "active": true,
    "created": "2020-01-29T19:42:20.752942Z",
    "modified": "2020-01-29T19:42:20.752942Z"
  },
  "slug": "product",
  "name": "OKI",
  "image": "http://127.0.0.1:8000/media/products/oki_MocJ0qn.jpg",
  "description": "OKI in bustine",
  "brand": "OKI",
  "quantity": 15,
  "price": "7.25",
  "shipping fee": "1.50",
  "featured": true,
  "active": true,
  "created": "2020-01-29T19:23:15.209856Z",
  "modified": "2020-02-05T13:28:45.438611Z"
}
```

allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
content-length: 975
content-type: application/json
date: Wed, 05 Feb 2020 13:29:16 GMT
server: WSGIServer/0.2 CPython/3.8.1
vary: Accept, Cookie
x-content-type-options: nosniff
x-frame-options: DENY

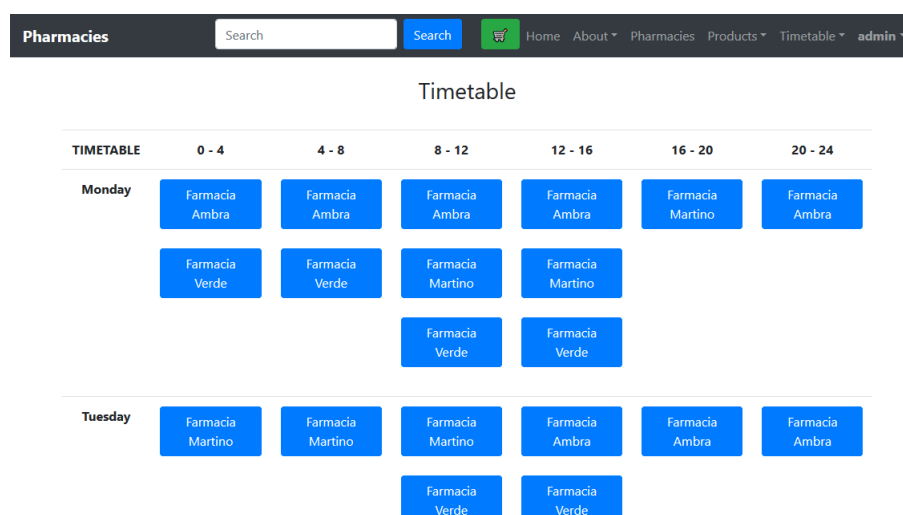
Figura 7.2.: Classes

Parte III.

Fase 3

8. Timetable

In questa fase è stata implementata l'app *timetable* che contiene un algoritmo per il calcolo degli orari settimanali delle farmacie.



The screenshot shows a web application interface for a pharmacy timetable. At the top is a dark navigation bar with the text 'Pharmacies', a search bar, a 'Search' button, a shopping cart icon, and a list of menu items: 'Home', 'About', 'Pharmacies', 'Products', 'Timetable', and 'admin'. Below the navigation bar is the title 'Timetable'. The main content area is a table with the following structure:

TIMETABLE	0 - 4	4 - 8	8 - 12	12 - 16	16 - 20	20 - 24
Monday	Farmacia Ambra	Farmacia Ambra	Farmacia Ambra	Farmacia Ambra	Farmacia Martino	Farmacia Ambra
	Farmacia Verde	Farmacia Verde	Farmacia Martino	Farmacia Martino		
			Farmacia Verde	Farmacia Verde		
Tuesday	Farmacia Martino	Farmacia Martino	Farmacia Martino	Farmacia Ambra	Farmacia Ambra	Farmacia Ambra
			Farmacia Verde	Farmacia Verde		

Figura 8.1.: Timetable

8.1. Descrizione

Si è deciso di implementare la funzione che si occupa di generare automaticamente l'orario settimanale di apertura delle farmacie.

Ogni volta che l'amministratore (che è l'unico utente abilitato per eseguire questa funzionalità) decide di ricalcolare il piano di apertura delle farmacie, il piano precedentemente calcolato viene eliminato dal DB.

La funzione tiene conto dei seguenti vincoli (come da specifica):

1. Ogni farmacia deve stare aperta almeno per j slot temporali alla settimana (uno slot consiste in un blocco di 4 ore contigue, suddivisibili tra i seguenti orari (6 slot giornalieri)).
2. Ogni farmacia può stare aperta al più 42 slot a settimana, che consiste nel rimanere aperta h 24 per 7 giorni.

3. Ad ogni slot settimanale dev'esserci almeno una farmacia aperta; tale requisito viene soddisfatto se e solo se la somma degli slot di tutte le farmacie di un determinato piano settimanale è ≥ 42 .

Sia $M = \sum_{i=1}^K SlotMin$, possiamo avere 3 casistiche possibili:

1. $M = 42$: in questo caso ogni farmacia sta aperta per un numero di slot pari al suo minimo, e non si avranno sovrapposizioni.
2. $M > 42$: in questo caso si ha una sovrapposizione per $(M - 42)$ slot settimanali. Tali sovrapposizioni si dovranno verificare maggiormente nelle ore diurne (quindi dalle 08 alle 20).
3. $M < 42$: in questo caso, essendo solo slot minimi, una farmacia può restare aperta più slot rispetto al suo minimo. Quindi, dovremo imporre ad alcune/tutte le farmacie (privilegiando le farmacie che hanno un minimo di slot minore) di essere aperte più slot temporali, per soddisfare il vincolo terzo vincolo descritto in precedenza.

Sia K il numero di farmacie, j_i il numero minimo di slot dell' i -esima farmacia, n_i il numero effettivo di slot in cui l' i -esima farmacia resterà aperta nel piano calcolato (al massimo sappiamo che sarà ≤ 42).

$$\sum_{i=1}^K n_i \geq 42 \quad (8.1)$$

$$\forall i : n_i \geq j_i \quad (8.2)$$

$$\forall j : j_i \leq 42 \quad (8.3)$$

8.2. Pseudocodice

Algoritmo 8.1 Pseudocodice di timetable

CalculateTimetable (lista di farmacie F) -> void

delete(timetable)

calculate (SumOfSlotMin)

if (SumOfSlotMin == 42) [1]

K=List[0...41]

foreach Pharm∈F

for i=0 to Pharm.slot-1

s=random.choice(K)

K.remove(s)

insert(item)

else if (SumOfSlotMin > 42) [2]

count=0

K=List[0...41]

foreach Pharm∈F

for i=0 to Pharm.slot-1

if (count < 42)

s=random.choice(K)

K.remove(s)

insert(item)

count++

else

while (slot is already full)

chooseSlot()

while (day in slot is already full)

insert(item)

else [3]

K=List[0...41]

for i=0 to 41 do L[i] <- 0

foreach Pharm∈F

for i=0 to Pharm.slot-1

s=random.choice(K)

K.remove(s)

insert(item)

while (K != empty)

s=random.choice(K)

choosePharmacy()

insert(item)

K.remove(s)

8.3. Analisi di complessità

Ora analizziamo la complessità dell'algoritmo ponendoci nel caso peggiore. Mettiamoci nel caso generale, cioè che il numero di slot settimanali da riempire siano n (nel nostro caso sappiamo 42). Prima di entrare in una delle tre casistiche viene effettuata la cancellazione delle n tuple dal database $O(n)$ e vengono calcolati gli slot minimi totale scorrendo tutte le farmacie $O(n)$. Supponiamo che le operazioni di cancellazione e inserimento nel database abbiano costo costante $O(1)$.

- Situazione con $SumOfSlotMin = n$ [1]

Vi è un doppio ciclo for: quello esterno itera sulle farmacie e quello interno sugli slot minimi delle farmacie. Tuttavia sapendo che ci saranno solo n slot minimi delle farmacie, il ciclo for interno itererà n volte (infatti nel caso una farmacia abbia slot minimi pari a zero il ciclo interno non itera). Considerando le due operazioni precedenti (*delete* e *calculate*) avremo una complessità pari ad $O(n) + O(n) + O(n) = O(n)$.

- Situazione con $SumOfSlotMin = m > n$ [2]

Definiamo $d = m - n$, ossia il numero di slot minimi aggiuntivi ai 42 settimanali.

Come prima cosa riempiamo gli n slot settimanali obbligatori, con complessità pari a sopra, cioè $O(n)$; per i restanti d avremo che il doppio ciclo while itererà al massimo n volte per ogni d e quindi avrà complessità $O(d * n)$. Considerando le due operazioni precedenti (*delete* e *calculate*) avremo una complessità pari ad $O(n) + O(n) + O(d * n) = O(d * n)$.

- Situazione con $SumOfSlotMin = m < n$ [3]

Definiamo $f = n - m$, ossia il numero di slot minimi mancanti che dovremo riempire.

Il doppio ciclo for effettuerà m iterazioni, quindi avremo una complessità pari ad $O(m)$.

Siccome dobbiamo riempire i restanti $n - m$ slot, lo facciamo con un while che itera f volte e avrà complessità $O(f)$. Considerando le due operazioni precedenti (*delete* e *calculate*) avremo una complessità pari ad $O(n) + O(n) + O(m) + O(f) = O(n)$.

La complessità dell'algoritmo nel caso peggiore è il secondo caso; infatti $f = n - m$ del terzo caso è limitata perchè non può andare oltre a n , invece $d = m - n$ può divergere se m tende all'infinito (tantissime sovrapposizioni). Quindi la complessità massima è $O(d * n)$.

8.4. Activity diagram

Nella figura 8.2 è rappresentato il flow chart dell'algoritmo di *timetable*.

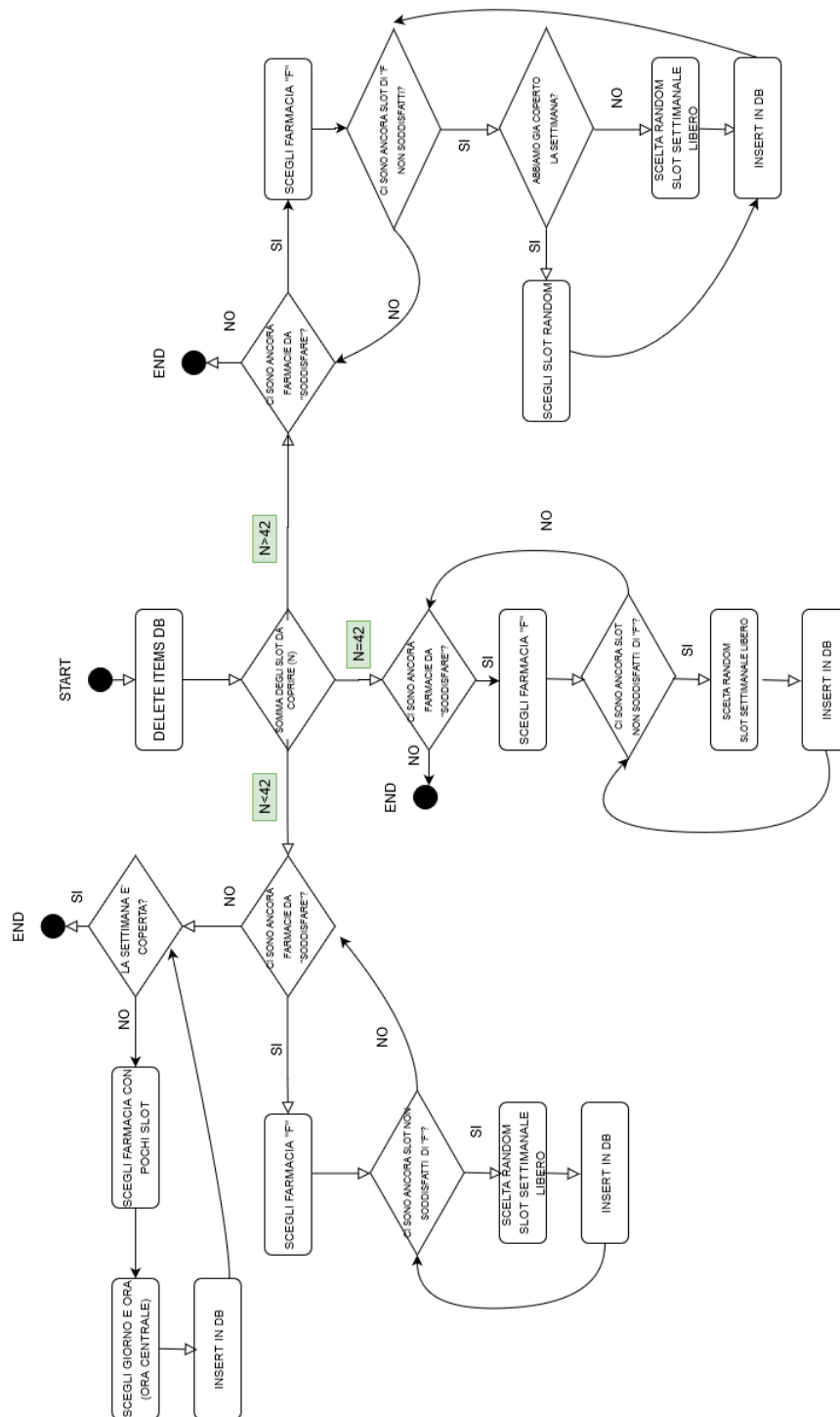


Figura 8.2.: Flow chart dell'algoritmo di timetable

8.5. Testing dell'algoritmo

Ora procediamo con la fase di testing dell'algoritmo per assicurarci che l'algoritmo funzioni nella maniera corretta.

Premettiamo che è la funzione *calculate(request)* che si occupa di chiamare l'algoritmo e prima di chiamarlo svolge un controllo sul numero di farmacie presenti nel database e sul fatto che sia l'admin ad essere loggato.

In particolare avremo due casistiche:

- se non è l'admin ad essere autenticato: stamperà a video *'You have to logged as admin in first to calculate the timetable'* e non chiamerà l'algoritmo.
- se è l'admin ad essere loggato:
 - se il numero di farmacie presenti nel database è > 3 , chiamerà l'algoritmo.
 - altrimenti: stamperà a video *'Three pharmacies are needed in order to calculate the timetable'* e non chiamerà l'algoritmo.

Per quanto riguarda la funzione *algorithm_timetable(request)*, essendo che non riceve nessun parametro in ingresso e non ne restituisce (opera il tutto nel database), ci possiamo limitare a controllare che, una volta eseguita, siano presenti nel database il numero giusto di tuple nella tabella timetable relative alle varie farmacie.

8.6. Struttura di un caso di test

Tutti i casi di test hanno la seguente struttura, in cui N indica l'indice del test:

```
class TimetableTestN(TestCase):
    def test_algorithm_TimetableN(self):
        <<inizializzazione>>
        <<algoritmo>>
        <<test>>
```

Si ha che <<inizializzazione>> indica l'inizializzazione delle variabili di input dell'algoritmo, <<algoritmo>> indica la chiamata dell'algoritmo con i valori di input scelti precedentemente e infine <<test>> si controlla che i risultati di output dell'algoritmo siano corretti.

Nel codice sorgente sono presenti 3 casi di test per coprire tutte le diverse casistiche: il caso in cui gli slot minimi di tutte le farmacie siano minori di 42, esattamente uguali a 42 o maggiori a 42 dove 42 è la copertura settimanale.

Il codice sottostante rappresenta il caso in cui la somma degli slot minimi delle farmacie sia minore di 42; in questo caso, come ampiamente spiegato sopra, l'algoritmo dev'essere in grado di coprire gli tutti gli slot settimanali; quindi siccome ogni slot è rappresentato da una tupla differente, nel database dovremo avere 42 tuple esatte.

```

class AlgoritmoCalculateTimetable1(TestCase):
    def test_algorithm_calculate1(self):
        user1 = User.objects.create(username='Testuser1')
        user2 = User.objects.create(username='Testuser2')
        user3 = User.objects.create(username='Testuser3')

        farmacia1 = Pharmacy.objects.create(owner=user1,
            ↪ name="farmacia1", image="farmacia.png", x=50,
            ↪ y=50, slot4hMinWeek=5, location="Bergamo",
            ↪ description="Text")
        farmacia2 = Pharmacy.objects.create(owner=user2,
            ↪ name="farmacia2", image="farmacia.png", x=50,
            ↪ y=50, slot4hMinWeek=5, location="Bergamo",
            ↪ description="Text")
        farmacia3 = Pharmacy.objects.create(owner=user3,
            ↪ name="farmacia3", image="farmacia.png", x=50,
            ↪ y=50, slot4hMinWeek=5, location="Bergamo",
            ↪ description="Text")

        farmacia1.save()
        farmacia2.save()
        farmacia3.save()

        algorithm_timetable(self)

        count = Timetable.objects.all().count()
        self.assertEqual(count, 42)

```

Essendo che l'algoritmo non riceve input ne produce output (tranne l'inserimento di tuple nel database), si ha agito in questo modo: prima di tutto si è istanziato le 3 farmacie, facendo sì che la somma degli slot min sia minore di 42 (nel nostro caso 15).

Viene chiamato l'algoritmo e successivamente andiamo a contare il numero di tuple nella tabella Timetable (ricordiamo che ogni volta che la funzione viene eseguita, come prima cosa vengono eliminate tutte le tuple presenti (cioè viene eliminato il vecchio piano calcolato la volta precedente).

```
self.assertEqual(count, 42)
```

Questo comando serve per effettuare il test vero e proprio sui risultati ottenuti dall'algoritmo. Il comando *self.assertEqual(argomento1, argomento2)* riceve due argomenti e controlla che siano uguali i valori. Inserendo al posto di argomento2 il valore che ci si aspetta (nel nostro caso 42 (numero di tuple)), si verifica se l'algoritmo funziona correttamente o meno.

Allo stesso modo si effettuano i test per i restanti due casi, contando anche singolarmente gli slot minimi delle farmacie.

```
c1 = Timetable.objects.filter(pharmacy=farmacia1).count()
c2 = Timetable.objects.filter(pharmacy=farmacia2).count()
c3 = Timetable.objects.filter(pharmacy=farmacia3).count()
count = Timetable.objects.all().count()

self.assertEqual(count, 80)
self.assertEqual(c1, 20)
self.assertEqual(c2, 25)
self.assertEqual(c3, 35)
```


Parte IV.

Fase 4

9. Transfer

In questa fase si è implementata l'app *transfer* che contiene un algoritmo greedy per il calcolo del trasferimento dei farmaci.

Pharmacies

Search

Search

Home About Pharmacies Products Timetable admin

Your transfers request has been sent!

Transfer request

Posizione GPS (X: 8, Y: 86)
Total quantity: 95

Pharmacies	[2] Farmacia Martino	[3] Farmacia Verde	[1] Farmacia Ambra
Quantity	Q: 35	Q: 45	Q: 15

Book now

Figura 9.1.: Transfer

9.1. Descrizione

L'algoritmo si prefigge di trovare un percorso che permetta di ottenere al farmacista la quantità richiesta di farmaci percorrendo un cammino visto come serie di farmacie presenti nel territorio. A ogni iterazione la farmacia verrà scelta tra le N disponibili, dalle quali verranno rimosse quelle già selezionate in precedenza. Ogni farmacia ha una posizione indicata con le coordinate (x,y) e ha a disposizione una nota quantità del farmaco richiesto. Viene tenuto conto anche della posizione iniziale in cui viene formulata la richiesta di trasferimento dei farmaci.

L'algoritmo avrà quindi i seguenti ingressi e uscite:

INPUT: Quantità farmaci richiesta con relativa categoria dei farmaci e posizione iniziale del cliente (data dal GPS, componente mock quindi random)

[QuantitàFarmaco, CategoriaFarmaco]

OUTPUT: Percorso da seguire come serie di farmacie

[Farmacia i-esima, PosizioneFarmacia, QuantitàFarmacoFarmacia]

Inoltre affinché l'algoritmo funzioni sono necessarie le seguenti precondizioni:

1. Non deve esistere una farmacia che da sola possa soddisfare la richiesta della quantità di farmaci (altrimenti basterebbe solo una scelta e non vi sarebbe un algoritmo).
2. La richiesta della quantità di farmaci deve poter essere soddisfatta dalle farmacie (altrimenti è irrealizzabile l'algoritmo).

$$\forall Farmacia : \sum_{Farmaci} \leq QuantitàRichiesta \quad (9.1)$$

$$\sum_{Farmacie} \sum_{Farmaci} \geq QuantitàRichiesta \quad (9.2)$$

9.2. Criterio scelta

Le farmacie restanti tra cui poter scegliere (quindi non ancora selezionate negli step precedenti) avranno un determinato valore della variabile *weight*, espressa come:

$$weight = \frac{N^{\circ}FarmaciDisponibili}{\sqrt{(x - x_i)^2 + (y - y_i)^2}}$$

dove $N^{\circ}FarmaciDisponibili$ è la quantità in dotazione nella i -esima farmacia del farmaco in questione e il denominatore è il calcolo della distanza tra due punti (teorema di Pitagora).

Questo rapporto che ha unità di misura $\frac{Farmaci}{Km}$ indica qual è la farmacia (x_i, y_i) che, considerando la posizione attuale in cui il richiedente si trova (x, y) , garantisce il maggior numero di farmaci da ricevere in funzione della distanza da percorrere.

Ad esempio, se il richiedente si trovasse in posizione $(0,0)$ e volesse 80 farmaci con due farmacie tra cui scegliere chiamate *farmacia1* in posizione $(x1=75, y1=50)$ con disponibilità di farmaci pari a 50 e *farmacia2* in posizione $(x2=40, y2=60)$ con disponibilità di farmaci pari a 45 avrò che *farmacia1* avrà un peso $w1=0.554$ farmaci/km e che *farmacia2* avrà un peso $w2=0.624$ farmaci/km.

La scelta golosa locale consiste nel compiere l' i -esima scelta della farmacia a cui andare in modo da massimizzare il valore del *weight* (scegliere sempre il candidato più promettente); scegliere il *weight* maggiore equivale ad andare alla farmacia che mi garantisce il maggior numero di farmaci disponibili in funzione della distanza che bisogna percorrere per arrivarci. Nel caso del nostro esempio sceglieremo quindi $w2$ essendo $w2 > w1$, ciò mi garantisce un maggior numero di farmaci al km.

9.3. Pseudocodice

Algoritmo 9.1 Pseudocodice di transfer

TransferMedicinali (quantità richiesta Q, categoria richiesta C) -> lista farmacie R

```
R <- 0
if (!vincolo2(Q,C)) return R
F <- lista di tutte le farmacie
while (Q > 0) do
  W <- calcolaPesi(F,C)
  X <- seleziona(F,W)
  F <- F - {x}
  R <- R ∪ {x}
  Q <- Q - X.quantityProduct
return R
```

I vincoli 1 e 2 sono le precondizioni definite in precedenza: in particolare il vincolo 1 è labile e l'algoritmo funziona ugualmente restituendo una sola farmacia da dove prendere tutta la quantità di farmaci necessaria. R rappresenta la lista di farmacie soluzioni in sequenza dell'algoritmo e viene inizializzata come vuota. Entrando più nello specifico delle righe di codice si ha che:

Algoritmo 9.2 Transfer :: vincolo2

vincolo2 (quantità richiesta Q, categoria richiesta C) -> Boolean

```
count <- 0
foreach product in Product
  if (product.category==C)
    count += product.quantity
if (count>Q) return false
else return true
```

Algoritmo 9.3 Transfer :: calcolaPesi

calcolaPesi (lista farmacie F, categoria richiesta C) -> lista pesi W

```
W <- 0
x, y: posizione del richiedente
foreach pharm in F:
  foreach product in Product:
    if (product.category==C && product.pharmacy == pharm)
      quantitàProdottiC <- product.quantity
      weight <- (quantitàProdottiC)/sqrt( (pharm.x2 - x12) +
(pharm.y2-y12) )
      W <- W ∪ {weight}
return W
```

Algoritmo 9.4 Transfer :: seleziona

```

seleziona (lista farmacie F, lista pesi W) -> farmacia X
  max <- W[0]
  indice <- 0
  for i ∈ idPharm:
    if (max < weight )
      indice <- i
      max <- W[indice]
  return F[indice]

```

9.4. Analisi di complessità

Ora analizziamo la complessità dell'algoritmo ponendoci nel caso peggiore. Supponiamo di costo costante le operazioni di lettura dal database e di gestione delle liste $O(1)$.

Poniamo p il numero di prodotti, f il numero di farmacie.

Avremo che:

- Complessità del *vincolo2*: il ciclo for scorre tutti i prodotti presenti nel database, la complessità è $O(p)$.
- Complessità del ciclo while: al massimo pari al numero di farmacie (ossia scorrendole tutte fino all'ultima), infatti se non esistesse abbastanza quantità disponibile nelle farmacie si uscirebbe subito grazie al vincolo 2; la complessità è $O(f)$.
 - All'interno del while la funzione *calcolaPesi()* ha due cicli for in cui quello esterno scorre tutte le farmacie presenti all'interno della lista F e il ciclo interno scorre tutti i prodotti ogni volta; la complessità nel caso peggiore è $O(f * p)$.
 - All'interno del while la funzione *seleziona()* ha un solo ciclo for che scorre tutti gli identificativi delle farmacie appartenenti alla lista F; la complessità nel caso peggiore è $O(f)$.

La complessità totale del ciclo while è $O(f) * (O(f * p) + O(f)) = O(f^2 * p) + O(f^2) = O(f^2 * p)$.

La complessità totale dell'algoritmo nel caso peggiore è $O(p) + O(f^2 * p) = O(f^2 * p)$.

9.5. Activity diagram

Nella figura 9.2 è rappresentato il flow chart dell'algoritmo di transfer.

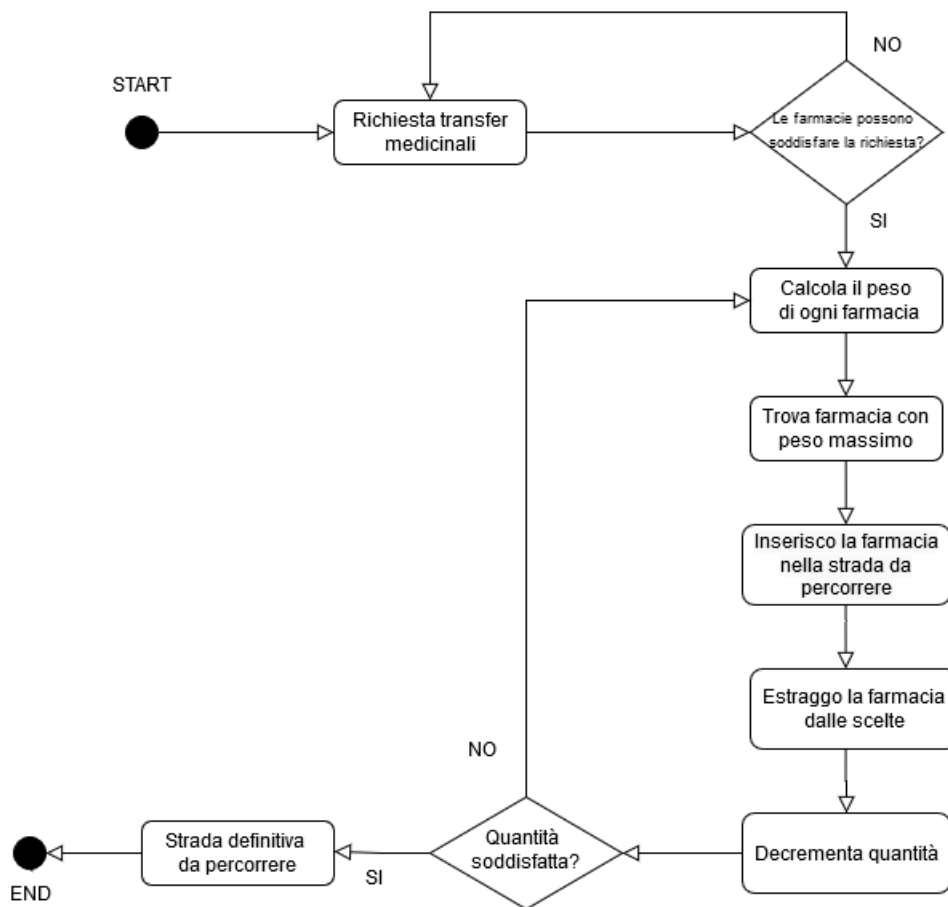


Figura 9.2.: Flow chart dell'algoritmo di transfer

9.6. Testing dell'algoritmo

Ora procediamo con la fase di testing per assicurarci che l'algoritmo funzioni nella maniera corretta.

Premettiamo che è la funzione *transfer(request)* che si occupa di chiamare l'algoritmo e prima di chiamarlo svolge un controllo sulla variabile quantitàRichiesta inserita dall'utente. In particolare avremo due casistiche:

- se la quantità presente nelle farmacie è insufficiente: stamperà a video *'There is not enough products in all pharmacies'* e non chiamerà l'algoritmo.
- se la quantità presente nelle farmacie è sufficiente: chiamerà l'algoritmo e successivamente stamperà a video *'Your transfers request has been sent!'*.

La prima casistica è coperta esternamente dall'unica funzione che invoca l'algoritmo, quindi non ci occuperemo di inserire valori in input che non possano essere soddisfatti dalle farmacie.

9.7. Struttura di un caso di test

Tutti i casi di test hanno la seguente struttura, in cui N indica l'indice del test:

```
class TransferTestN(TestCase):
    def test_algorithm_TransferN(self):
        <<inizializzazione>>
        <<algoritmo>>
        <<test>>
```

Si ha che <<inizializzazione>> indica l'inizializzazione delle variabili di input dell'algoritmo, <<algoritmo>> indica la chiamata dell'algoritmo con i valori di input scelti precedentemente e infine <<test>> si controlla che i risultati di output dell'algoritmo siano corretti.

Nel codice sorgente sono presenti 7 casi di test per coprire tutte le diverse casistiche. In questa sezione allegheremo uno dei più rilevanti dei 7 casi di test e evidenzieremo le caratteristiche del suo funzionamento commentando le linee salienti del codice.

In questo caso si è cercato il trade-off che facesse cambiare la prima scelta su quale farmacia selezionare e coerentemente con il funzionamento dell'algoritmo si è constatato che per valori di quantità di prodotti di farmacia1 minori o uguali a 669 si ha che la prima farmacia scelta è la farmacia2 e invece per valori di quantità di prodotti di farmacia1 maggiori o uguali a 670 si ha che la prima farmacia scelta è la farmacia1.

Infatti procedendo con i calcoli matematici si ha:

$$w1 = \frac{N^{\circ}Farmaci}{\sqrt{(x-x_i)^2+(y-y_i)^2}} = \frac{670}{\sqrt{(50-0)^2+(50-0)^2}} = 9.47$$

$$w2 = \frac{N^{\circ}Farmaci}{\sqrt{(x-x_i)^2+(y-y_i)^2}} = \frac{201}{\sqrt{(50-65)^2+(50-65)^2}} = 9.47$$

Per questi valori si ha parità di pesi e quindi è indifferente compiere una scelta o l'altra. L'algoritmo per come implementato prenderà quella con indice minore e sceglierà la farmacia1.

Qui la parte di codice <<inizializzazione>> del caso 7:

```
xScelta=50
yScelta=50
quantitaScelta =1000
categoriaScelta= Category.objects.create(name=" antinfiammatori ",
                                          description="Text ")
```



```
user1 = User.objects.create(username='Testuser1')
user2 = User.objects.create(username='Testuser2')
user3 = User.objects.create(username='Testuser3')
user4 = User.objects.create(username='Testuser4')

farmacia1= Pharmacy.objects.create(owner=user1, name="farmacia1"
    ↪ ,
    image="farmacia1.png", x=0, y=0,slot4hMinWeek
    ↪ =5,
    location="Bergamo",description="Text")
farmacia2= Pharmacy.objects.create(owner=user2, name="farmacia2"
    ↪ ,
    image="farmacia2.png", x=65, y=65,slot4hMinWeek
    ↪ =5,
    location="Bergamo",description="Text")
farmacia3= Pharmacy.objects.create(owner=user3, name="farmacia3"
    ↪ ,
    image="farmacia3.png", x=22, y=22,slot4hMinWeek
    ↪ =5,
    location="Bergamo",description="Text")
farmacia4= Pharmacy.objects.create(owner=user4, name="farmacia4"
    ↪ ,
    image="farmacia4.png", x=45, y=50,slot4hMinWeek
    ↪ =5,
    location="Bergamo",description="Text")

farmacia1.save()
farmacia2.save()
farmacia3.save()
farmacia4.save()

prodotto1=Product.objects.create(name="prodotto1",
    category=categoriaScelta, pharmacy=farmacia1,
    image="prodotto1.png", description="Text",
    brand="brand", quantity=670, price=10,
    ↪ shipping_fee=2)
prodotto2=Product.objects.create(name="prodotto2",
    category=categoriaScelta, pharmacy=farmacia2,
    image="prodotto2.png", description="Text",
    brand="brand", quantity=201, price=10,
    ↪ shipping_fee=2)
prodotto3=Product.objects.create(name="prodotto3",
    category=categoriaScelta, pharmacy=farmacia3,
    image="prodotto3.png", description="Text",
    brand="brand", quantity=290, price=10,
    ↪ shipping_fee=2)
```

```

prodotto4=Product.objects.create(name="prodotto4",
                                   category=categoriaScelta, pharmacy=farmacia4,
                                   image="prodotto4.png", description="Text",
                                   brand="brand", quantity=10, price=10,
                                   ↪ shipping_fee=2)

prodotto1.save()
prodotto2.save()
prodotto3.save()
prodotto4.save()

```

Nella parte iniziale sono inserite le richieste: si indica la posizione iniziale $[xScelta, yScelta]$, la quantità di farmaco voluta $[quantitàScelta]$ e $[categoriaScelta]$ che è la categoria del farmaco richiesto.

In seguito sono stati creati tutti gli input dell'algoritmo: si sono istanziate 4 farmacie e inserite nel database; nell'istanziare le farmacie è richiesto un owner che è stato referenziato con 4 user differenti; poi si sono istanziati i 4 prodotti, ognuno corrispondente alla rispettiva farmacia e inseriti nel database.

Qui la parte di codice `<<algoritmo>>` del caso 7:

```

doppia=algorithm_transfer(categoriaScelta, quantitaScelta,
                           ↪ xScelta, yScelta)

```

Viene invocato l'algoritmo e viene salvato il risultato nel vettore di due elementi chiamato *doppia*: è formato da *doppia[0]* che contiene l'ID delle farmacie usate e *doppia[1]* che contiene la quantità presa dalla farmacia usata.

Qui la parte di codice `<<test>>` del caso 7:

```

self.assertEqual(doppia[0], [farmacia1.id, farmacia3.id,
                              ↪ farmacia2.id])
self.assertEqual(doppia[1], [670, 290, 40])

```

Questi comandi servono per effettuare il test vero e proprio sui risultati ottenuti dall'algoritmo. Il comando *self.assertEqual(argomento1, argomento2)* riceve due argomenti e controlla che siano uguali i valori. Inserendo al posto di argomento2 il valore che ci si aspetta, si verifica se l'algoritmo funziona correttamente o meno.

Parte V.

Testing e verifica del software

10. Code Testing

La parte di testing sugli algoritmi è presente alla fine della *Fase III* per l'algoritmo *timetable* e alla fine della *Fase IV* per l'algoritmo *transfer*.

TODO (Algoritmi)

10.1. Test (unittest)

In questa sezione ci si concentrerà sulla creazione e l'implementazione di numerosi test sulle varie componenti dell'applicazione con particolare attenzione ai models, ai forms e alle views.

Per effettuare i test useremo *unittest*, un framework per il testing ispirato a *JUnit*. In particolar modo, nel file *settings.py* dell'applicazione è stata definita la creazione automatica di un database di prova *testdatabase* in modo da non avere ripercussioni sul database reale: ad ogni sessione di test verrà creato da zero un database sulla base dei models definiti e verrà distrutto al termine dei test.

10.1.1. Models tests

Sono stati eseguiti numerosi test sulla creazione di ogni modello presente nelle varie componenti dell'applicazione.

Un esempio di caso di test è il seguente per la creazione di una farmacia:

```
class PharmacyModelTest(TestCase):
    def create_pharmacy(self, name="Farmacia", image="farmacia.
        ↪ png", x=50, y=0, slot4hMinWeek=5, location="Bergamo",
        ↪ description="Text"):
        user = User.objects.create(username='TestUser')
        return Pharmacy.objects.create(owner=user, name=name,
            ↪ image=image, x=x, y=y, slot4hMinWeek=slot4hMinWeek,
            ↪ location=location, description=description)

    def test_pharmacy_creation(self):
        w = self.create_pharmacy()
        self.assertTrue(isinstance(w, Pharmacy))
        fields = w.id, w.owner, w.name
        self.assertEqual(w.__unicode__(), fields)
```

Un altro esempio di caso di test riguardante la creazione di un prodotto:

```
class ProductModelTest(TestCase):
    def create_product(self, name="Oki", image="pharmacy.png",
        ↪ description="Text", brand="Brand", quantity=30, price
        ↪ =20, shipping_fee=10):
        user = User.objects.create(username='TestUser')
        farmacia = Pharmacy.objects.create(owner=user, name="
            ↪ Farmacia", image="farmacia.png", x=50, y=50,
            ↪ slot4hMinWeek=5, location="Bergamo", description="
            ↪ Text")
        categoria = Category.objects.create(name="
            ↪ Antinfiammatorio", description="Text", slug=slugify
            ↪ ("Antinfiammatorio").__str__())
        return Product.objects.create(name=name, category=
            ↪ categoria, pharmacy=farmacia, image=image,
            ↪ description=description, brand=brand, quantity=
            ↪ quantity, price=price, shipping_fee=shipping_fee,
            ↪ slug=slugify(1).__str__())

    def test_product_creation(self):
        w = self.create_product()
        self.assertTrue(isinstance(w, Product))
        fields = w.id, w.name, w.description
        self.assertEqual(w.__unicode__(), fields)
```

In totale sono state create 12 classi per testare i models (compresa la funzione `__str__`), ognuna con i rispettivi metodi e parametri per verificare la corretta istanziazione di un modello nel database.

10.1.2. Forms tests

Sono stati eseguiti numerosi test sulla compilazione di ogni form presente nelle varie componenti dell'applicazione.

Un esempio di caso di test è il seguente per la compilazione del *ContactForm*:

```
class ContactFormTest(TestCase):
    def test_valid_form(self):
        data = {'name': "Test", 'email': "test@mail.com", '
            ↪ subject': "Text", 'message': "Text"}
        form = ContactForm(data=data)
        self.assertTrue(form.is_valid())

    def test_invalid_form(self):
        data = {'name': "Test", 'email': "testmail.com", 'subject
            ↪ ': "Text", 'message': "Text"}
```

```
form = ContactForm(data=data)
self.assertFalse(form.is_valid())
```

In totale sono state create 15 classi per testare i forms (anche parametrici), ognuna con i rispettivi metodi per verificare la corretta compilazione dei campi e l'invio del form. In particolare, un metodo testa un caso positivo in cui sono passati correttamente tutti i parametri necessari, e un altro metodo testa un caso negativo in cui si testa il rifiuto dell'invio del form dovuto alla mancanza di campi da compilare oppure a parametri passati non correttamente.

È stata fatta successivamente anche una versione parametrizzata (vedi sezione *Test parametrici*) molto più completa.

10.1.3. Views tests

Sono stati eseguiti numerosi test sulle varie funzioni dell'applicazione.

Un esempio di caso di test è il seguente che verifica se la pagina di dettaglio relativa a una farmacia è raggiungibile:

```
class ViewTest(TestCase):
    def test_pharmacies_detail(self):
        user = User.objects.create(username='TestUser')
        farmacia = Pharmacy.objects.create(owner=user, name="
            ↪ Farmacia", image="farmacia.png", x=50, y=50,
            ↪
            ↪ slot4hMinWeek=5, location="Bergamo", description="
            ↪ Text")
        url = reverse('shop:pharmacies_detail', args=(farmacia.id
            ↪ ,))
        resp = self.client.get(url)
        self.assertEqual(resp.status_code, 200)
```

Un altro esempio riguardante l'aggiunta di un prodotto al carrello:

```
class ViewTest(TestCase):
    def test_add_cart(self):
        user = User.objects.create(username='TestUser')
        farmacia = Pharmacy.objects.create(owner=user, name="
            ↪ Farmacia", image="farmacia.png", x=50, y=50,
            ↪ slot4hMinWeek=5, location="Bergamo", description="
            ↪ Text")
        categoria = Category.objects.create(name="
            ↪ Antinfiammatorio", description="Text", slug=slugify
            ↪ ("Antinfiammatorio").__str__())
        prodotto = Product.objects.create(name="Tachipirina",
            ↪ category=categoria, pharmacy=farmacia, image="image.
```

```

    ↪ png",
    ↪ description="Text", brand="Brand", quantity=10,
    ↪ price=2, shipping_fee=1, slug=slugify(1).__str__())
url = reverse('shop:add_cart', args=(prodotto.slug,))
resp = self.client.get(url)
self.assertEqual(resp.status_code, 302) # Redirect

```

In totale sono stati creati 26 casi di test per le varie funzionalità offerte dall'applicazione a livello di views, in aggiunta a 3 test speciali sul pagamento e l'upload dell'immagine dei prodotti usando i mock (vedi sezione *Mock*).

Nella figura 10.1 è riportato l'output dei test eseguiti con *unittest*; sono anche compresi i test effettuati sui due algoritmi (3 + 8 casi di test) e sui permessi (10 casi di test usando mock).

```
test_homepage (authentication.tests.ViewTest) ... ok
test_login (authentication.tests.ViewTest) ... ok
test_logout (authentication.tests.ViewTest) ... ok
test_signup (authentication.tests.ViewTest) ... ok
test_signup (authentication.tests.ViewTest) ... ok
test_invalid_form (shop.tests.BuyerDeliveryFormTest) ... ok
test_valid_form (shop.tests.BuyerDeliveryFormTest) ... ok
test_form (shop.tests.BuyerDeliveryFormTest_0_Test) ... ok
test_form (shop.tests.BuyerDeliveryFormTest_1_Test) ... ok
test_form (shop.tests.BuyerDeliveryFormTest_2_Test) ... ok
test_buyer_creation (shop.tests.BuyerModelTest) ... ok
test_buyer_str (shop.tests.BuyerModelTest) ... ok
test_category_creation (shop.tests.CategoryModelTest) ... ok
test_category_str (shop.tests.CategoryModelTest) ... ok
test_invalid_form (shop.tests.ContactFormTest) ... ok
test_valid_form (shop.tests.ContactFormTest) ... ok
test_form (shop.tests.ContactFormTestParameterized_0_Test) ... ok
test_form (shop.tests.ContactFormTestParameterized_1_Test) ... ok
test_form (shop.tests.ContactFormTestParameterized_2_Test) ... ok
test_form (shop.tests.ContactFormTestParameterized_3) ... ok
test_form (shop.tests.ContactFormTestParameterized_4_LongNameLongNameLong
test_category_creation (shop.tests.CategoryModelTest) ... ok
test_contact_str (shop.tests.ContactModelTest) ... ok
test_pharmacy_creation (shop.tests.PharmacyModelTest) ... ok
test_pharmacy_str (shop.tests.PharmacyModelTest) ... ok
test_pharmacy_creation_image (shop.tests.PharmacyModelTestMockFile) ...
test_product_creation (shop.tests.ProductModelTest) ... ok
test_product_str (shop.tests.ProductModelTest) ... Failed to populate sl

test_invalid_form (shop.tests.ReviewFormTest) ... ok
test_valid_form (shop.tests.ReviewFormTest) ... ok
test_form (shop.tests.ReviewFormTest_0_Test) ... ok
test_form (shop.tests.ReviewFormTest_1) ... ok
test_form (shop.tests.ReviewFormTest_2_LongReviewLongReviewLongReviewLong
test_form (shop.tests.ReviewFormTest_3_LongReviewLongReviewLongReviewLong
test_form (shop.tests.ReviewFormTest_4_LongReviewLongReviewLongReviewLong
test_form (shop.tests.ReviewFormTest_5_LongReviewLongReviewLongReviewLong
test_form (shop.tests.ReviewFormTest_6_LongReviewLongReviewLongReviewLong
test_review_creation (shop.tests.ReviewModelTest) ... ok
test_review_str (shop.tests.ReviewModelTest) ... ok
test_about (shop.tests.ViewTest) ... ok
test_add_cart (shop.tests.ViewTest) ... ok
test_add_review (shop.tests.ViewTest) ... ok
test_buy_items (shop.tests.ViewTest) ... ok

test_cart (shop.tests.ViewTest) ... ok
test_categories (shop.tests.ViewTest) ... ok
test_checkout (shop.tests.ViewTest) ... ok
test_contact (shop.tests.ViewTest) ... ok
test_homepage (shop.tests.ViewTest) ... ok
test_payment (shop.tests.ViewTest) ... ok
test_products_detail (shop.tests.ViewTest) ... ok
test_pharmacy_list (shop.tests.ViewTest) ... ok
test_products_detail (shop.tests.ViewTest) ... ok
test_products_list (shop.tests.ViewTest) ... ok
test_reset_cart (shop.tests.ViewTest) ... ok
test_search (shop.tests.ViewTest) ... ok
test_sell_product (shop.tests.ViewTest) ... ok
test_algorithm_calculate1 (timetable.tests.AlgorithmCalculate timetable1) ... ok
test_algorithm_calculate2 (timetable.tests.AlgorithmCalculate timetable2) ... ok
test_algorithm_calculate3 (timetable.tests.AlgorithmCalculate timetable3) ... ok
test_calculate (timetable.tests.ViewTest) ... ok
test_homepage (timetable.tests.ViewTest) ... ok
test_view (timetable.tests.ViewTest) ... ok
test_algorithm_transfer1 (transfer.tests.TransferTest1) ... ok
test_algorithm_transfer2 (transfer.tests.TransferTest2) ... ok
test_algorithm_transfer3 (transfer.tests.TransferTest3) ... ok
test_algorithm_transfer4 (transfer.tests.TransferTest4) ... ok
test_algorithm_transfer5 (transfer.tests.TransferTest5) ... ok
test_algorithm_transfer6 (transfer.tests.TransferTest6) ... ok
test_algorithm_transfer7 (transfer.tests.TransferTest7) ... ok
test_algorithm_transfer8 (transfer.tests.TransferTest8) ... ok
test_permissions_IsAdminReadOnly_false (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsAdminReadOnly_true (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsAdmin_false (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsAdmin_true (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsAuthenticatedReadOnly_false (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsAuthenticatedReadOnly_true (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsStaffReadOnly_false (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsStaffReadOnly_true (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsStaff_false (pharmacies.tests.TestPermissions) ... ok
test_permissions_IsStaff_true (pharmacies.tests.TestPermissions) ... ok
test_process_cc_with_credit (shop.tests.PaymentTestCase) ... ok
test_process_cc_without_credit (shop.tests.PaymentTestCase) ... ok

-----
Ran 77 tests in 16.679s
```

Figura 10.1.: Tests

Molti di questi test hanno permesso di individuare falle sia per quanto riguarda la compilazione di form, sia per eventuali funzioni che non facevano correttamente il loro dovere.

10.2. Test parametrici

Nei test parametrici è possibile passare dei parametri come dati di ingresso al test; in questo progetto è stato usato il package *parameterized* per *unittest*. Di seguito sono mostrati degli esempi di test parametrici inerenti alla compilazione dei form in cui vengono passati i valori dei vari campi da compilare e il risultato atteso (booleano) indicante se il form sia valido o meno. In particolare sul *ContactForm* sono stati

effettuati controlli sulla digitazione corretta dell'indirizzo email passandogli stringhe senza la chiocciola o senza il punto del dominio.

```
@parameterized_class(('name', 'email', 'subject', 'message',
    ↪ 'expected_result'), [
    ("Test", "test@email.com", "Text", "Text", True),
    ("Test", "without.at", "Text", "Text", False),
    ("Test", "without@domain", "Text", "Text", False),
    ("", "test@email.com", "Text", "Text", False),
])
class ContactFormTestParametrized(TestCase):
    def test_form(self):
        data = {'name': self.name, 'email': self.email, 'subject'
            ↪ ': self.subject, 'message': self.message}
        form = ContactForm(data=data)
        self.assertEqual(form.is_valid(), self.expected_result)
```

Un altro esempio riguardante il *BuyerForm*:

```
@parameterized_class(('full_name', 'phone', 'city', 'address'
    ↪ ', 'expected_result'), [
    ("Test", 123, "Bergamo", "Text", True),
    ("Test", 123, "CityNotInChoices", "Text", False),
    ("Test", True, "Text", "Text", False),
])
class BuyerDeliveryFormTest(TestCase):
    def test_form(self):
        data = {'full_name': self.full_name, 'phone': self.phone,
            ↪ 'city': self.city, 'address': self.address}
        form = BuyerDeliveryForm(data=data)
        self.assertEqual(form.is_valid(), self.expected_result)
```

Attraverso questi test sono stati individuati delle falle anche negli altri form, tra cui:

- Venivano accettate le mail senza la chiocciola.
- Venivano accettati i numeri di telefono composti da lettere.
- Venivano accettate stringhe vuote nei vari campi.
- Venivano accettate stringhe troppo lunghe rispetto al vincolo sulla lunghezza presente nei models.
- Venivano accettate città al di fuori di una lista presente nel file *choices.py*.

Tutti questi errori sono stati corretti attraverso l'inserimento degli asserts (vedi sezione *Assertions*) oppure modificando i vincoli dei models.

TODO (Estendere)

10.3. Criteri di copertura

In questa sezione verifichiamo la copertura del codice, ossia quanta porzione del codice è coperta dai test implementati.

Il pacchetto *coverage* è molto utile per misurare la copertura del codice Python. È possibile anche visualizzare l'output in HTML o in XML con le relative percentuali di copertura del codice.

Questi sono i comandi necessari per l'utilizzo:

```
$ coverage run manage.py test -v 2
$ coverage report
$ coverage html
$ coverage xml
$ coverage erase
```

In una prima stesura di test si era raggiunto il 49% di copertura del codice. In figura 10.2 sono rappresentati le parti meno coperte. La parte di autenticazione era stata in gran parte trascurata, così come non erano state coperte tutte le funzioni presente nei vari file *views.py*.

Coverage report: 49%				
Module ↓	statements	missing	excluded	coverage
authentication\tokens.py	6	1	0	83%
authentication\views.py	76	65	0	14%
authentication\views.py	76	65	0	14%
manage.py	12	2	0	83%
pharmacies\permission.py	33	21	0	36%
shop\models.py	95	50	0	47%
shop\views.py	186	119	0	36%
timetable\views.py	218	100	0	54%
transfer\views.py	96	34	0	65%

Figura 10.2.: Coverage precedente

Usando i report si è potuto indagare i moduli scoperti e si è proceduto a implementare ulteriori casi di test fino a raggiungere il 90% di copertura del codice.

10.3 Criteri di copertura

Coverage report: 90%									
Module ↓	statements	missing	excluded	coverage					
authentication__init__.py	0	0	0	100%	timetable__init__.py	0	0	0	100%
authentication\forms.py	10	0	0	100%	timetable\admin.py	7	1	0	86%
authentication\migrations__init__.py	0	0	0	100%	timetable\choices.py	2	0	0	100%
authentication\models.py	0	0	0	100%	timetable\migrations\0001_initial.py	7	0	0	100%
authentication\serializer.py	8	0	0	100%	timetable\migrations__init__.py	0	0	0	100%
authentication\tests.py	19	0	0	100%	timetable\models.py	16	1	0	94%
authentication\tokens.py	6	1	0	83%	timetable\serializer.py	8	0	0	100%
authentication\urls.py	8	0	0	100%	timetable\tests.py	74	0	0	100%
authentication\views.py	76	41	0	46%	timetable\urls.py	7	0	0	100%
manage.py	12	2	0	83%	timetable\views.py	218	17	0	92%
pharmacies__init__.py	0	0	0	100%	transfer__init__.py	0	0	0	100%
pharmacies\permission.py	33	3	0	91%	transfer\forms.py	6	0	0	100%
pharmacies\settings.py	34	0	0	100%	transfer\migrations__init__.py	0	0	0	100%
pharmacies\tests.py	54	0	0	100%	transfer\models.py	0	0	0	100%
pharmacies\urls.py	14	0	0	100%	transfer\tests.py	207	0	0	100%
shop__init__.py	0	0	0	100%	transfer\urls.py	4	0	0	100%
shop\admin.py	10	0	0	100%	transfer\views.py	96	34	0	65%
shop\choices.py	1	0	0	100%	Total	1568	156	0	90%
shop\forms.py	18	0	0	100%					
shop\migrations\0001_initial.py	9	0	0	100%					
shop\migrations\0002_auto_20200205_1806.py	4	0	0	100%					
shop\migrations\0003_auto_20200509_0934.py	4	0	0	100%					
shop\migrations__init__.py	0	0	0	100%					
shop\models.py	95	1	0	99%					
shop\serializer.py	31	0	0	100%					
shop\tests.py	272	12	0	96%					
shop\urls.py	12	0	0	100%					
shop\views.py	186	43	0	77%					

Figura 10.3.: Coverage finale

Inoltre è stato usato *Codecov*, un sito interattivo per la gestione e la comparazione dei rapporti di copertura del codice creati e caricati automaticamente ad ogni push grazie a *GitHub Actions* (vedi sezione *Continuous Integration*). Grazie a questo strumento è possibile interagire con i report di copertura e analizzarli nel dettaglio.

/ pharmacies							
Files					Coverage		
authentication	127	85	0	42		66.93%	
pharmacies	135	132	0	3		97.78%	
shop	642	586	0	56		91.28%	
timetable	339	323	0	16		95.28%	
transfer	313	279	0	34		89.14%	
manage.py	12	10	0	2		83.33%	
Folder Totals (6 files)	1,568	1,415	0	153		90.24%	
Project Totals (34 files)	1,568	1,415	0	153		90.24%	

Figura 10.4.: Codecov

Nell'immagine 10.5 sono rappresentati i vari moduli dell'applicazione (settori del grafico) colorati usando una sfumatura che parte dal rosso (copertura insufficiente) fino al verde (copertura massima): quello a sinistra era in una fase intermedia del processo di testing, quello a destra in una fase finale.

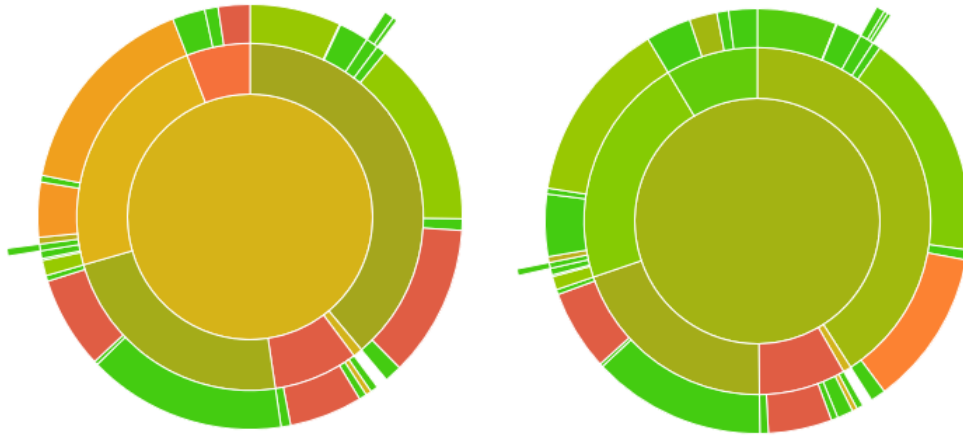


Figura 10.5.: Codecov Sunburst

TODO (Codeclimate, Codacy)

10.4. Criteri avanzati di copertura

10.4.1. Mutation Testing

Il *mutation testing* valuta la qualità dei test modificando il codice sorgente di un programma in piccole parti. Una test suit che non rileva e rifiuta il codice mutato è considerata difettosa. Queste mutazioni si basano su operatori di mutazione ben definiti che imitano i tipici errori di programmazione (come l'uso dell'operatore sbagliato o il nome della variabile) o costringono alla creazione di test significativi (come portare ogni espressione a zero). Lo scopo è aiutare lo sviluppatore a sviluppare test efficaci, a individuare punti deboli nel codice e a individuare sezioni in cui non si accede mai o solo raramente durante l'esecuzione.

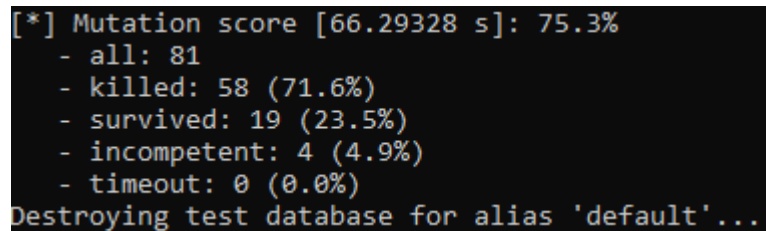
È stato usato il pacchetto *django-mutpy* ottimizzato per Django e basato su *mutpy* eseguibile usando questo comando:

```
$ python manage.py muttest shop
```

Una lista degli operatori di mutazione di *mutpy*: AOD (Arithmetic operator deletion), AOR (Arithmetic operator replacement), ASR (Assignment operator replacement), BCR (Break continue replacement), COD (Conditional operator deletion), COI (Conditional operator insertion), CRP (Constant replacement), DDL (Decorator deletion), EHD (Exception handler deletion), EXS (Exception swallowing), IHD (Hiding variable deletion), IOD (Overriding method deletion), IOP (Overridden method calling position change), LCR (Logical connector replacement), LOD (Logical operator deletion), LOR (Logical operator replacement), ROR (Relational operator

replacement), SCD (Super calling deletion), SCI (Super calling insert), SIR (Slice index remove).

Il risultato finale dei test, dopo aver fatto varie iterazioni per migliorarlo, di mutazione è il seguente:



```
[*] Mutation score [66.29328 s]: 75.3%
- all: 81
- killed: 58 (71.6%)
- survived: 19 (23.5%)
- incompetent: 4 (4.9%)
- timeout: 0 (0.0%)
Destroying test database for alias 'default'...
```

Figura 10.6.: MutPy

10.4.2. TODO

TODO (MCDC, Logici, Dataflow, Loop, Random)

10.5. Continuous Integration (CI)

Nello sviluppo di questo progetto è stata usata la continuous integration offerta da *GitHub Actions* e integrata nel repository del progetto stesso. Risulta molto comoda in quanto basta configurare al meglio il file `.github/workflows/django-testing.yml` (ad esempio per il testing) indicando i comandi da eseguire. Ad ogni push infatti vengono eseguiti numerosi test automaticamente che ci permettono di avere in tempo reale un indicatore se qualcosa è andato storto o meno dopo le ultime modifiche apportate al codice e caricate sul repository.

In particolare, la sequenza di passi che viene eseguita da `django-testing.yml` è la seguente:

- Viene configurato l'ambiente (*Ubuntu*) e installata la versione corretta di Python, in questo caso la 3.8.
- Vengono soddisfatte tutte le dipendenze installando i pacchetti indicati nel file `requirements.txt` (sfruttando anche la cache).
- Vengono fatte le migrazioni di Django.
- Vengono eseguiti tutti i test implementati con `unittest` e implicitamente verificati i contratti stabiliti con `icontract`.
- Viene effettuato un report di copertura del codice attraverso `coverage` (in formato HTML e XML) e caricato automaticamente su *CodeCov* per un'analisi dettagliata e un confronto tra i vari reports precedenti.

Inoltre nel file *django-verification.yml* vi è la parte verifica del codice (vedi sezione (illustrata successivamente *Integrazione con CI*).

10.6. Capture and Replay

Selenium è un framework portatile per testare le applicazioni web e fornisce uno strumento di riproduzione. In particolare simula l'interazione che l'utente fa con l'applicazione web attraverso, in questo caso, mouse e tastiera usando il browser *Firefox*.

Un esempio è quello della compilazione del form di contatto per la richiesta di supporto:

```
class ContactViewSeleniumTest(LiveServerTestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        last_height = self.driver.execute_script("return document
            ↪ .body.scrollHeight")

    def test_selenium_contact(self):
        self.driver.get("http://localhost:8000/shop/contact")
        self.driver.find_element_by_id('id_name').send_keys("Gino
            ↪ ")
        self.driver.find_element_by_id('id_email').send_keys("
            ↪ gino@mail.com")
        self.driver.find_element_by_id('id_subject').send_keys("Text")
        self.driver.find_element_by_id('id_message').send_keys("
            ↪ Text")
        self.driver.find_element_by_id('submit').click() #
            ↪ Submit button
        self.assertIn("http://localhost:8000/shop/contact", self.
            ↪ driver.current_url)

    def tearDown(self):
        self.driver.quit
```

Necessita che il server dell'applicazione sia in esecuzione in locale su un altro terminale e la presenza di *geckodriver.exe* (vedi *Manuale utente*).

TODO (Estendere)

10.7. Mock

È stato usato *unittest.mock*, una libreria per il testing in Python che permette di sostituire parti del sistema con oggetti mock e fare assertions sul loro funzionamento.

Il decoratore di *patch()* semplifica il testing rendendo mock classi o oggetti specificati. L'oggetto verrà sostituito con un oggetto fittizio durante il test e ripristinato al termine del test.

Nel seguente esempio viene illustrato l'utilizzo del decoratore *patch()* per sostituire la funzione *calculate_amount()* con una fittizia che restituisce un determinato valore. Nel primo caso si testa il pagamento tramite carta in cui si ha abbastanza credito per portare a termine la transazione, viceversa nel secondo caso.

```
class PaymentTestCase(unittest.TestCase):
    @mock.patch('shop.views.calculate_amount', autospec=True)
    def test_process_cc_with_credit(self, mock_calculate_amount
        ↪ ):
        cc = FakeCreditCard(50)
        mock_calculate_amount.return_value = 25
        payment = Payment(1, cc)
        status = payment.process(self)
        self.assertEqual(status, 'processed')

    @mock.patch('shop.views.calculate_amount', autospec=True)
    def test_process_cc_without_credit(self,
        ↪ mock_calculate_amount):
        cc = FakeCreditCard(50)
        mock_calculate_amount.return_value = 200
        payment = Payment(1, cc)
        status = payment.process(self)
        self.assertEqual(status, 'cancelled')
```

Usando *MagicMock*, una sottoclasse di *Mock* con tutti i metodi magici precedentemente creati e pronti all'uso, si è creato un oggetto di tipo file (*spec=File*) usato come parametro durante la creazione di un prodotto nel database. In particolare quest'oggetto fittizio va a simulare un'immagine in formato *.png* del prodotto.

```
class PharmacyModelTestMockFile(TestCase):
    def create_pharmacy_image(self, image, name="Farmacia", x
        ↪ =50, y=0, slot4hMinWeek=5, location="Bergamo",
        ↪ description="Text"):
        user = User.objects.create(username='TestUser')
        file_mock = mock.MagicMock(spec=File)
        file_mock.name = image
        return Pharmacy(owner=user, name=name, image=file_mock, x
            ↪ =x, y=y, slot4hMinWeek=slot4hMinWeek, location=
            ↪ location, description=description)

    def test_pharmacy_creation_image(self):
        w = self.create_pharmacy_image("image.png")
        self.assertTrue(isinstance(w, Pharmacy))
        fields = w.id, w.owner, w.name
```

```
self.assertEqual(w.__unicode__(), fields)
self.assertEqual(w.image.name, "image.png")
```

Inoltre *MagicMock* è stato impiegato anche nel testare i permessi dei vari utenti presenti nel *permission.py*:

```
class TestPermissions(TestCase):
    def test_permissions_IsAdmin_true(self):
        self.request = MagicMock(user=MagicMock())
        self.request.user.is_superuser = True
        self.view = MagicMock()
        self.assertTrue(IsAdmin.has_permission(self, self.request
        ↪ , self.view))

    def test_permissions_IsAdmin_false(self):
        self.request = MagicMock(user=MagicMock())
        self.request.user.is_superuser = False
        self.view = MagicMock()
        self.assertFalse(IsAdmin.has_permission(self, self.
        ↪ request, self.view))
```


11. Code Verification

TODO (Descrizione)

11.1. Assertions

Gli *asserts* devono essere usati per testare condizioni che non dovrebbero mai accadere con lo scopo di inviare un segnale di errore e arrestarsi in anticipo sollevando un'eccezione nel caso di funzionamento non corretto. Quini gli asserts servono per trovare bug facilmente e velocemente.

Come si possono usare gli asserts:

- Controllo di tipi di parametri, classi o valori.
- Verifica degli invarianti delle strutture dei dati.
- Verifica di situazioni «impossibili» (duplicati in un elenco, variabili di stato contraddittorie).
- Dopo aver chiamato una funzione, per assicurarsi che il valore di ritorno sia ragionevole.

Nota: Non usare gli asserts per la convalida dei dati! Gli asserts possono essere disattivati globalmente nell'interprete Python, quindi non bisogna fare affidamento sugli asserts per la validazione dei dati.

Un esempio di utilizzo degli asserts riguardante la classe *Payment*:

```
class Payment():
    def __init__(self, invoice_id, credit_card):
        assert isinstance(credit_card, FakeCreditCard), "
            ↪ credit_card is not a FakeCreditCard instance"
        self.credit_card = credit_card

    def process(self, request):
        amount = calculate_amount()
        assert amount >= 0, "amount should be positive"
        if self.credit_card.has_enough_credit(amount):
            self.credit_card.withdraw(amount)
            self.status = 'processed'
        else:
```

```
self.status = 'cancelled'
return self.status
```

Si può notare che è stato usato un assert per verificare che la variabile *credit_card* passata come argomento alla funzione sia effettivamente un'istanza della classe *FakeCreditCard*. Poco più sotto si verifica che il quantitativo richiesto per il pagamento sia un valore positivo. In entrambi i casi, se l'assert non è verificato viene sollevata un'eccezione.

Inoltre anche all'interno dei test abbiamo usato gli assert per verificare che il risultato di un test sia identico a un valore atteso. Un esempio particolare è nell'ottavo test dell'algoritmo *transfer* in cui si verifica che venga lanciata un'eccezione causata da un numero di prodotti insufficiente per soddisfare la richiesta di trasferimento:

```
# views.py
if (len(listaProducts) == 0):
    raise Exception("Non ci sono abbastanza prodotti")
    break;

# tests.py
self.assertRaises(Exception)
```

TODO (Estendere)

11.2. Design by Contract

Per quanto concerne il Design by Contract in Python vi è una specifica «*PEP 316 - Programming by Contract for Python*» ma con status *deferred*; bisogna quindi fare affidamento su altri pacchetti come ad esempio *icontract*.

Questo pacchetto fornisce due function decorators, *require* e *ensure* che richiedono e garantiscono rispettivamente le precondizioni e le postcondizioni. Inoltre, fornisce anche un class decorator, *invariant*, per stabilire i class invariant.

Le precondizioni sono scritte con il decoratore *require* e in lambda vi sono i parametri passati alla funzione. Le postcondizioni sono scritte con il decoratore *ensure* e in lambda vi è perlomeno un parametro *result*, ossia il valore ritornato dalla funzione.

Nota: Gli invarianti si possono usare solo con le classi, quindi gran parte delle funzioni implementate non poteva avere gli invarianti; ogni invariante richiede *self* come unico parametro.

Di seguito un esempio di contratto della funzione *algorithm_transfer()*:

```
@icontract.require(lambda quantity: quantity > 0, "quantity
    ↪ must be positive")
@icontract.require(lambda x: x >= 0 and x <= 100, "coordinate
    ↪ 0 <= x <= 100")
```

```

@icontract.require(lambda y: y >= 0 and y <= 100, "coordinate
    ↪ 0 <= y <= 100")
@icontract.require(lambda category: Product.objects.all().
    ↪ filter(category=category).count() >= 1, "at least one
    ↪ product of that category is required")
@icontract.ensure(lambda result: len(result[0]) == len(result
    ↪ [1]))
@icontract.ensure(lambda result, quantity: sum(result[1]) ==
    ↪ quantity)
def algorithm_transfer(request, category: Category, quantity:
    ↪ int, x: int, y: int) -> list:
    # Codice

```

In questo esempio vengono verificati che i parametri passati siano corretti, ossia che *quantity* sia strettamente positiva e che le coordinate *x* e *y* siano comprese tra 0 e 100; inoltre viene anche verificato che esista un prodotto di quella categoria specifica all'interno del database. Alla fine dell'algoritmo si verifica che il risultato, composto da una lista di liste (la prima lista interna di ID delle farmacie, la seconda lista interna di quantità prelevata dalla determinata farmacia in quella posizione), abbia le due liste interne di uguale lunghezza e che la somma delle quantità prelevate dalle farmacie sia pari alla quantità richiesta per il trasferimento.

In modo simile è stato fatto un contratto alla funzione *findGreedy()*:

```

@icontract.require(lambda listaProducts: len(listaProducts) >
    ↪ 0, "listaProducts must not be empty")
@icontract.require(lambda x: x >= 0 and x <= 100, "coordinate
    ↪ 0 <= x <= 100")
@icontract.require(lambda y: y >= 0 and y <= 100, "coordinate
    ↪ 0 <= y <= 100")
@icontract.ensure(lambda result: Pharmacy.objects.filter(id=
    ↪ result[0]).count() >= 1)
@icontract.ensure(lambda result: result[1] >= 0)
@icontract.ensure(lambda result: 0 <= result[2] <= 100)
@icontract.ensure(lambda result: 0 <= result[3] <= 100)
def findGreedy(listaProducts: list, x: int, y: int):
    # Codice

```

Il controllo di un invariante è stato fatto ad esempio nella classe *FakeCreditCard* in cui il bilancio della carta non può mai essere negativo:

```

@icontract.invariant(lambda self: self.balance >= 0, "balance
    ↪ must not be negative")
class FakeCreditCard:
    def __init__(self, balance=50):
        assert balance >= 0, "balance should not be negative"
        self.balance = balance

```

```
def has_enough_credit(self, amount):
    return self.balance > amount

def withdraw(self, amount):
    self.balance = self.balance - amount
    assert self.balance >= 0, "balance should not be negative
    ↪ "
```

TODO (Estendere)

11.3. Program Verification

La *program verification* mira a utilizzare prove formali per dimostrare che i programmi si comportano secondo le specifiche.

TODO

11.4. Analisi statica

Nella programmazione *lint* o *lint-like tools* indicano strumenti che eseguono un'analisi statica del codice.

11.4.1. Pylint

Il tool multifunzionale *pylint* è stato usato per l'analisi statica e comprende:

- Coding standards: check sulla lunghezza delle linee di codice, check sui nomi delle variabili (notazioni), check sui moduli importati e usati.
- Error detection: check sulle interfacce e la loro implementazione.
- Refactoring: detection di codice duplicato.
- UML diagrams: *pylint* integra *pyreverse* che permette l'esportazione automatica di diagrammi UML riguardanti le classi, i package e la struttura del database in formato *.dot* o *.png*.

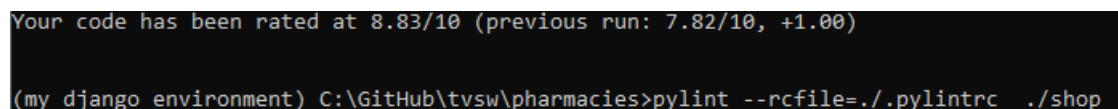
Grazie ai seguenti comandi è possibile installare pylint, successivamente fare una scansione automatica dell'app shop scegliendo se farla completa, solo sugli errori evitando i warning oppure fare anche un test sulla performance del database.

```
$ pylint --rcfile=./.pylintrc ./shop
$ pylint --rcfile=./.pylintrc --errors-only ./shop
$ pylint --rcfile=./.pylintrc --load-plugins pylint_django --
  ↪ load-plugins pylint_django.checkers.db_performance ./
  ↪ shop
```

Grazie al file `.pylintrc` è possibile personalizzare interamente i parametri della ricerca andando a settare le varie impostazioni al proprio interno; la configurazione si può trovare nel file `.pylintrc` presente nella cartella base. Gli errori più comuni sono:

- C0103 Doesn't conform to snake_case naming style
- C0114 Missing-module-docstring
- C0115 Missing class docstring
- C0116 Missing function or method docstring
- C0301 Line too long
- C0330 Wrong hanging indentation before block
- E1101 Class 'X' has no 'objects' member

Nella figura 11.1 è riportato l'output di `pylint` dove viene valutato il codice su una scala da 0 a 10.



```
Your code has been rated at 8.83/10 (previous run: 7.82/10, +1.00)

(my django environment) C:\GitHub\tvsw\pharmacies>pylint --rcfile=./pylintrc ./shop
```

Figura 11.1.: Pylint

TODO (Risolvere)

11.4.2. Flake8

Il pacchetto *flake8* è un altro strumento per l'analisi del codice.

Al suo interno integra i seguenti tools:

- *pyflakes*
- *pycodestyle*
- *Ned Batchelder's McCabe script*

Si basa anche sulla specifica «*PEP 8 -- Style Guide for Python Code*» per il controllo dello stile del codice Python.

Dopo averlo installato, si può eseguire il seguente comando per una rapida analisi del codice:

```
$ flake8 -v --count
```

Verrà restituito in output un elenco di violazioni ognuna con uno specifico codice identificativo del tipo di violazione; violazioni comuni sono:

- E501 Line too long (90 > 79 characters)

- F405 'User' may be undefined, or defined from star imports

È possibile nascondere questi errori comuni per far risaltare altri errori magari più gravi usando questo comando:

```
$ flake8 -v --count --ignore=E501,F405
```

Dopo varie iterazioni, sfruttando tool come *black* per risolvere l'errore E501 ad esempio, si è potuto ridurre di molto le violazioni.

11.4.3. Bandit

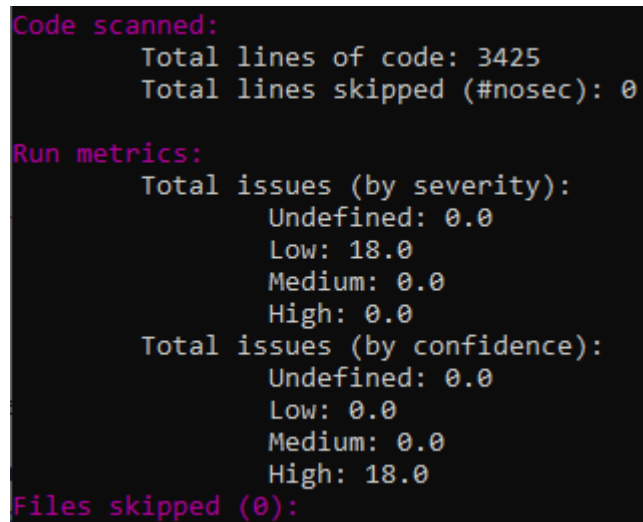
Bandit è un framework per eseguire analisi di sicurezza del codice sorgente Python, utilizzando il modulo *ast* della Python Standard Library.

Il modulo *ast* viene utilizzato per convertire il codice sorgente in un albero analizzato di nodi di sintassi Python. Bandit consente agli utenti di definire test personalizzati che vengono eseguiti su tali nodi. Al termine del test, viene generato un rapporto che elenca i problemi di sicurezza identificati nel codice sorgente di destinazione.

Dopo averlo installato, si può eseguire il seguente comando:

```
$ bandit -r -v .
```

Verranno elencate le vulnerabilità trovate in un report con sommario finale come in figura 11.2.



```
Code scanned:
  Total lines of code: 3425
  Total lines skipped (#nosec): 0

Run metrics:
  Total issues (by severity):
    Undefined: 0.0
    Low: 18.0
    Medium: 0.0
    High: 0.0
  Total issues (by confidence):
    Undefined: 0.0
    Low: 0.0
    Medium: 0.0
    High: 18.0

Files skipped (0):
```

Figura 11.2.: Bandit

Dopo aver eliminato e risolto vulnerabilità anche gravi trovate nel codice, le uniche rimaste sono:

- «*[B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes*» che tuttavia si può ignorare perchè i numeri random generati nel codice non riguardano algoritmi crittografici o relativi alla sicurezza del sistema.
- «*[B101:assert_used] Use of assert detected. The enclosed code will be removed when compiling to optimised byte code*» che tuttavia si può ignorare perchè sono stati effettuati ulteriori controlli oltre agli assert.

11.4.4. Pyreverse e GraphViz

I tool *pyreverse* e *GraphViz* sono usati per esportare diagrammi UML riguardanti le class e i packages di ogni modulo. Inoltre è possibile esportare pure lo schema entità relazione del database.

Dopo aver installato *pyreverse* e *GraphViz* (necessita anche di aggiungere *bin/gvedit.exe* nel path delle variabili d'ambiente) si possono usare con i seguenti comandi:

```
$ pyreverse -o png -A -s 0 -a 0 -k authentication shop
  ↪ timetable transfer --ignore=migrations,tests,tests.py
$ pyreverse -o png -A -s 0 -a 0 -k shop --ignore=migrations,
  ↪ tests,tests.py

$ python manage.py graph_models -a -o models.png
$ python manage.py graph_models authentication shop timetable
  ↪ transfer -o apps.png
```

Nella figura 11.3 sono rappresentate le classi dell'intera applicazione:

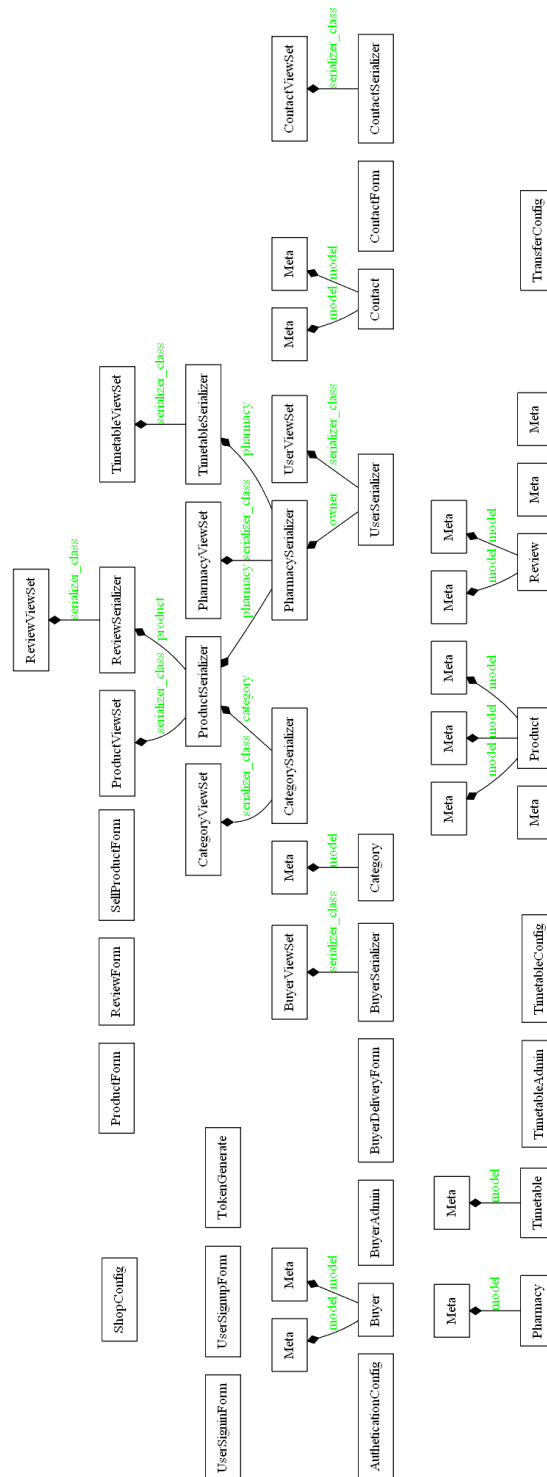


Figura 11.3.: Classes

Nella figura 11.4 sono rappresentate le interazioni tra i vari package dell'applicazione.

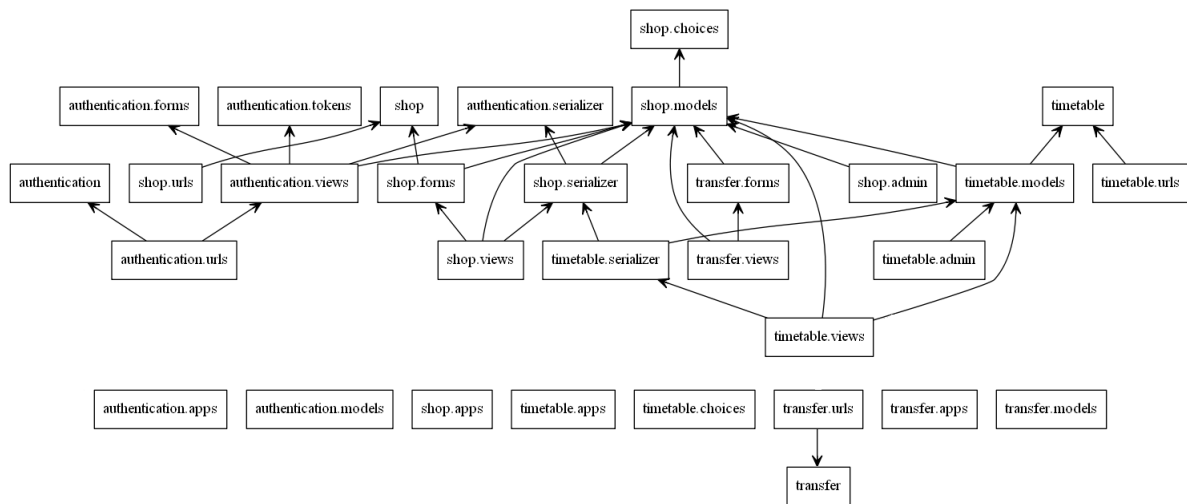


Figura 11.4.: Packages

11.5. Code Refactoring

Il *code refactoring* è il processo di modifica di un sistema software in modo tale da migliorare la sua struttura interna del codice senza alterarne il comportamento esterno.

È una fase molto importante dello sviluppo del software, rende il codice più semplice da leggere e più facile da estendere. Molto spesso ci si accorge che man mano che il progetto diventa grande, sia necessaria una nuova struttura del codice e dei vari componenti.

App:

In questo caso inizialmente l'intero progetto era contenuto in un'unica app, tuttavia la necessità di semplificare il codice e raggruppare le funzioni ha portato alla divisione del progetto in quattro distinte componenti:

- *authentication*: modulo che si occupa di tutta la parte di autenticazione e registrazione degli utenti e dei permessi relativi.
- *shop*: modulo principale che si occupa di gestire tutta la parte di presentazione, acquisto e vendita dei farmaci e della gestione delle farmacie.
- *timetable*: modulo che si occupa della presentazione e del calcolo degli orari settimanali delle farmacie.
- *transfer*: modulo che si occupa del trasferimento dei farmaci tra la farmacia.

Dopo aver deciso la nuova struttura del progetto raggruppando in base al contesto, bisogna spostare tutti i modelli attinenti alla componente più appropriata. In *authentication* sono presenti i modelli relativi agli utenti, in *timetable* quelli relativi

agli orari delle farmacie, in *transfer* quelli relativi alle richieste dei medicinali e in *shop* la restante grande parte che si occupa di gestire le funzionalità del sito.

Grazie a questa divisione si è potuto rendere il codice più leggibile, rendere chiaro i collegamenti e le interdipendenze tra i vari moduli e si è potuto sviluppare ogni algoritmo (*timetable*, *transfer*) in modo isolato. Inoltre è stato possibile accorpare ed eliminare funzionalità duplicate o modelli inutilizzati.

I numerosi test presenti nel progetto sono anch'essi divisi in base al modulo di appartenenza e possono essere eseguiti in modo indipendente attraverso appositi comandi.

Naturalmente spostare tutti i modelli e fare il refactoring del codice non è stata una cosa semplice. I principali passaggi effettuati possono concentrarsi nei seguenti punti:

- Rinominare tutte le tabelle durante lo spostamento per adattarsi alla nuova struttura di destinazione ed evitare conflitti.
- Fare una migrazione per spostare i modelli nell'app di destinazione.
- Aggiornare eventuali foreign keys and relations tra i vari modelli (anche di app differenti).
- Fare una migrazione per eliminare i modelli dall'applicazione di origine.

API:

Allo stesso modo le varie Application Programming Interface sono state separate nei vari moduli.

Vedi REST API.

Algoritmi:

Isolati gli algoritmi, la loro stesura è stata migliorata e divisa in più funzioni ognuna con uno scopo preciso. Si è usato anche un data flow chart per ristrutturare il codice in modo da rendere chiara la divisione in varie casistiche.

Vedi fasi precedenti riguardanti gli algoritmi.

Codice:

È stato usato *black*, un formattatore di codice per Python in cui ogni riga del codice viene formattata automaticamente liberando lo sviluppatore di questo compito.

Può essere usato con il seguente comando, in cui l'opzione *-check* mostra in output i file che verrebbero riformattati in *dry run*:

```
$ black . --line-length 79 --check
$ black . --line-length 79
```

Documentazione:

Nel pannello admin, all'indirizzo *admin/doc/* è possibile visualizzare la documentazione presa dalle docstrings dei models, views, template tags e template filters di ogni app installata nel progetto.

Vedi REST API.

Performance:

L'analisi della performance degli algoritmi si può trarre dall'analisi di complessità degli algoritmi descritta nelle fasi precedenti.

Vedi analisi di complessità degli algoritmi.

Variabili:

In generale nell'applicazione sono state rinominate molte variabili per rendere più esplicito e facilmente intuibile il loro scopo usando anche lo stile *snake_case*; anche gli import dei pacchetti necessari sono stati riorganizzati.

TODO (Vario)

11.6. Integrazione con CI

In *GitHub Actions* sono state integrate gran parte delle verifiche sul codice nel file *django-verification.yml*. I passi sono i seguenti:

- Viene configurato l'ambiente (*Ubuntu*) e installata la versione corretta di Python, in questo caso la 3.8.
- Vengono soddisfatte tutte le dipendenze installando i pacchetti indicati nel file *requirements.txt* (sfruttando anche la cache).
- Viene effettuata l'analisi del codice con *pylint*.
- Viene eseguito black per formattare il codice Python.

Nota: I contratti sono già stati verificati quando vengono eseguiti i casi di test nella fase di testing.

11.7. Code Inspection

La *checklist* è un elemento fondamentale dell'ispezione classica. Riassumono l'esperienza accumulata nei progetti precedenti per guidare le sessioni di revisione. Una checklist contiene una serie di domande che aiutano a identificare i difetti del modulo ispezionato. Una buona checklist dovrebbe essere aggiornata regolarmente per rimuovere le parti obsolete e per aggiungere nuovi controlli suggeriti dall'esperienza accumulata. Tipici controlli riguardano:

- Design and Architecture errors

- Computation errors
- Comparison errors
- Control flow errors
- Subroutine parameter errors
- Input/Output errors
- Memory allocation errors
- Error discovered from previous code reviews
- Other check (Pass the lint test?...)

Uso la seguente checklist sulla mia app shop:

All Tests:

- The CI jobs on the pull request have passed. *Yes, I controlled the last push.*
- It is obvious what the test is trying to test. *Yes, it is documented.*
- The test passes when it's supposed to pass. *Yes, positive case passed.*
- The test fails when it's supposed to fail. *Yes, negative case passed.*
- The test is testing what it thinks it's testing. *Yes.*
- The spec backs up the expected behavior in the test. *Yes, conforms to specifications.*
- The test is automated as either reftest or a script test unless there's a very good reason for it not to be. *Yes, the test is automated and started with a command line.*
- The test does not use external resources. *No, it use only packages imported.*
- The test does not use proprietary features (vendor-prefixed or otherwise). *No, it doesn't use proprietary features.*
- The test does not contain commented-out code. *Yes, only a Selenium test is commented in order to avoid long execution times.*
- The test is placed in the relevant directory. *Yes, every test related to a specific app is located inside tests.py in the app's directory.*
- The test has a reasonable and concise filename. *No, it's just tests.py.*
- If the test needs code running on the server side, the server code must be written in Python, and the Python code must not do anything potentially unsafe. *Yes, only in localhost for now.*
- If the test needs to be run in some non-standard configuration or needs user interaction, it is a manual test. *No, it is all automated.*
- Nit: The title is descriptive but not too wordy. *Yes, the title has coincided but clear.*

Reftests Only: *(Not used)*

- The reference file is accurate and will render pixel-perfect identically to the test on all platforms.
- The reference file uses a different technique that won't fail in the same way as the test.
- The test and reference render within a 800x600 viewport, only displaying scrollbars if their presence is being tested.
- Nit: The test has a self-describing statement.
- Nit: The self-describing statement is accurate, precise, simple, and self-explanatory. Someone with no technical knowledge should be able to say whether the test passed or failed within a few seconds, and not need to spend several minutes thinking or asking questions.

Script Tests Only:

- The number of tests in each file and the test names are consistent across runs and browsers. It is best to avoid the pattern where there is a test that asserts that the feature is supported and bails out without running the rest of the tests in the file if it isn't. *Yes, the number of tests in each file and the test names are consistent.*
- The test avoids patterns that make it less likely to be stable. In particular, tests should avoid setting internal timeouts, since the time taken to run it may vary on different devices; events should be used instead (if at all possible). *Yes, test avoids patterns that make it less likely to be stable.*
- The test uses the most specific asserts possible (e.g. doesn't use `assert_true` for everything). *Yes, it use the proper assert, for example `assertEqual`, `assertTrue`, `assertException`.*
- Nit: Tests in a single file are separated by one empty line. *Yes, everything is in order.*

Visual Tests Only: *(Not used)*

- The test has a self-describing statement.
- The self-describing statement is accurate, precise, simple, and self-explanatory. Someone with no technical knowledge should be able to say whether the test passed or failed within a few seconds, and not need to spend several minutes thinking or asking questions.
- The test renders within a 800x600 viewport, only displaying scrollbars if their presence is being tested.
- The test renders to a fixed, static page with no animation.

TODO (Aggiungere)

12. Model Verification

TODO

13. Model Based Testing

TODO

Parte VI.

Manuale utente

Installazione

Le varie versioni dei pacchetti da installare sono definite nel file *requirements.txt*. La versione di *Django* usata è la 3.0.1.

Assicurarsi di aver installato *Python* (versione 3.8) nel proprio sistema e di aver aggiornato *pip*. In seguito creare l'ambiente virtuale e installare tutti i pacchetti necessari contenuti nel file *requirements.txt* tramite gli appositi comandi riportati di seguito. Successivamente creare il superuser ed effettuare tutte le migrazioni di *Django*. Usare il comando *runserver* per avviare il server in locale all'indirizzo *http://127.0.0.1:8000/*.

Ulteriori comandi si possono trovare nel file *Commands.md* presente nella cartella del progetto.

Usare il terminale ed eseguire i seguenti comandi per le operazioni sopra indicate:

```
// Code path
$ cd /indirizzo/pharmacies

// pip
$ python -m pip install --upgrade pip

// Virtual Environment
$ pip3 install virtualenvwrapper-win
$ mkvirtualenv my_django_environment
$ workon my_django_environment

// Requirements
$ pip install -r requirements.txt

// Django
$ python manage.py createsuperuser
$ python manage.py migrate
$ python manage.py migrate --run-syncdb
$ python manage.py makemigrations
$ python manage.py runserver
```

Se tutto va a buon fine dovreste ottenere l'output in figura 13.1.

```
C:\GitHub\I3B\pharmacies>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
February 05, 2020 - 11:22:41
Django version 3.0.1, using settings 'pharmacies.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Figura 13.1.: Server online

Configurazione

Accedendo all'indirizzo indicato tramite un browser si otterrà la homepage dell'applicazione web. Bisognerà accedere alla sezione admin al seguente indirizzo <http://127.0.0.1:8000/admin> con le credenziali del superuser creato in precedenza: all'interno è possibile creare nuovi utenti o modificarne esistenti in modo tale che essi risultino clienti semplici oppure farmacisti (semplicemente mettendo la spunta su «*is staff*»).

Per popolare il database si può importare il file *db.json* che sovrascriverà l'attuale database: prima di tutto cancellare il file *db.sqlite3* per evitare conflitti; a terminale eseguire *migrate* e creare il proprio superuser admin e rieseguire *migrate*; successivamente importare il *db.json* con il comandi sottostanti ed avviare il server nuovamente.

```
// Data
$ python manage.py loaddata db.json
```

Per visualizzare la documentazione delle API installare *Swagger UI* con il seguente comando:

```
// Swagger UI
$ pip install django-rest-swagger
```

Testing e verifica del software

Per usare i vari tools aggiuntivi necessari per il testing seguire i seguenti comandi:

```
// --[Code Testing]-- //

// unittest
// Integrato

// parameterized
$ pip install parameterized
```

```
// coverage
$ pip install coverage

// Codecov
// Sito

// django-mutpy
$ pip install django-mutpy

// Selenium (Firefox necessario e geckodriver.exe)
$ pip install selenium
$ pip install lxml
$ pip install defusedxml

// mock
$ pip install mock

// --[Code Verification]-- //

// icontract
$ pip install icontract

// pylint
$ pip install pylint
$ pip install pylint-django

// flake8
$ pip install flake8

// bandit
$ pip install bandit

// pyreverse
$ pip install pyreverse

// GraphViz
// Istruzioni per l'installazione sul sito

// black
$ pip install black
```

Per usare *Selenium* è necessario avere installato *Firefox* ed è necessaria la presenza dell'eseguibile *geckodriver.exe*.

Bibliografia

- [1] C. Demetrescu, I. Finocchi, G. F. Italiano: *Algoritmi e strutture dati*, McGraw-Hill
- [2] Kenneth A. Lambert: *Programmazione in Python*, Apogeo
- [3] Django Documentation (<https://docs.djangoproject.com/en/3.0/>)
- [4] Django REST Framework Tutorial (<https://www.django-rest-framework.org/tutorial/quickstart/>)
- [5] StackOverflow (<https://stackoverflow.com/>)