

Documentazione di progetto

Informatica III - Modulo di Progettazione e Algoritmi

Michele Beretta - 1054365

Bianca Crippa - 1053356

Pape Alpha Toure - 1053327

A.A. 2020/2021

Indice

1 Iterazione 0	1
1.1 Introduzione	1
1.2 Analisi dei requisiti	2
1.2.1 Analisi del contesto	2
1.2.2 Studio di fattibilità	2
1.3 Specifiche	2
1.3.1 Casi d'uso	2
1.3.2 Requisiti funzionali	11
1.4 Architettura software	13
1.4.1 Component Diagram	13
1.4.2 Class Diagram	13
1.5 Architettura hardware	14
1.5.1 Architettura hardware	15
1.5.2 Deployment Diagram	15
2 Iterazione 1	17
2.1 Introduzione	17

2.2	Back-end e REST API	18
2.2.1	Autenticazione	18
2.2.2	App AI	19
2.2.3	Console di amministrazione	21
2.2.4	API endpoints	22
2.3	Front-end	23
2.3.1	Struttura SPA	23
2.3.2	Pagine visitabili	24
2.4	Analisi statica	25
2.4.1	Back-end	25
2.4.2	Front-end	26
2.5	Analisi dinamica	26
2.5.1	Testing	26
2.5.2	Coverage	31
3	Iterazione 2	34
3.1	Introduzione	34
3.2	Back-end	35
3.2.1	Rimozione di Django REST Framework	35
3.2.2	API endpoints	35
3.2.3	Testing ed analisi statica	37
3.3	Front-end	37
3.3.1	Pagine aggiunte	37

3.3.2	Testing, coverage ed analisi statica	41
4 Iterazione 3		43
4.1	Descrizione dell'algoritmo	43
4.2	Flowchart	45
4.3	Pseudocodice	48
4.4	Analisi di complessità	51
4.4.1	Classificazione ticket	51
4.4.2	Stima ARIMA del tempo di esecuzione	53
5 Iterazione 4		55
5.1	Continuous Integration	55
5.2	Electron	55
5.2.1	Integrazione con il front-end	56
5.2.2	Gestione dei cookies	57
6 Toolchain		58
6.1	Modellazione	58
6.2	Linguaggi e librerie	59
6.2.1	Back-end	59
6.2.2	Front-end	59
6.3	Infrastruttura	61
6.4	Documentazione	61
6.5	Versioning	61
6.6	Ambienti e IDEs	61

Capitolo 1

Iterazione 0

1.1 Introduzione

In ambito aziendale è solito avere un sistema cosiddetto di *ticketing* per il tracciamento e la risoluzione di problemi, principalmente relativi ai prodotti software venduti, tra i dipendenti dell'azienda stessa e gli utenti utilizzatori. Questa gestione consente di tenere traccia dell'evoluzione della comunicazione e dei vari passi risolutivi che portano alla chiusura del ticket ed alla sistemazione di eventuali bug e regressioni.

Il sistema di *ticketing* da noi realizzato implementa tutte le funzionalità comunemente presenti negli altri sistemi, ma si differenzia per via dell'assegnamento automatico dei ticket aperti a varie unità organizzative per un miglior filtraggio ed una migliore organizzazione di informazioni. Questo assegnamento automatico è effettuato tramite tecniche di classificazione non supervisionate, come ad esempio *k-means*, che analizzano solamente i dati che definiscono un ticket.

1.2 Analisi dei requisiti

1.2.1 Analisi del contesto

La gestione dei ticket, seppur necessaria e fondamentale per un'azienda, può rappresentare un costo in termini sia di risorse sia di tempo, in quanto il ticket dev'essere letto, compreso e smistato per poterlo indirizzare alla persona (o gruppo di persone) più in grado di risolvere il problema.

Il processo di evasione dei ticket rappresenta dunque un costo ulteriore che può diminuire l'efficienza dei processi aziendali.

1.2.2 Studio di fattibilità

Un sistema per l'assegnamento automatico che si basa su metodi non supervisionati può ridurre i costi definiti alla sezione precedente. Per realizzare questa funzionalità bisogna quindi implementare un algoritmo di classificazione che, estraendo opportune informazioni dal contenuto di un ticket, provvede alla sua classificazione in più gruppi definiti dall'azienda stessa.

I costi principali relativi alla soluzione sono legati allo sviluppo del software e alla manutenzione del server per l'*hosting* del sistema.

1.3 Specifiche

In questa sezione sono presenti le varie specifiche che definiscono la struttura ed il comportamento del sistema.

1.3.1 Casi d'uso

Nelle seguenti sezioni sono analizzati gli attori ed i casi d'uso che definiscono il comportamento del sistema.

Attori

Gli attori che devono utilizzare il sistema sono:

- *Utente registrato*, un utente esterno all'azienda che si è registrato al portale per la gestione dei ticket;
- *Utente non registrato*, un qualsiasi utente esterno all'azienda che non si è registrato al portale;
- *Operatore*, un dipendente dell'azienda;
- *Amministratore*, un dipendente dell'azienda con compiti di gestione.

Elenco dei casi d'uso

Nella tabella 1.1 vengono descritti i casi d'uso individuati con i rispettivi codici, mentre il diagramma UML dei casi d'uso è visibile in figura 1.1.

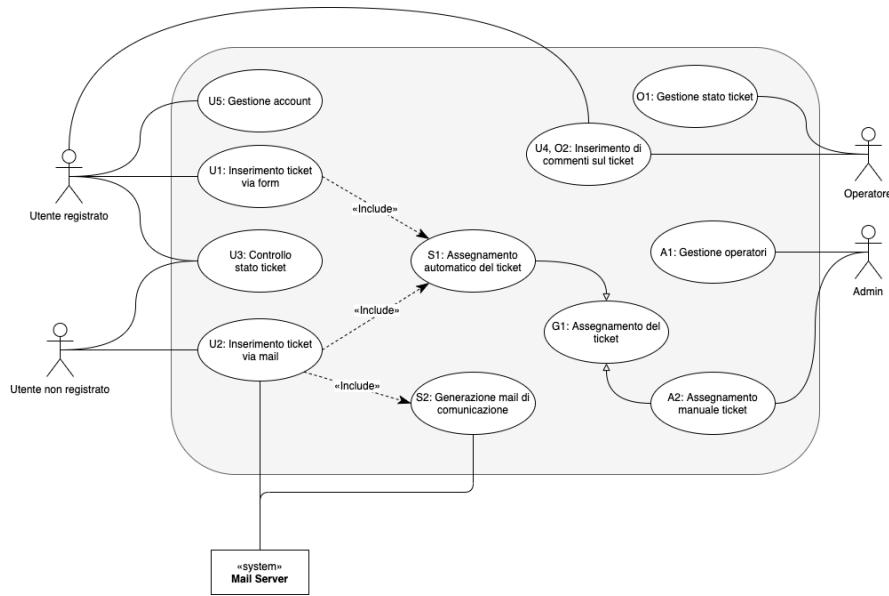


Figura 1.1: Diagramma UML dei casi d'uso

Codice	Descrizione
U1	Un utente registrato crea un nuovo ticket da form
U2	Un utente non registrato crea un nuovo ticket inviando una mail
U3	Un utente (sia registrato sia non registrato) può controllare lo stato del ticket
U4	Un utente registrato può usare il portale per comunicare con un operatore, inserendo dei commenti su un ticket
U5	Un utente crea e gestisce l'account
S1	Il sistema assegna in automatico un nuovo ticket ad un gruppo di operatori
S2	Il sistema fornisce una mail ad un utente non registrato per poter comunicare con un operatore
O1	Un operatore può cambiare lo stato del ticket (aperto, chiuso, assegnato, in corso)
O2	Un operatore può inserire commenti sui ticket assegnati al proprio gruppo
A1.1	Un amministratore interno all'azienda inserisce un nuovo operatore
A1.2	Un amministratore interno all'azienda modifica un operatore
A1.3	Un amministratore interno all'azienda elimina un operatore
A2	Un amministratore interno all'azienda assegna manualmente i ticket ad gruppo di operatori
G1	Caso d'uso generico che rappresenta l'inserimento di un ticket

Tabella 1.1: Casi d'uso

Dettaglio dei casi d'uso

Nel seguente elenco sono analizzati in maniera dettagliata tutti i casi d'uso, identificandone descrizione, attori, precondizioni, passi principali, situazioni eccezionali e postcondizioni.

- **U1:** *Creazione ticket via form*

- **Descrizione:** l'utente desidera creare un nuovo ticket tramite form
- **Attori:** utente
- **Precondizioni:** l'utente ha effettuato il login e si trova nel form di creazione ticket
- **Passi principali:**
 1. L'utente inserisce titolo e descrizione del ticket
 2. L'utente crea il nuovo ticket
 3. Il sistema inserisce il nuovo ticket in database
 4. Il sistema assegna in automatico il ticket
- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** il ticket è inserito in database

- **U2:** *Creazione ticket via mail*

- **Descrizione:** l'utente desidera creare un nuovo ticket tramite mail
- **Attori:** utente
- **Precondizioni:** nessuna
- **Passi principali:**
 1. L'utente invia una mail al sistema di ticketing
 2. Il sistema inserisce i dati in database
 3. Il sistema genera una mail di comunicazione
 4. L'utente viene notificato dal sistema dell'avvenuta ricezione

- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** il ticket è inserito in database

- **U3:** *Controllo stato ticket*

- **Descrizione:** l'utente controlla lo stato del ticket (aperto, assegnato, in corso, chiuso)
- **Attori:** utente
- **Precondizioni:** l'utente conosce l'identificativo del ticket
- **Passi principali:**
 1. L'utente naviga alla pagina di stato dello specifico ticket
 2. Il sistema invia all'utente le varie informazioni sul ticket, tra cui titolo, descrizione e stato
- **Situazioni eccezionali:** se il ticket è stato creato tramite mail, è visibile solamente lo stato del ticket
- **Postcondizioni:** nessuna

- **U4:** *Inserimento di commenti sul ticket*

- **Descrizione:** l'utente inserisce un commento in un ticket già aperto
- **Attori:** utente
- **Precondizioni:** l'utente deve aver effettuato il login ed il ticket deve esistere
- **Passi principali:**
 1. L'utente inserisce un nuovo commento sul ticket desiderato
 2. L'utente salva il commento
 3. Il sistema inserisce il commento in database
- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** il commento è inserito in database

- **U5:** *Gestione account*

- **Descrizione:** l’utente modifica i dati relativi al suo account inseriti in fase di registrazione

- **Attori:** utente

- **Precondizioni:** l’utente deve aver effettuato il login

- **Passi principali:**

1. L’utente inserisce i nuovi dati da aggiornare

2. L’utente salva i nuovi dati

3. Il sistema riceve i dati e aggiorna le informazioni in database

- **Situazioni eccezionali:** nessuna

- **Postcondizioni:** il ticket viene inserito in database dal sistema

- **S1:** *Assegnamento automatico del ticket*

- **Descrizione:** il ticket è assegnato automaticamente ad un gruppo

- **Attori:** nessuno

- **Precondizioni:** il sistema ha ricevuto e inserito in database un nuovo ticket

- **Passi principali:**

1. Il sistema valuta il miglior gruppo a cui assegnare il ticket sulla base delle informazioni presenti in database

2. Il sistema aggiorna le informazioni in database di conseguenza

3. Il sistema notifica il corrispondente gruppo di operatori

- **Situazioni eccezionali:** il ticket può non essere assegnato se non sono presenti abbastanza informazioni

- **Postcondizioni:** il ticket è assegnato ad un gruppo di operatori

- **S2:** *Generazione mail di comunicazione*

- **Descrizione:** il sistema genera una mail che l’utente userà per la comunicazione con gli operatori

- **Attori:** nessuno

- **Precondizioni:** il sistema ha ricevuto e inserito in database un nuovo ticket
- **Passi principali:**
 1. Il sistema genera un indirizzo mail per le comunicazioni sullo specifico ticket
 2. Il sistema comunica all'utente questo indirizzo mail
- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** l'utente è in possesso di un indirizzo mail per la comunicazione con gli operatori

- **O1:** *Gestione stato ticket*

- **Descrizione:** l'operatore cambia lo stato del ticket
- **Attori:** operatore
- **Precondizioni:** il ticket deve esistere in database, l'operatore deve aver effettuato il login;
- **Passi principali:**
 1. L'operatore modifica lo stato del ticket
 2. Il sistema aggiorna lo stato del ticket in database
 3. Il sistema notifica l'utente dell'avvenuto cambiamento di stato
- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** il ticket ha il nuovo stato e l'utente è stato avvisato

- **O2:** *Inserimento di commenti sul ticket*

- **Descrizione:** un operatore aggiunge commenti su uno specifico ticket
- **Attori:** operatore
- **Precondizioni:** il ticket deve esistere in database e dev'essere stato assegnato, l'operatore deve aver effettuato il login
- **Passi principali:**

1. L'operatore inserisce un nuovo commento
2. L'operatore salva il commento
3. Il sistema inserisce il commento in database
4. Il sistema notifica l'utente dell'avvenuto inserimento del commento

- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** il commento è inserito in database

- **A1.1:** *Gestione operatori - inserimento*

- **Descrizione:** un amministratore inserisce un nuovo operatore
- **Attori:** amministratore
- **Precondizioni:** l'amministratore deve aver effettuato il login
- **Passi principali:**

1. L'amministratore inserisce i dati del nuovo operatore
2. L'amministratore salva i nuovi dati
3. Il sistema inserisce i dati in database

- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** il nuovo operatore risulta inserito

- **A1.2:** *Gestione operatori - modifica*

- **Descrizione:** un amministratore modifica un operatore
- **Attori:** amministratore
- **Precondizioni:** l'amministratore deve aver effettuato il login
- **Passi principali:**

1. L'amministratore modifica i dati dell'operatore
2. L'amministratore salva i nuovi dati
3. Il sistema modifica i dati in database

- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** l'operatore risulta aggiornato

- **A1.3:** *Gestione operatori - elimina*
 - **Descrizione:** un amministratore elimina un operatore
 - **Attori:** amministratore
 - **Precondizioni:** l'amministratore deve aver effettuato il login
 - **Passi principali:**
 1. L'amministratore elimina un operatore
 2. Il sistema rimuove l'operatore dal database
 - **Situazioni eccezionali:** nessuna
 - **Postcondizioni:** l'operatore risulta eliminato
- **A2:** *Assegnamento manuale del ticket*
 - **Descrizione:** un amministratore assegna manualmente il ticket ad un gruppo di operatori
 - **Attori:** amministratore
 - **Precondizioni:** l'amministratore deve aver effettuato il login
 - **Passi principali:**
 1. L'amministratore cambia i dati del ticket inserendo il gruppo a cui assegnarlo
 2. L'amministratore salva i dati
 3. Il sistema modifica i dati in database
 - **Situazioni eccezionali:** nessuna
 - **Postcondizioni:** il ticket risulta assegnato al gruppo specificato
- **G1:** *Assegnamento del ticket*
 - **Descrizione:** caso d'uso generico che rappresenta l'assegnamento di un ticket ad un gruppo di operatori
 - **Attori:** nessuno
 - **Precondizioni:** nessuna
 - **Passi principali:** nessuno

- **Situazioni eccezionali:** nessuna
- **Postcondizioni:** il ticket risulta assegnato ad un gruppo di operatori

1.3.2 Requisiti funzionali

Di seguito è presente l'elenco completo di tutte le specifiche funzionali necessarie al fine della realizzazione del software:

1. *Creazione di un account*, ovvero l'inserimento dei dati relativi per la creazione di un account utente (tra cui nome, password ed email);
2. *Gestione dell'account*, ovvero la modifica delle informazioni dell'account utente;
3. *Login e logout* di utenti ed operatori;
4. *Creazione di ticket tramite form* da parte di un utente autenticato;
5. *Creazione di ticket tramite mail* da parte del sistema;
6. *Controllo dello stato del ticket*, sia con informazioni aggiuntive (per utenti autenticati) sia senza di esse (per utenti non autenticati);
7. *Assegnamento automatico del ticket* da parte del sistema ad un gruppo di operatori;
8. *Assegnamento manuale del ticket* da parte dell'amministratore ad un gruppo di operatori;
9. *Creazione, modifica ed eliminazione di operatori* da parte dell'amministratore.

Tutte queste funzionalità sono riassunte nella tabella 1.2.

Codice	Descrizione
FU1	Creazione account
FU2	Gestione account
FU3	Login e logout
FT1	Creazione ticket tramite form
FT2	Creazione ticket tramite mail
FT3	Controllo stato ticket
FT4	Inserimento commenti
FT5	Gestione stato ticket
FT6	Assegnamento automatico ticket
FT7	Assegnamento manuale ticket
F01	Creazione operatore
F02	Modifica operatore
F03	Eliminazione operatore

Tabella 1.2: Requisiti funzionali

1.4 Architettura software

1.4.1 Component Diagram

In figura 1.2 è visibile il *component diagram* del sistema. Questo diagramma rappresenta la struttura interna del sistema dal punto di vista dei componenti principali e delle relazioni tra di essi.

Come si può vedere, il sistema è suddiviso in due parti:

- il *front-end* che si occupa della gestione dell’interfaccia grafica;
- il *back-end* che gestisce la *business logic*.

Il *back-end* fornisce delle API al *front-end* tramite i vari componenti e comunica sia con il DBMS sia con il server mail esterno.

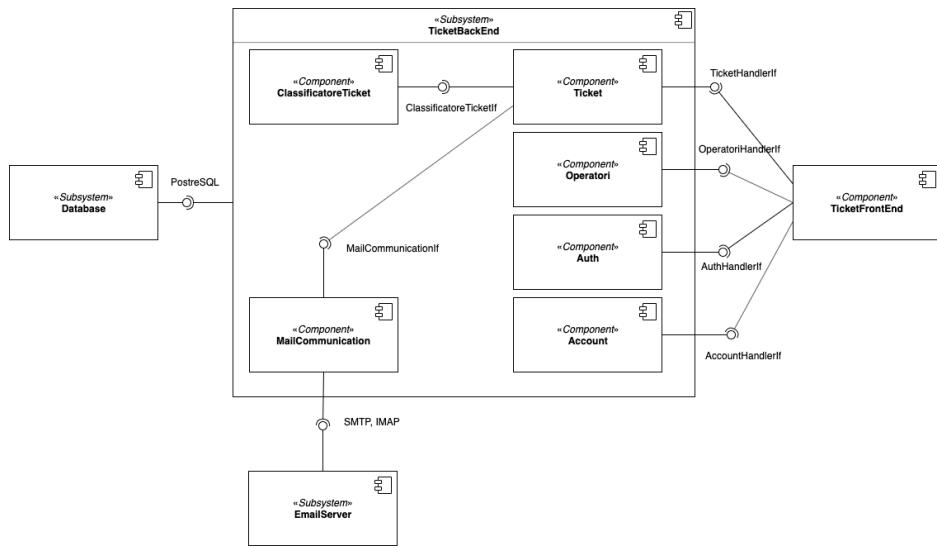


Figura 1.2: Diagramma dei componenti

1.4.2 Class Diagram

Il *class diagram* visibile in figura 1.3 descrive le varie classi che compongono il sistema.

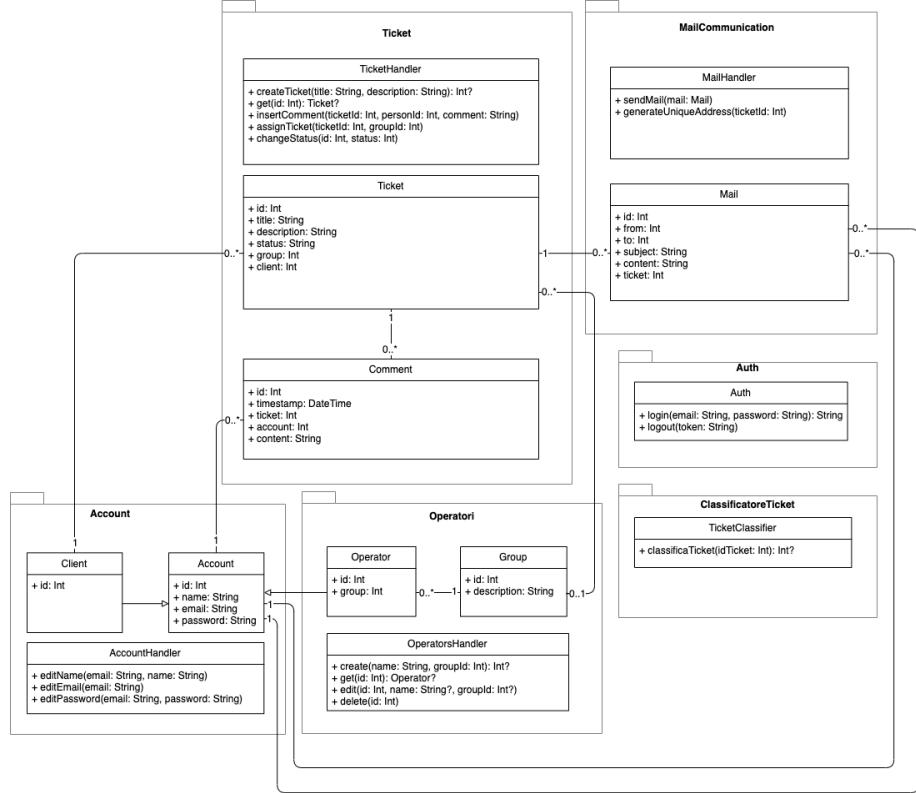


Figura 1.3: Diagramma delle classi

Le classi *Handler* si occupano della comunicazione con il client e implementano nel codice i vari casi d'uso. La classe *TicketClassifier* implementa invece l'algoritmo per la classificazione dei ticket ed il loro assegnamento ai gruppi di operatori.

Notare che la nostra classe `Client` fa riferimento all'attore di tipo “utente”: questo perché `User` fa riferimento ad una classe già esistente e fornita dal framework, come sarà visibile nella sezione 2.2.1.

1.5 Architettura hardware

L'architettura del sistema è una *2-tier architecture*, composta da un client per la GUI e l'interazione con l'utente e da un server che implementa le logiche

che definiscono il sistema.

1.5.1 Architettura hardware

La figura 1.4 rappresenta un diagramma architettonico del sistema.

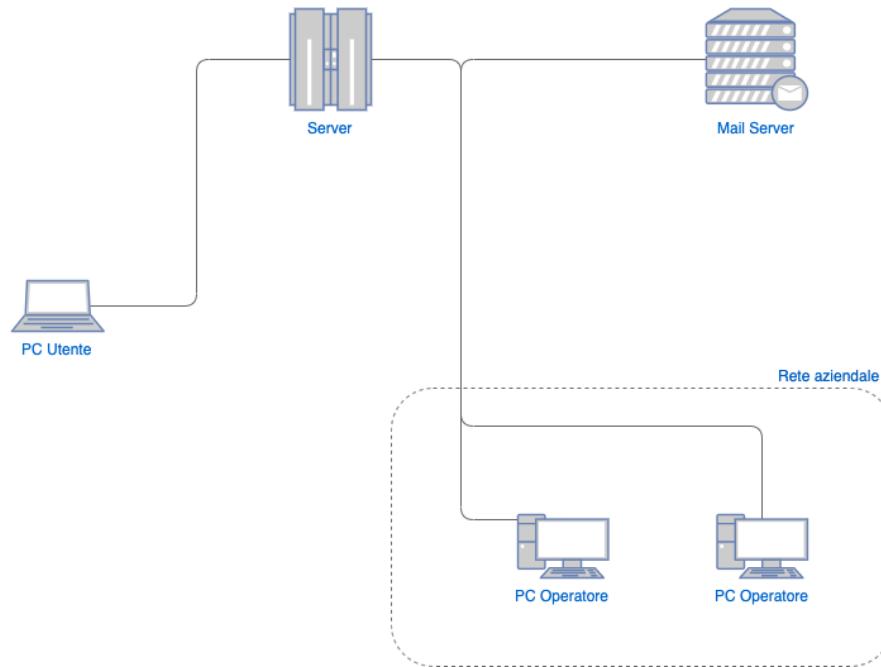


Figura 1.4: Architettura hardware

1.5.2 Deployment Diagram

Il *deployment diagram* in figura 1.5 evidenzia i dispositivi considerati e come i vari componenti identificati alla sezione 1.4.1 sono istanziati su di essi.

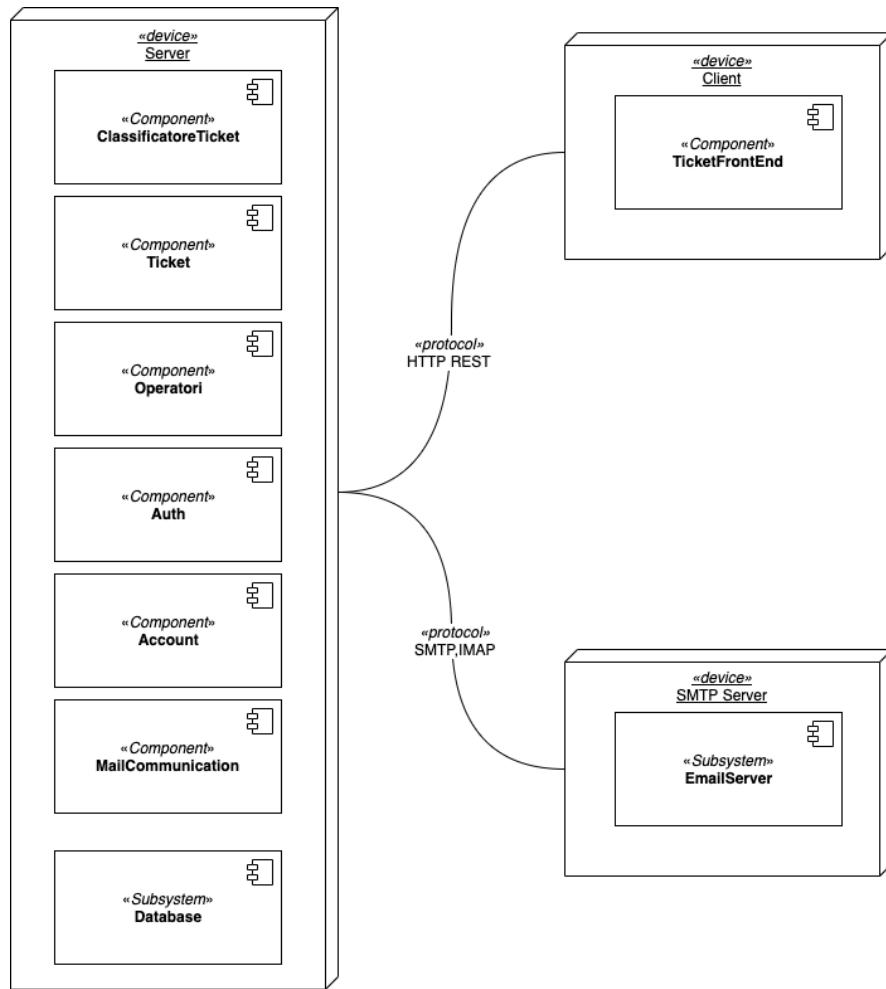


Figura 1.5: Diagramma di deployment

Nel diagramma sono presenti tre *device*:

1. Un client che gestirà il *front-end* del sistema;
2. Un nostro *server* che si occuperà anche del database a fine di semplificare e velocizzare l'interazione tra logica e dati;
3. Un *server email* esterno per la gestione delle caselle mail.

Capitolo 2

Iterazione 1

2.1 Introduzione

In questa fase vengono implementate le strutture e le funzioni utili al fine di avere un *Minimum Viable Product* funzionante.

La funzione principale implementata è la FU3, ovvero il *login* ed il *logout* di persone con account (sia utenti sia operatori). Altre funzioni realizzate sono

- FU1, la creazione di un account;
- FT7, l'assegnamento manuale di un ticket;
- F01, la creazione di un operatore;
- F02, la modifica di un operatore;
- F03, l'eliminazione di un operatore.

Oltre all'autenticazione, è stata realizzata anche la struttura dell'applicazione sia lato back-end, in linguaggio *Python* e con il framework *Django*, sia lato front-end, usando i linguaggi *JavaScript*, *HTML* e *CSS* e con la libreria *React*.

2.2 Back-end e REST API

Il back-end è un server REST realizzato grazie ai framework *Django* e *Django REST Framework*, che si occupano rispettivamente delle comuni necessità di un'applicazione web¹.

2.2.1 Autenticazione

L'autenticazione è gestita trasparentemente grazie a *Django*, che offre tutte le possibili funzioni per la gestione di utenti e di sessioni.

Come esempio, di seguito è visibile la definizione di un **Account**, entità usata per l'accesso al portale.

```
from django.db import models
from django.contrib.auth.models import User

class Account(models.Model):
    """
        Account models
    """

    user = models.OneToOneField(User, on_delete=models.CASCADE)
    name = models.CharField(max_length=72, null=True)
    email = models.CharField(max_length=72)
```

Come si può vedere, però, la classe **Account** non estende la classe predefinita **User**, questo perché il framework stesso consiglia la creazione di un riferimento ad un'istanza di utente piuttosto che la classica ereditarietà².

Qui si vede inoltre perché, come anticipato dal diagramma delle classi, non abbiamo chiamato **User** la classe relativa agli attori di tipo “utente”: per

¹Comprendente accesso a database, templating, autorizzazioni, etc.

²<https://docs.djangoproject.com/en/3.1/topics/auth/customizing/>

evitare un conflitto di nomi con le classi fornite dal framework.

Per limitare l'accesso invece a particolari endpoint viene fornito il decoratore `@login_required`, che va posto sopra le funzioni e/o metodi che necessitano di una sessione attiva, come si può qui vedere nell'estratto di codice.

```
@login_required
@api_view(['POST'])
def add_operator(request):
    """
    Let admins add operators
    """

    try:
        user = User.objects.create_user(username=
            request.data["username"], password=request.
            data["password"])

        user.save()
        acc = Account(user=user, email=request.data["
            username"])
        acc.save()
        operator = Operator(account=acc, group=None)
        operator.save()

    except Exception as e:
        return Response(status=status.
            HTTP_500_INTERNAL_SERVER_ERROR)
    return Response(status=status.HTTP_201_CREATED)
```

2.2.2 App AI

Una *app* in *Django* è un pacchetto Python che fornisce delle funzionalità e ha delle proprie configurazioni. Nel nostro caso, `ai` è la app principale (e anche l'unica) che offre tutte le funzionalità definite nella fase precedente.

Seppur in *Django* è possibile utilizzare dei *template HTML* per la realizzazione anche dell’interfaccia grafica, essi non sono usati nel nostro progetto, in quanto il front-end è separato e realizzato in modo indipendente dal back-end.

I file presenti di particolare interesse sono:

- `models.py` che contiene tutti i modelli usati per derivare la struttura del database e le migrazioni;
- `tests.py` contenente i vari test;
- `urls.py` per la definizione degli endpoint forniti;
- `views.py`, il quale elenca le varie rappresentazioni dei dati e le azioni possibili su di essi.

Le relazioni statiche presenti nella app AI sono visibili in figura 2.1. Come si può vedere, sono presenti sia classi definite da noi sia classi del framework.

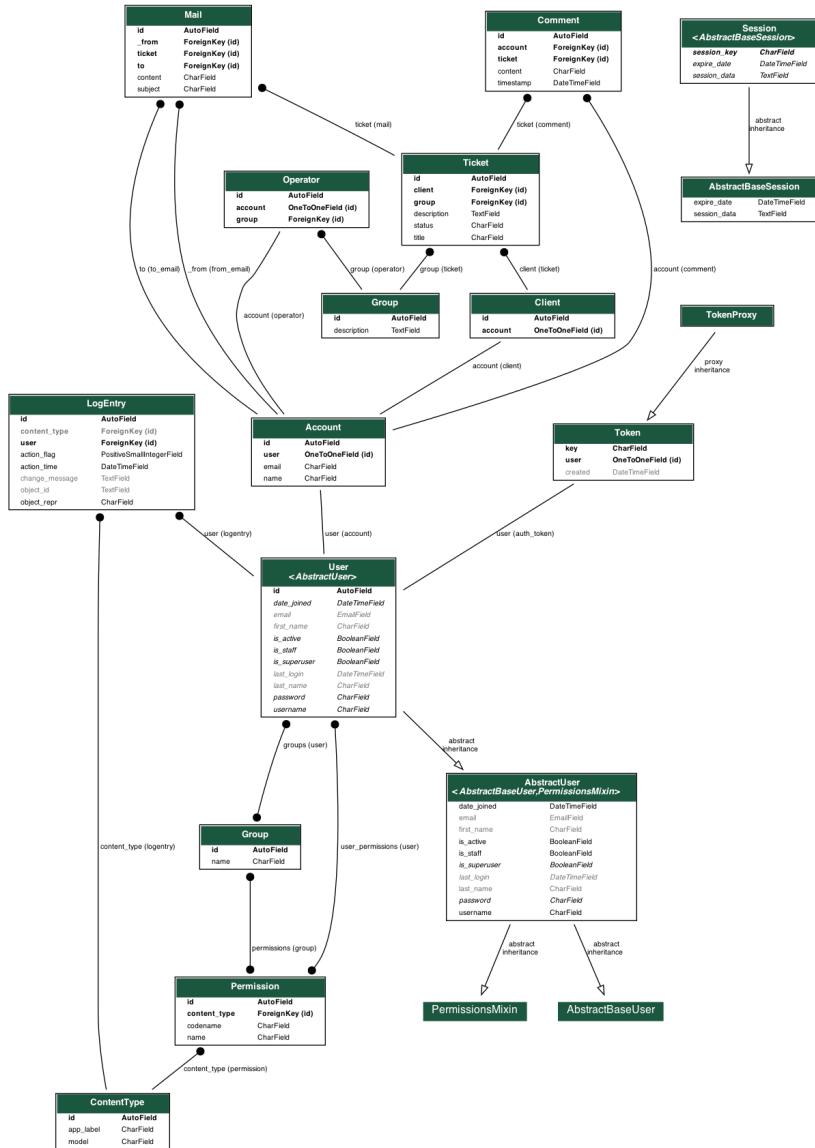


Figura 2.1: Relazioni tra classi di AI

2.2.3 Console di amministrazione

La console di amministrazione, usata dagli attori *Amministratori*, è generata automaticamente dal framework *Django* su un sottoinsieme delle entità da noi definite a livello di codice.

Grazie ad essa è possibile

- assegnare manualmente un operatore ad un gruppo;
- assegnare manualmente un ticket ad un gruppo;
- creare, modificare ed eliminare operatori ed i loro account.

Nella figura 2.2 è visibile il form per l'accesso a questa console, disponibile all'endpoint `/admin/`.

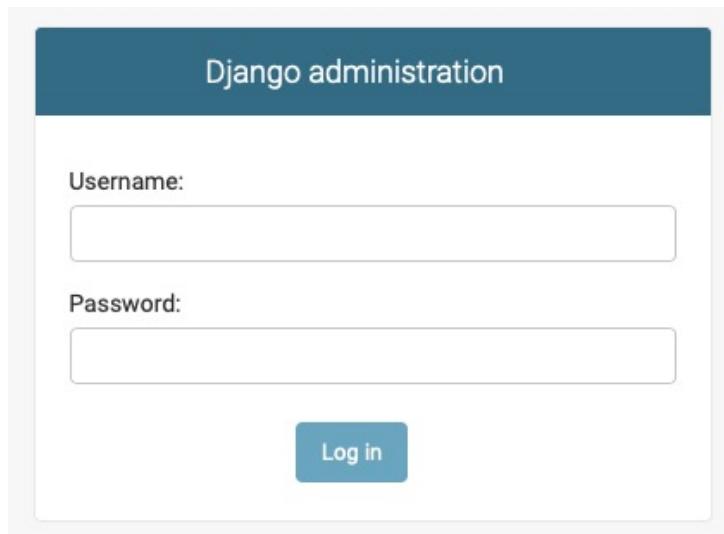


Figura 2.2: Admin console di Django

2.2.4 API endpoints

Il server mette a disposizione degli endpoint per fornire, creare e modificare i dati relativi al dominio. Questi sono visibili nella tabella 2.1.

Notare che questi endpoint sono solamente quelli realizzati in questa iterazione: nelle iterazioni future se ne aggiungeranno altri in base alle necessità implementative di ogni funzionalità.

Metodo	Endpoint	Funzione
POST	v1/auth/	Login
POST	v1/signup/	Registrazione
POST	v1/operator/add/	Creazione di operatori
GET/PUT/DELETE	v1/operator/<int:pk>/	Gestione di operatori
POST	v1/client/add/	Creazione di clienti (i.e. utenti)
GET/PUT/DELETE	v1/client/<int:pk>/	Gestione di clienti
POST	v1/group/add/	Creazione di gruppi
GET/PUT/DELETE	v1/group/<int:pk>/	Gestione di gruppi

Tabella 2.1: API fornite dal server

2.3 Front-end

Il front-end è una *Single Page Application* realizzata in *JavaScript* con l'ausilio della libreria open source *React*. Grazie ad essa è possibile costruire componenti grafici riutilizzabili, i quali possono anche contenere logica interna complessa.

2.3.1 Struttura SPA

Il codice è suddiviso in varie cartelle³, ma la cartella `src` contiene tutto il codice. A sua volta, `src` è divisa in:

- `components`, contenente componenti di pura grafica, indipendenti da dati di dominio;
- `resources`, per alcune risorse statiche, come ad esempio immagini ed icone;
- `scenes`, che racchiude le varie pagine con cui l'utente interagisce;

³La struttura del progetto non è data da un framework, ma da esperienza personale.

- `theme`, per i CSS ed i font usati nel sito.

Non appartenenti a questa cartella sono `App.jsx`, che si occupa del rendering della SPA, e `AppRouting.jsx`, che definisce le varie rotte visitabili.

2.3.2 Pagine visitabili

Le pagine visitabili sono⁴

- una pagina di *login*;
- una pagina di *registrazione*;
- una home;
- una pagina per la visualizzazione dei propri dati;
- pagine per la visualizzazione dei ticket, del loro stato e per l'inserimento di commenti.

Come esempio, in figura 2.3 è visibile il form di registrazione utente.

⁴La home, la pagina di account e le pagine relative ai ticket sono implementate in iterazioni future

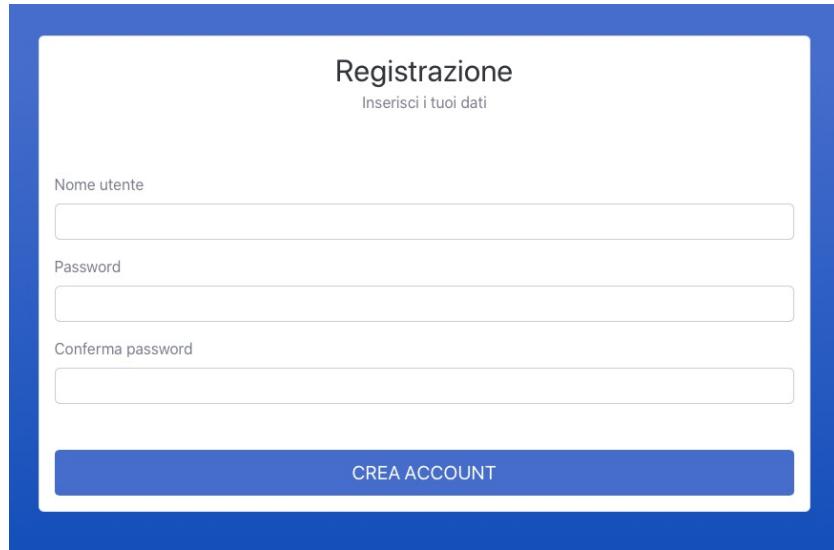


Figura 2.3: Pagina di registrazione

2.4 Analisi statica

2.4.1 Back-end

Per l'analisi statica del back-end sono stati usati diversi tool, dei quali il principale è *pylint*, un *linter* configurabile per il linguaggio *Python*. Esso mette a disposizione varie funzionalità:

- imposizione di *coding standards* e di *code style*;
- *error detection*, sia a livello sintattico sia a livelli di tipi;
- *refactoring* per codice non usato e/o duplicato;
- integrazione con vari IDE al fine di fornire tutto ciò in tempo reale.

Pylint è configurato tramite il file `.pylintrc` presente nella cartella del back-end, il quale contiene tutti parametri e le impostazioni da noi scelti per

diminuire la complessità del codice ed aumentarne la chiarezza. È possibile utilizzare lo script `lint-backend.sh` per eseguire il linting del progetto.

Oltre a *warning* ed errori, nell'output è presente anche un singolo numero (su una scala da 1 a 10) che valuta il codice. In particolare, alla fine di questa iterazione questo valore è pari a 8.78.

2.4.2 Front-end

Per il front-end è stata utilizzata *ESLint*, una libreria JavaScript che analizza staticamente il codice e risolve problemi sia di stile, sia di code quality.

In particolare, *ESLint* dà la possibilità di definire dei propri standard del codice, eventualmente derivandoli da altri standard. Ad esempio, lo standard da noi usato prende spunto dallo stile di *Airbnb*.

È possibile avere un report dell'analisi statica tramite il comando `yarn lint` nella cartella `front-end`. Gli specifici stili estesi e le regole sovrascritte sono invece visibili nel file `package.json`.

2.5 Analisi dinamica

2.5.1 Testing

Ci siamo occupati di scrivere dei casi di test per poter avere una buona copertura del codice implementato in questa fase, sia lato front-end sia lato back-end.

Testing back-end

I casi di test sono definiti nel file `tests.py`. Il framework *Django* mette a disposizione un package per la definizione di *unit test*, ed in particolare

offre la classe `TestCase` che è possibile estendere per definire casi di test personalizzati.

I casi di test sono stati definiti su tutti i modelli e tutte le *views* realizzati nell'iterazione. Come esempio, di seguito è visibile il caso di test per la gestione dell'autenticazione (commenti esclusi).

```
class AuthTestCase(TestCase):
    def setUp(self):
        self.factory = RequestFactory()
        self.user = User.objects.create_user(username='name',
                                             password='sur')

    def test_auth(self):
        request = self.factory.post('/auth/', {
            'username': 'name',
            'password': 'sur',
        })

    def test_logout(self):
        request = self.factory.get('/logout/')
```

O ancora, i casi di test per gli endpoint relativi all'entità `Client` (rimossi commenti e docstring per brevità).

```
class ClientTestCase(TestCase):
    def setUp(self):
        self.factory = RequestFactory()
        self.user = User.objects.create_user(username='em@ma.il',
                                             password='sur')
        self.user.save()
        self.acc = Account(user=self.user, email='em@ma.il')
        self.acc.save()
        self.client = Client(account=self.acc)
        self.client.save()
```

```

def test_delete_client(self):
    request = self.factory.delete('/client/1/')
    request.user = self.user
    response = handle_client(request, 1)
    self.assertEqual(response.status_code, 200)

def test_patch_client(self):
    request = self.factory.patch('/client/1')
    request.user = self.user
    response = handle_client(request, 1)
    self.assertEqual(response.status_code, 405)

def test_delete_fail_client(self):
    request = self.factory.delete('/client/12345/')
    request.user = self.user
    response = handle_client(request, 12345)
    self.assertEqual(response.status_code, 404)

def test_get_client(self):
    request = self.factory.get('/client/1/')
    request.user = self.user
    response = handle_client(request, 1)
    self.assertEqual(response.status_code, 200)

def test_get_fail_client(self):
    request = self.factory.get('/client/12345/')
    request.user = self.user
    response = handle_client(request, 12345)
    self.assertEqual(response.status_code, 404)

def test_put_client(self):
    request = self.factory.put('/client/1/', {
        "account": {

```

```

        'id': 1,
        'email': 'changed@ma.il'
    },
}, content_type='application/json')
request.user = self.user
response = handle_client(request, 1)
self.assertEqual(response.status_code, 200)

def test_add_client(self):
    request = self.factory.post('/client/add/', {
        "username": "op",
        "password": "psw"
    })

    request.user = self.user
    response = add_client(request)
    self.assertEqual(response.status_code, 201)

def test_add_fail_client(self):
    request = self.factory.post('/client/add', {
        "noparam": "op",
        "password": "psw"
    })

    request.user = self.user
    response = add_client(request)
    self.assertEqual(response.status_code, 500)

```

Testing front-end

Lato front-end, è stato usato il framework *Jest* e la libreria *testing-library* al fine di esaminare il più possibile la struttura ed il comportamento dell'applicazione web.

Il framework *Jest* riconosce in automatico tutti i file `Name.test.js` come dei file contenenti casi di test, li analizza e li esegue tutti in automatico, a qualsiasi livello essi si trovino.

Ad esempio, un estratto del file `AppRouting.test.jsx` è visibile nel seguente snippet.

```
describe('App routing', () => {
  it('has a home page', () => {
    renderWithRoute('/');
    expect(screen.getByText(/home/i)).toBeInTheDocument();
  });
}

it('has an account page', () => {
  renderWithRoute('/account');
  screen.getAllByText(/account/i).forEach((x) =>
    expect(x).toBeInTheDocument());
});

it('has a 404 page', () => {
  renderWithRoute('/some/random/route');
  expect(screen.getByText(/404/i)).toBeInTheDocument();
});
});
```

Tramite la libreria di testing usata è possibile anche simulare inserimenti e click dell'utente, come è visibile in questo estratto di `Login.test.jsx`.

```
describe('Login page', () => {
  it('has a username input', () => {
    renderWithHistory(<Login />);
    expect(screen.getByPlaceholderText('Inserisci il nome utente')).toBeInTheDocument();
  });
});
```

```

it('has a password input', () => {
  renderWithHistory(<Login />);
  expect(screen.getByPlaceholderText('Inserisci la
    password')).toBeInTheDocument();
}) ;

it('correctly accepts inputs', () => {
  renderWithHistory(<Login />);
  fireEvent.change(
    screen.getByPlaceholderText('Inserisci il nome
      utente'),
    { target: { value: 'username' } },
  );
  fireEvent.change(
    screen.getByPlaceholderText('Inserisci la
      password'),
    { target: { value: 'password123' } },
  );
  fireEvent.click(screen.getByTestId('login-button'))
  ;
}) ;
});

```

2.5.2 Coverage

Back-end

Per il back-end è stato usato *Coverage.py*, uno strumento atto a misurare la copertura del codice per il linguaggio Python. È anche disponibile un output interattivo in HTML, il cui output per il codice di questa iterazione è visibile in figura 2.4.

<i>Module ↗</i>	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
ai/__init__.py	0	0	0	100%
ai/admin.py	6	0	0	100%
ai/apps.py	3	0	0	100%
ai/migrations/0001_initial.py	7	0	0	100%
ai/migrations/__init__.py	0	0	0	100%
ai/models.py	39	0	0	100%
ai/tests.py	154	0	0	100%
ai/urls.py	5	0	0	100%
ai/views.py	136	23	0	83%
manage.py	12	2	0	83%
ticket/__init__.py	0	0	0	100%
ticket/asgi.py	4	4	0	0%
ticket/settings.py	19	0	0	100%
ticket/urls.py	3	0	0	100%
ticket/wsgi.py	4	4	0	0%
Total	392	33	0	92%

Figura 2.4: Back-end code coverage

Se si desidera, è possibile utilizzare lo script `coverage-backend.sh` per generare i report HTML nella cartella `back-end/htmlcov`.

Front-end

Il framework di testing *Jest* permette anche di generare automaticamente la coverage del codice tramite il comando `yarn coverage`, esplorabile interattivamente tramite un documento HTML. In figura 2.5 è visibile la pagina principale di questo documento, che presenta la copertura di istruzioni, branches, funzioni e righe per ogni cartella del progetto.

Metrica	Copertura percentuale
Istruzioni	91.30%
Branches	85.71%
Funzioni	97.67%
Righe	90.91%

Tabella 2.2: Coverage totale per il front-end

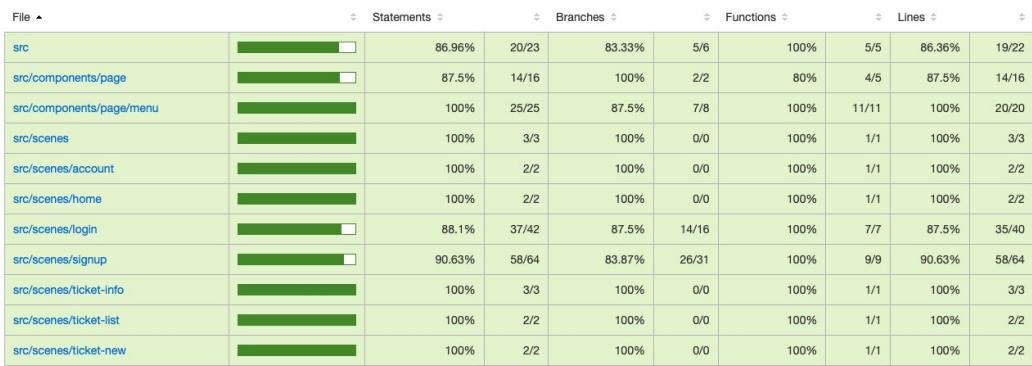


Figura 2.5: Front-end code coverage

In totale, secondo questo primo calcolo, la coverage assume i valori visibili in tabella 2.2.

Capitolo 3

Iterazione 2

3.1 Introduzione

In questa iterazione vengono implementate le funzionalità base per la gestione dei ticket, il core della nostra applicazione.

In particolare, le funzionalità implementate sono le seguenti:

- FT1, la creazione di ticket tramite form;
- FT3, il controllo dello stato del ticket, ma solamente per utenti loggati;
- FT4, l'inserimento di commenti sul ticket da parte di utenti loggati ed operatori;
- FT5, la gestione dello stato del ticket da parte di operatori.

È quindi garantita una comunicazione tra operatori e clienti, funzionalità di base per una buona *User Experience* in questo ambito.

3.2 Back-end

3.2.1 Rimozione di Django REST Framework

In questa iterazione abbiamo deciso di rimuovere *Django REST Framework* per motivi sia di prestazioni sia di complessità nella gestione dell'autenticazione.

Infatti, il framework dà dei problemi con l'autenticazione fornita da *Django*, in quanto può non effettuare il parsing corretto dei cookies di autenticazione in presenza di richieste di tipo `OPTIONS` fatte dai browser. Questo tipo di problema ci portava ad avere un'autenticazione funzionante ma, ad una successiva chiamata che avesse il tipo `OPTIONS`, il framework rispondeva con `302` ed effettuava il re-indirizzamento all'endpoint con il metodo corretto, senza però instradare anche i cookies.

Ciò risultava in un login funzionante, ma ogni altra richiesta era come se venisse fatta da un utente anonimo.

3.2.2 API endpoints

Nel back-end sono stati realizzati gli endpoint visibili in tabella 3.1.

Questi endpoint sono le fondamenta per il funzionamento base della comunicazione tra operatore e cliente:

1. l'utente crea un nuovo ticket tramite l'endpoint `/v1/ticket/add/`;
2. l'utente visualizza i propri ticket con `/v1/tickets`;
3. volendo, sia gli utenti sia gli operatori possono visualizzare un ticket con `/v1/ticket/<pk>` ed aggiungere commenti con `/v1/comment/add/`.

Di seguito un estratto di codice che rappresenta le varie funzioni per la creazione di ticket e di commenti.

Endpoint	Descrizione
GET /v1/tickets	Ritorna l'elenco dei ticket
POST /v1/ticket/add/	Crea un nuovo ticket
GET /v1/ticket/<pk>/	Ottiene le informazioni di un ticket
PUT /v1/ticket/<pk>/	Aggiorna lo stato di un ticket
POST /v1/comment/add/	Crea un nuovo commento su un ticket

Tabella 3.1: Endpoint

```

@login_required
@renderer_classes(JSONRenderer)
def add_ticket(request):
    try:
        request.POST = decode_json_body(request)
        ticket = Ticket.objects.create(
            title=request.POST["title"],
            description=request.POST["description"],
            client=Client.objects.get(pk=request.POST[
                "client"]))
        ticket.save()
    except:
        return HttpResponse(status=status.
                            HTTP_500_INTERNAL_SERVER_ERROR)
    return HttpResponse(status=status.HTTP_201_CREATED)

@login_required
@renderer_classes(JSONRenderer)
def add_comment(request):
    try:
        request.POST = decode_json_body(request)
        account = Account.objects.get(pk=request.POST[

```

```

        account"])
ticket = Ticket.objects.get(pk=request.POST["ticket"])
comment = Comment.objects.create(
    timestamp=request.POST["timestamp"],
    ticket=ticket,
    account=account,
    content=request.POST["content"])

comment.save()
except:
    return HttpResponse(status=status.HTTP_500_INTERNAL_SERVER_ERROR)
return HttpResponse(status=status.HTTP_201_CREATED)

```

3.2.3 Testing ed analisi statica

Alla fine di questa iterazione la qualità del codice è di 9.30 su 10 secondo *pylint*.

La coverage invece è visibile nella relativa cartella ed ammonta all'85%, con 30 test positivi.

3.3 Front-end

3.3.1 Pagine aggiunte

Il front-end rispecchia le nuove API aggiunte. In particolare, sono state realizzate le seguenti pagine:

- lista dei ticket;
- creazione di un ticket, solo per gli utenti loggati;

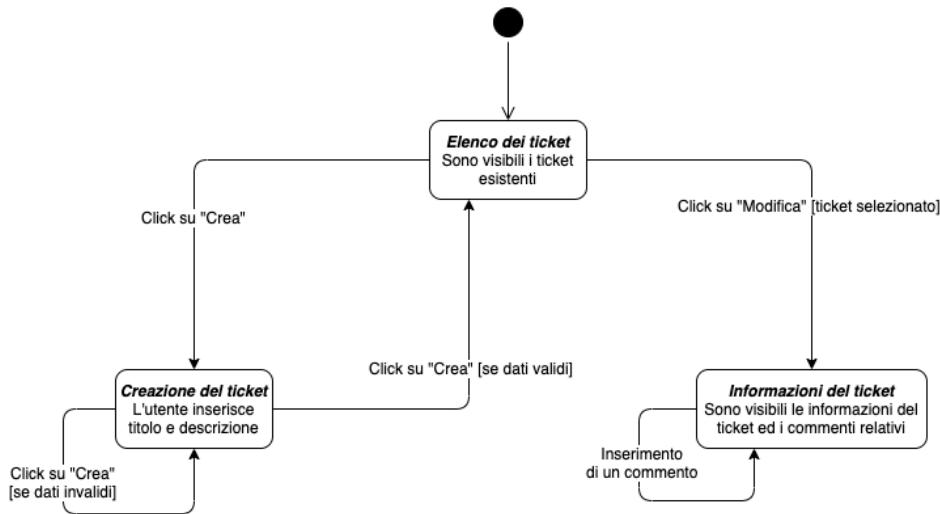


Figura 3.1: Relazioni tra le pagine dei ticket

- visualizzazione delle informazioni di un ticket, per operatori ed utenti loggati.

Queste tre pagine racchiudono l'intera UX e UI che permettono ad un utente di sapere quali ticket ha aperto, in che stato si trovano e come gli operatori stanno gestendo il problema.

Nella figura 3.1 sono visibili queste tre pagine e l'utente può navigare da una all'altra. È da ricordare che, su browser, in qualunque momento è possibile ritornare alla pagina precedente grazie alla navigazione integrata.

Di seguito sono presenti varie immagini che rappresentano le pagine implementate su front-end.

AI TICKETING

Numero	Titolo	Stato
<input type="checkbox"/> Numero 1	Problemi di login	Aperto
<input type="checkbox"/> Numero 2	App non installabile	Chiuso

Figura 3.2: Pagina di elenco ticket

AI TICKETING

Inserisci i dati del ticket

Titolo
[Text input field]

Contenuto
[Text input field]

CREA TICKET

Figura 3.3: Pagina di creazione ticket

AI TICKETING

Ticket >

Account >

Informazioni ticket 1

cl1 ↗

Informazioni del ticket

Numero	1
Titolo	Problemi di login
Contenuto	Non riesco a effettuare il login nel portale
Gruppo	Nessuno
Stato	Aperto

Commenti

Giovanni Rossi

Ci sono novità?

Inserisci un nuovo commento

Commenta qualcosa... COMMENTA

Figura 3.4: Le informazioni del ticket

AI TICKETING

Ticket >

Account >

Informazioni ticket 2

op1 ↗

Informazioni del ticket

Numero	2
Titolo	App non installabile
Contenuto	Non riesco a installare la app su Android
Gruppo	Nessuno
Stato	Chiuso

Commenti

Aperto

Assegnato

In corso

Chiuso

Inserisci un nuovo commento

Commenta qualcosa... COMMENTA

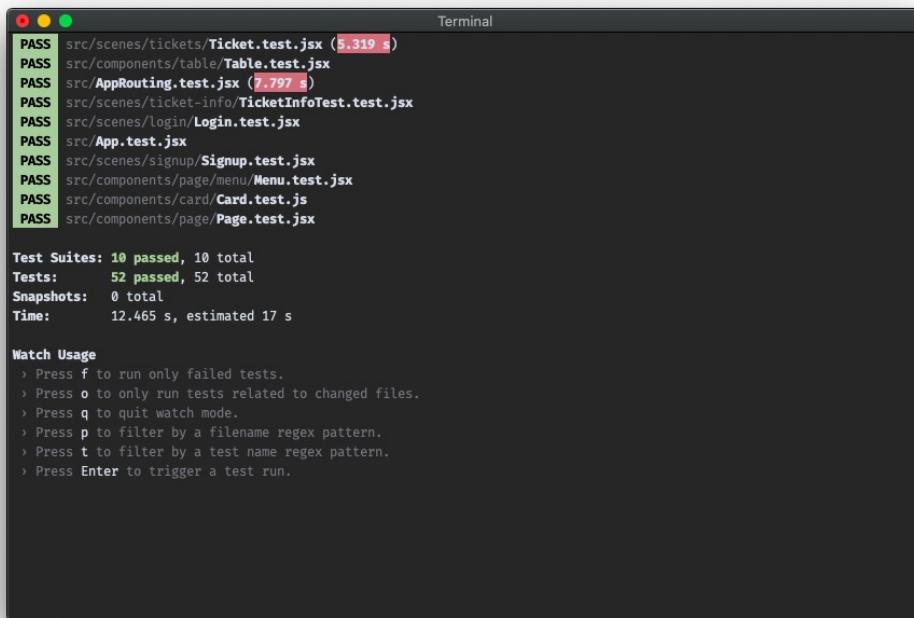
Figura 3.5: Un operatore cambia lo stato del ticket

3.3.2 Testing, coverage ed analisi statica

Alla fine di questa iterazione, *ESLint* non ha trovato alcun problema nel codice.

Come visibile in figura 3.6, i test del front-end consistono in questo istante dello sviluppo:

1. 10 suite di test;
2. 52 casi di test.



The screenshot shows a terminal window with the title "Terminal". The output of the Jest test runner is displayed, showing 10 test suites and 52 total tests passed. The terminal also includes a "Watch Usage" section with keyboard shortcuts for running tests.

```
PASS  src/scenes/tickets/Ticket.test.jsx (5.319 s)
PASS  src/components/table/Table.test.jsx
PASS  src/AppRouting.test.jsx (7.797 s)
PASS  src/scenes/ticket-info/TicketInfoTest.test.jsx
PASS  src/scenes/login/Login.test.jsx
PASS  src/App.test.jsx
PASS  src/scenes/signup/Signup.test.jsx
PASS  src/components/page/menu/Menu.test.jsx
PASS  src/components/card/Card.test.jsx
PASS  src/components/page/Page.test.jsx

Test Suites: 10 passed, 10 total
Tests:      52 passed, 52 total
Snapshots:  0 total
Time:       12.465 s, estimated 17 s

Watch Usage
> Press f to run only failed tests.
> Press o to only run tests related to changed files.
> Press q to quit watch mode.
> Press p to filter by a filename regex pattern.
> Press t to filter by a test name regex pattern.
> Press Enter to trigger a test run.
```

Figura 3.6: Unit testing front-end

La coverage ammonta ai risultati visibili nella tabella 3.2.

Metrica	Copertura percentuale
Istruzioni	84.07%
Branches	77.24%
Funzioni	82.14%
Righe	84.92%

Tabella 3.2: Coverage totale per il front-end

Capitolo 4

Iterazione 3

In questa iterazione abbiamo implementato il cuore algoritmico del progetto: l'assegnamento non supervisionato di un ticket ad un gruppo e la stima statistica del tempo di risoluzione.

4.1 Descrizione dell'algoritmo

L'algoritmo proposto si occupa dell'individuazione automatica di un gruppo di operatori a cui assegnare il ticket. Ciò è fatto tramite una variante dell'algoritmo *k-means*, usato in ambito di Machine Learning per la classificazione non supervisionata di un esemplare in diversi gruppi derivati dalle *feature* della popolazione.

I passi base di *k-means* sono:

1. Dato un esemplare da classificare e una popolazione già classificata, calcolare una *distanza* tra questo esemplare e gli altri esemplari della popolazione;
2. Prendere i k elementi con la distanza minore;
3. Assegnare all'esemplare il gruppo che compare più volte in questi k elementi;

4. Se non è possibile assegnare un gruppo o la distanza è troppo alta, allora l'elemento appartiene ad un nuovo gruppo.

La “distanza” è intesa in modo astratto, in quanto rappresenta solamente quanto due elementi sono vicini in base alle *feature* scelte, non necessariamente una distanza geografica.

Nel contesto in esame i gruppi sono creati a priori, poiché devono necessariamente riflettere l'organigramma aziendale.

Inoltre, il ciclo di vita di un ticket prevede che esso mantenga una forma di persistenza nel sistema per un tempo indefinito anche dopo la sua chiusura, aumentando monotonicamente il numero complessivo di essi nel corso del tempo. Per questo motivo, abbiamo deciso di ottimizzare l'algoritmo introducendo strutture dati di supporto sui gruppi che racchiudono una sintesi delle feature dei ticket appartenenti al gruppo stesso. Queste strutture dati sono poi aggiornate ad ogni assegnamento di un nuovo ticket.

Tale struttura è semplicemente un *dizionario* che usa delle parole come chiavi e dei numeri reali come valori. Le parole non sono tutte quelle della lingua italiana, ma devono soddisfare almeno uno dei seguenti criteri:

1. Essere lunghe almeno 5 caratteri;
2. Appartenere al dominio operativo dell'azienda.

Grazie a questa ottimizzazione è quindi possibile evitare di analizzare un numero elevato di ticket permettendo all'algoritmo di operare in una modalità *just in time*, al momento della creazione stessa del ticket. Così facendo l'integrazione con il codice già esistente risulta semplificata e non è necessario richiedere un'azione esterna al sistema.

I passi base del nostro algoritmo sono:

1. Dato un ticket da assegnare, analizzare il suo contenuto e confrontarlo con le informazioni derivate del gruppo;

2. Calcolare un *punteggio*, che è tanto maggiore quanto la possibilità che un ticket appartenga ad un gruppo;
3. Trovare il gruppo con il punteggio massimo;
4. Assegnare il ticket al gruppo e aggiornare la struttura dati aggiuntiva del gruppo relativo.

È presente un’ulteriore funzionalità: *la stima del tempo di risoluzione*. Dopo che un ticket viene assegnato ad un gruppo, è possibile basarsi sugli ultimi x ticket chiusi per poter stimare in quanto tempo il ticket verrà chiuso la prima volta. È stato considerato un numero finito di ticket sia per motivi di ottimizzazione della stima di un modello stocastico, sia per evitare che dati troppo vecchi, che potrebbero potenzialmente rispecchiare una diversa e meno performante struttura dell’azienda, vengano presi in influsso nel calcolo.

4.2 Flowchart

Nell’immagine 4.1 è visibile il flowchart che descrive la logica del nostro algoritmo principale, ovvero quello dell’assegnamento di un ticket ad un gruppo. Sono anche presenti dei riferimenti ad altri flowchart (i rettangoli con i doppi bordi), che sono definiti successivamente.

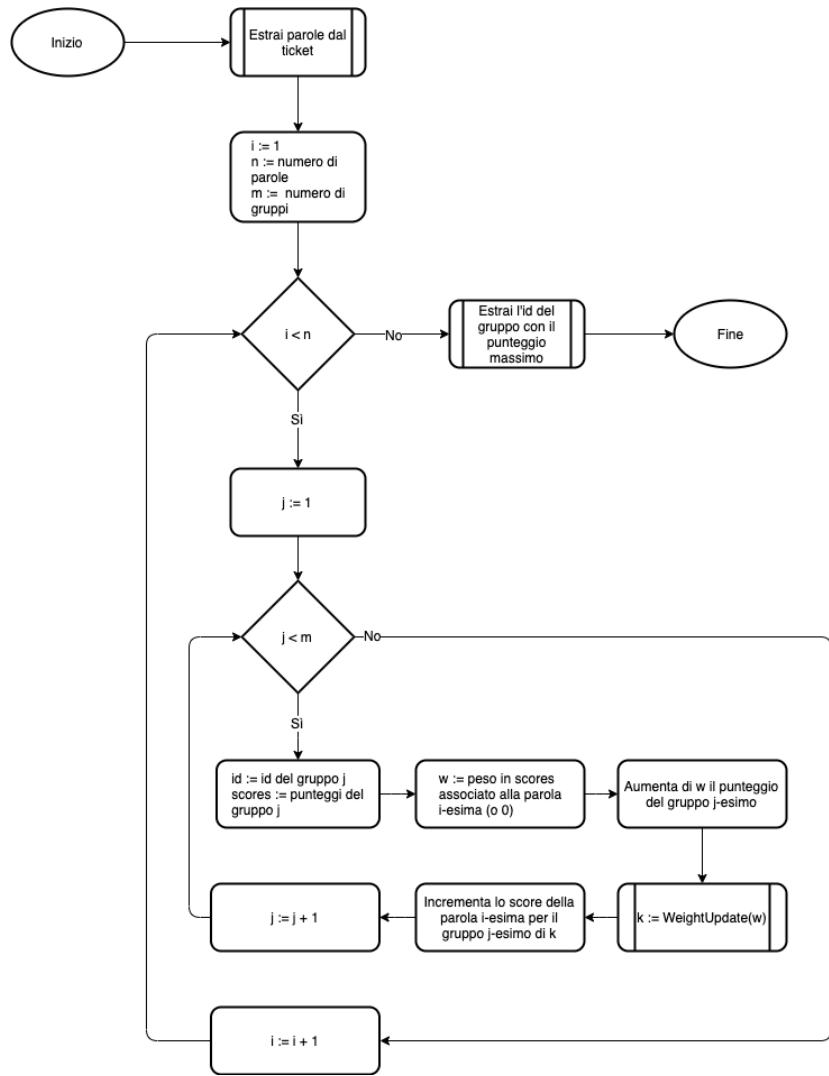


Figura 4.1: Flowchart di assegnamento di un ticket ad un gruppo

Il flowchart 4.2 rappresenta la logica di aggiornamento dei pesi:

- il peso viene aumentato in modo sostanziale se il suo valore è alto;
- il peso viene aumentato di poco se il suo valore è basso;
- il peso viene diminuito se è negativo.

Questo dà la possibilità di penalizzare alcune parole e dare più importanza ad altre nella ricerca di un gruppo adatto al ticket in esame.

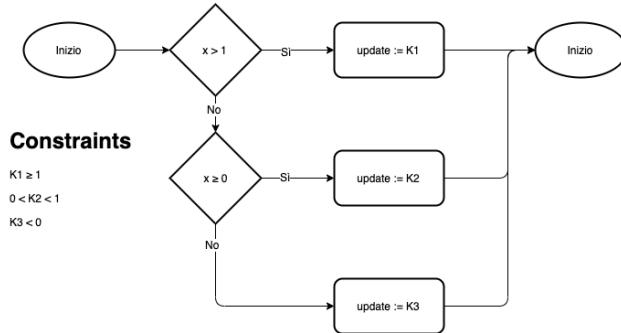


Figura 4.2: Flowchart per l'aggiornamento del peso di una parola in un gruppo

Infine, in figura 4.3 è presente l'algoritmo per la ricerca del massimo in un dizionario.

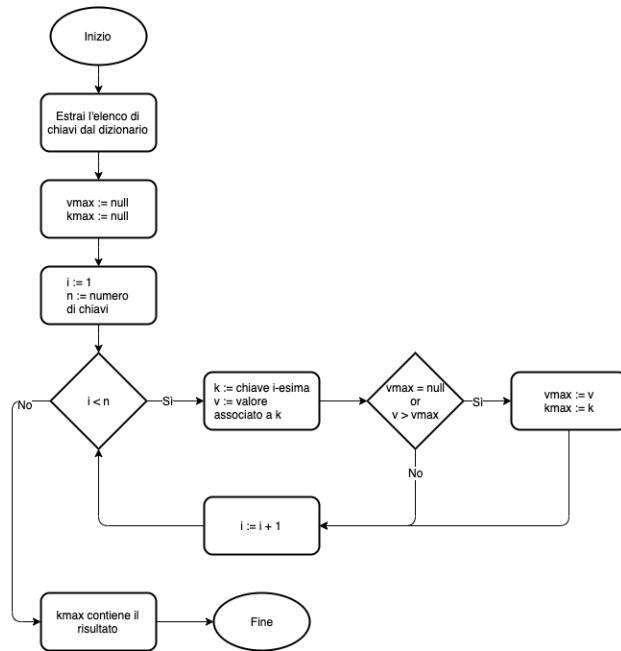


Figura 4.3: Flowchart per la ricerca della chiave con il valore massimo in un dizionario

4.3 Pseudocodice

Lo pseudocodice dell'algoritmo di assegnamento è l'algoritmo 1. In esso è possibile vedere alcune funzioni particolari:

- `ExtractWords`, usata per ottenere l'elenco delle parole dal titolo e dalla descrizione del ticket, rimuovendo la punteggiatura;
- `Push`, che aggiunge un elemento ad una lista;
- `Get` per ottenere il valore di una chiave da un dizionario;
- `Increment`, che aumenta un valore in un dizionario secondo una chiave, o lo inserisce se non esiste.

Nel codice è presente anche una soglia S , che rappresenta il valore minimo per cui si può considerare di aver trovato una vera appartenenza.

L'algoritmo 2 rappresenta invece come vengono aggiornati i pesi nei gruppi:

1. se il peso esistente è alto, significa che vogliamo associare la parola al gruppo, quindi viene aumentato;
2. se il peso esistente è piccolo, lo si aumenta di poco, per diminuire la possibilità di falsi positivi;
3. se il peso è invece negativo, allora si penalizza ancora di più.

In questo algoritmo, K_1 rappresenta un numero positivo maggiore di 1, K_2 è un valore positivo compreso tra 0 e 1 e K_3 è un numero negativo. Questi numeri saranno tarati su base di prove sperimentali.

All'inizio i pesi possono essere tutti nulli, oppure l'azienda ha la possibilità di impostarne alcuni manualmente. Ad esempio, per un ipotetico gruppo di operatori che si occupano di un'applicazione mobile, si potrebbe assegnare un peso alto alla parola “mobile” oppure “app”.

Algoritmo 1 Assegnamento di un ticket ad un gruppo

```
1: function ASSIGNGROUPTOTICKET(ticket, groups, S)
2:    $P \leftarrow \emptyset$                                  $\triangleright$  Dizionario vuoto
3:    $words \leftarrow \text{EXTRACTWORDS}(ticket)$ 
4:
5:   for all  $word \in words$  do
6:     for all  $(id, scores) \in groups$  do
7:       if  $word \in scores$  then
8:          $w \leftarrow \text{GET}(scores, word)$ 
9:       else
10:         $w \leftarrow 0$ 
11:      end if
12:      INCREMENT( $P, id, w$ )
13:       $k \leftarrow \text{WEIGHTUPDATE}(w)$ 
14:      INCREMENT( $scores, word, k$ )
15:    end for
16:  end for
17:
18:   $id \leftarrow \text{MAXINDICT}(P)$ 
19:   $v \leftarrow \text{GET}(P, id)$ 
20:  if  $v > S$  then
21:    return  $(id, groups)$ 
22:  else
23:    return  $(\text{null}, groups)$ 
24:  end if
25: end function
```

Algoritmo 2 Calcolo del termine di aggiornamento di un peso

```
1: function WEIGHTUPDATE( $x$ )
2:   if  $x > 1$  then
3:     return  $K_1$ 
4:   else if  $x \geq 0$  then
5:     return  $K_2$ 
6:   else
7:     return  $K_3$ 
8:   end if
9: end function
```

L'algoritmo 3 è invece la ricerca della chiave con il valore massimo in un dizionario. Abbiamo supposto che sia possibile accedere all'elenco delle chiavi di un dizionario per poterle analizzare una ad una.

Algoritmo 3 Ottenimento della chiave con il valore massimo in un dizionario

```
1: function MAXINDICT( $h$ )
2:    $v_{max} \leftarrow \text{null}$ 
3:    $k_{max} \leftarrow \text{null}$ 
4:   for all  $k \in h$  do
5:      $v \leftarrow \text{GET}(h, k)$ 
6:     if  $v_{max} = \text{null} \vee v > v_{max}$  then
7:        $v_{max} \leftarrow v$ 
8:        $k_{max} \leftarrow k$ 
9:     end if
10:   end for
11:   return  $k_{max}$ 
12: end function
```

Infine, l'algoritmo 4 racchiude i passi base per la stima del tempo di risoluzione di un ticket. Dato un elenco di ticket passati, l'algoritmo calcola prima tutte gli intervalli di tempo per cui dei ticket sono stati aperti. A questo punto, viene stimato un modello ARIMA su queste differenze e poi, grazie ai coefficienti trovati, viene effettuata una predizione.

Algoritmo 4 Stima tempo di risoluzione

```
1: function ESTIMATETIME(tickets)
2:   times  $\leftarrow \emptyset$                                       $\triangleright$  Lista vuota
3:   for all t  $\in$  tickets do
4:     if STATO(t) = ‘Chiuso’ then
5:        $\Delta t \leftarrow \text{FINE}(t) - \text{INIZIO}(t)$ 
6:       PUSH(tickets,  $\Delta t$ )
7:     end if
8:   end for
9:   coefs  $\leftarrow$  FITARIMA(times)
10:  p  $\leftarrow$  PREDICT(coefs, times)
11:  return p
12: end function
```

4.4 Analisi di complessità

4.4.1 Classificazione ticket

Per l’assegnazione del ticket in un gruppo, è necessario prendere in considerazione la dimensione degli input.

Poniamo quindi n il numero di parole contenute nel titolo e nella descrizione del ticket e m il numero di gruppi pre-esistenti.

Possiamo anche supporre che k sia il numero di caratteri complessivi del ticket (compresi sia nel titolo sia nella descrizione). In questo caso l'**estrazione delle parole** si compone di due fasi:

1. la suddivisione del testo in parole con rimozione dei caratteri non alfabetici;
2. la conversione in minuscolo di ogni parola.

Integralmente la stringa composta dall’accostamento di titolo e descrizione dev’essere attraversata due volte: la prima per l’esecuzione dell’espressione

regolare e la seconda per la conversione in minuscolo. Di conseguenza la complessità dei passi eseguiti per l'ottenimento delle parole risulta essere

$$O(2k) = O(k)$$

Inoltre, poiché $k \propto n$, si ha che la complessità può essere espressa in termini sia di k sia di n , tenendo conto che, in un numero significativo dei casi, $k \gg n$. Noi la esprimeremo in funzione di n .

Il prossimo passo nell'algoritmo consiste nei due **cicli annidati**. Si ha che

- il ciclo più esterno viene eseguito esattamente n volte, una volta per parola;
- il ciclo più interno viene eseguito m volte per ogni parola.

Bisogna quindi stimare il corpo del ciclo interno. Supponendo di usare una struttura dati dizionario con tempi di accesso, aggiunta, modifica e lookup delle *chiavi* pari a $O(1)$, è facile notare che:

- la funzione `Increment` è $O(1)$ per via dell'uso di questa struttura;
- la funzione `WeightUpdate` è $O(1)$ poiché non dipende in alcun modo dalla dimensione dati in input.

Ne consegue quindi che il corpo del ciclo interno è a sua volta $O(1)$. Quindi il tempo dei due cicli annidati è complessivamente

$$\Theta(nm)$$

Infine, la **ricerca del gruppo con il punteggio massimo**. Come è possibile intuire dallo pseudocodice nell'algoritmo 3, bisogna analizzare ogni gruppo una volta. Il ciclo viene quindi eseguito m volte, e con un corpo con complessità $O(1)$ risulta che quest'ultima operazione ha complessità

$$\Theta(m)$$

Le ultime azioni (righe 19-24) hanno invece complessità costante.

Quindi, per riassumere

$$\begin{cases} O(n) & \text{separazione delle parole} \\ \Theta(nm) & \text{costruzione dei punteggi} \implies O(n + nm + m) \text{ complessità totale} \\ \Theta(m) & \text{ricerca del massimo} \end{cases}$$

4.4.2 Stima ARIMA del tempo di esecuzione

La complessità di questo algoritmo verrà calcolata in base al numero n di ticket usati per la stima.

Il **calcolo dei tempi** in cui ogni ticket è stato aperto ha complessità $O(1)$ e viene eseguito esattamente n volte. Inoltre, si suppone che l'ottenimento della data di apertura e della data di chiusura di un ticket siano operazioni $O(1)$. Di conseguenza, il calcolo iniziale dei tempi risulta essere

$$\Theta(n)$$

La **stima dei coefficienti** è il prossimo passo. Seppur lasciata ad una libreria esterna, questa utilizza il metodo della *MLE* tramite *innovazioni* con un algoritmo di *gradient descent*. Questo tipo di algoritmo dipende da due dati: il numero n di dati in ingresso, dei quali sarà calcolato il gradiente, ed un numero p di regressori usati. Inoltre, questo tipo di algoritmo ha prestazioni che possono variare fortemente in base al problema preso in considerazione.

Per i nostri scopi, è lecito supporre che $p \ll n$ e che l'operazione di gradiente sia dipendente linearmente da n , quindi la complessità risulta

$$O(n)$$

Infine, la **predizione** è semplicemente una combinazione lineare tra i p coefficienti stimati e gli ultimi p tempi di risoluzione. Poiché abbiamo supposto $p \ll n$, questa operazione è effettivamente $O(1)$, ma anche togliendo questa supposizione si ha complessità pari a $\Theta(n)$ nel caso peggiore.

In conclusione

$$\begin{cases} \Theta(n) & \text{calcolo dei tempi} \\ O(n) & \text{stima} \\ O(1) & \text{predizione} \end{cases} \implies O(n) \text{ complessità totale}$$

Capitolo 5

Iterazione 4

5.1 Continuous Integration

A questo punto dello sviluppo, avendo creato un MVP funzionante, è stato deciso di apportare miglioramenti all'aspetto infrastrutturale, utilizzando le idee del paradigma *Continuous Integration*.

Tale miglioramento è stato implementato utilizzando *Travis*, un software in grado di eseguire script e segnalare lo stato di terminazione di tali istruzioni. Essi possono essere utilizzati per automatizzare l'esecuzione dei test ogni qualvolta viene aggiunto del codice al repository git di riferimento.

In questo modo abbiamo semplificato di molto il ciclo di sviluppo e modifica del codice, garantendo di avere sempre l'esecuzione dei test senza doverli avviare manualmente.

5.2 Electron

Electron è una libreria JavaScript utilizzata per la creazione di applicazioni cross-platform tramite i linguaggi che si trovano tipicamente in ambito Web, ossia JavaScript, HTML e CSS. È usata in molti progetti, come ad esempio

Visual Studio Code, Twitch desktop, Microsoft Teams, Postman e svariate altre¹.

5.2.1 Integrazione con il front-end

L'integrazione con il codice front-end è stata relativamente facile. È stato creato un file (`main.js`) che si occupasse della gestione delle finestre dell'applicazione tramite le API di Electron.

Inoltre, è necessario indicare quali file considerare nella creazione dei vari *artifact* prodotti da Electron: abbiamo deciso di includere i file già compressi e ottimizzati per il web, in modo da accelerare le performance della applicazione.

Infine, sono state anche presi in considerazione i diversi comportamenti standard che i principali sistemi operativi (macOS, Linux e Windows) definiscono per i propri programmi. Ad esempio, se il progetto viene compilato con Electron per macOS, allora la chiusura della finestra non determinerà la chiusura dell'applicazione, a differenza di quanto accade se compilato per Linux o Windows, come si può vedere nel seguente estratto del file `main.js`².

```
app.whenReady().then(createWindow);

app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit();
  }
});

app.on('activate', () => {
  if (BrowserWindow.getAllWindows().length === 0) {
    createWindow();
  }
}
```

¹<https://www.electronjs.org/apps>

²Il sistema macOS è identificato dalla piattaforma 'darwin'

```
});
```

In ogni caso, oltre agli eseguibili ed alle dovute risorse, vengono compilati anche i vari installer per ogni piattaforma, così da garantire una maggiore facilità di installazione e condivisione.

5.2.2 Gestione dei cookies

L'unico problema incontrano nell'integrazione con Electron è stato nella gestione dei cookies. Per motivi di sicurezza, i cookie sicuri non vengono salvati sul protocollo `file`, che è quello usato da Electron per la gestione dei file HTML.

Quindi, per garantire il corretto funzionamento, è stato implementato un piccolo modulo che si occupa di intercettare le richieste di rete fatte da e verso il nostro server e di aggiungere in modo trasparente i cookie scambiati.

Questi cookie esistono solamente in memoria³ e non sono mai inviati a domini diversi da quello a cui risiede il back-end, garantendo così un'adeguata sicurezza.

³Sono quindi persi alla chiusura dell'applicazione

Capitolo 6

Toolchain

Questo capitolo comprende tutte le scelte di librerie, framework e piattaforme che sono state effettuate per l'implementazione del progetto.

6.1 Modellazione

Per la modellazione sono stati usati i seguenti tool:

- **Diagrammi UML di componenti, classi, architettura e deployment:** *Draw.io*, uno strumento sia online sia offline open source per la realizzazione di diagrammi generici, che comprende anche strumenti appositi per la modellazione UML;
- **Diagramma dell'implementazione delle classi:** *pyreverse*, un tool in Python in grado di estrarre diagrammi rappresentanti le relazioni tra delle classi pre-esistenti in un progetto.

6.2 Linguaggi e librerie

6.2.1 Back-end

Per l'implementazione del back-end sono stati usati:

- **Python** come linguaggio di programmazione, scelto per la sua semplicità e portabilità;
- **PIP** come package manager per Python;
- **Django**, un framework Python per la realizzazione di server, gestione di database e templating;
- **SQLite3** come *DBMS*, un dialetto SQL che offre un'ottima integrazione con Python; pur non essendo il più performante, è stato scelto per rapidità di implementazione, e in iterazioni future è possibile scambiarlo con altri DBMS più performanti;
- **Pylint** per l'*analisi statica* del codice Python, l'identificazione real-time di errori, warning e la fornitura di ottimizzazioni e suggerimenti;
- **Django Test Framework** e **Coverage.py** per l'*analisi dinamica* del codice Python, dei quali il primo è integrato in Django, mentre il secondo è uno strumento esterno per misurare la copertura del codice che fornisce anche un output interattivo in HTML.

6.2.2 Front-end

Per l'implementazione del front-end sono stati usati:

- **JavaScript** come linguaggio di programmazione;
- **HTML e CSS** come linguaggi di markup e styling delle pagine web;

- Il framework **Bootstrap**, che comprende svariati stili di default per velocizzare lo sviluppo di una UI coerente e reattiva;
- **React.js** come libreria per la gestione di componenti riusabili ed ottimizzati grazie all'implementazione di un *virtual DOM* per il dispatching intelligente di aggiornamenti grafici della pagina web e che offre anche la gestione dello stato e dei dati usati dalla Single Page Application;
- **Axios** come libreria JavaScript per effettuare delle richieste AJAX al back-end;
- **Moment**, libreria JavaScript per la gestione di istanti di tempo, giorni, mesi ed intervalli di tempo;
- **Ag-grid** per le tavole e le griglie, con ottimizzazioni quali la virtualizzazione di righe e colonne, la selezione di righe e la ricerca rapida;
- **Lodash**, una libreria JavaScript che fornisce un set di funzioni di utilità base comunemente usate;
- **Jest**, un framework per il *testing* e l'*analisi dinamica* del codice, soprattutto con la libreria React;
- **Testing Library**, una libreria che fornisce alcune utilità per il testing;
- **ESLint**, una libreria per l'*analisi statica* del codice, che fornisce anche suggerimenti, rafforza degli standard stilistici e corregge eventuali violazioni di regole custom;
- **Node.js** come JavaScript runtime environment per la compilazione della Single Page Application;
- **NPM** e **Yarn** come package manager per l'ambiente Node, usati per l'installazione delle varie dipendenze;
- **Electron**, una libreria usata per poter compilare il codice JavaScript, HTML e CSS in una app cross-platform.

6.3 Infrastruttura

Per le varie componenti dell'infrastruttura sono stati usati:

- **Travis**, un tool per la *Continuous Integration* ed il testing automatizzato.

6.4 Documentazione

La documentazione è stata scritta in L^AT_EX, compilata tramite il tool `lutexmk`.

6.5 Versioning

Il sistema di versioning utilizzato è **Git**, un sistema di versioning distribuito, open source e gratuito.

Il codice del progetto è presente anche su **GitHub**, un sito per remote hosting di progetti git.

6.6 Ambienti e IDEs

Lo strumento principale utilizzato è **Visual Studio Code**, un editor di testo open source e gratuito realizzato per i sistemi operativi *Linux*, *macOS* e *Windows*.

Esso non è un IDE, in quanto non offre funzionalità di build automation e debugging, ma si basa invece su un sistema di *estensioni*, sia per syntax highlighting, linting, testing e building, mentre tutti gli strumenti per il funzionamento dei linguaggi sono locali e forniti dal sistema operativo.

Fra le molte funzionalità, è presente anche la *condivisione live del codice*, in modo da poterne discutere in real-time e poter usare tecniche di sviluppo agili che richiedono più persone, come il *pair programming*, anche a distanza.

Come esempio, nella figura 6.1 è visibile l'editor Visual Studio Code, che permette anche di gestire diversi progetti, scritti in diversi linguaggi e con diverse tecnologie, senza dover cambiare editor o IDE.

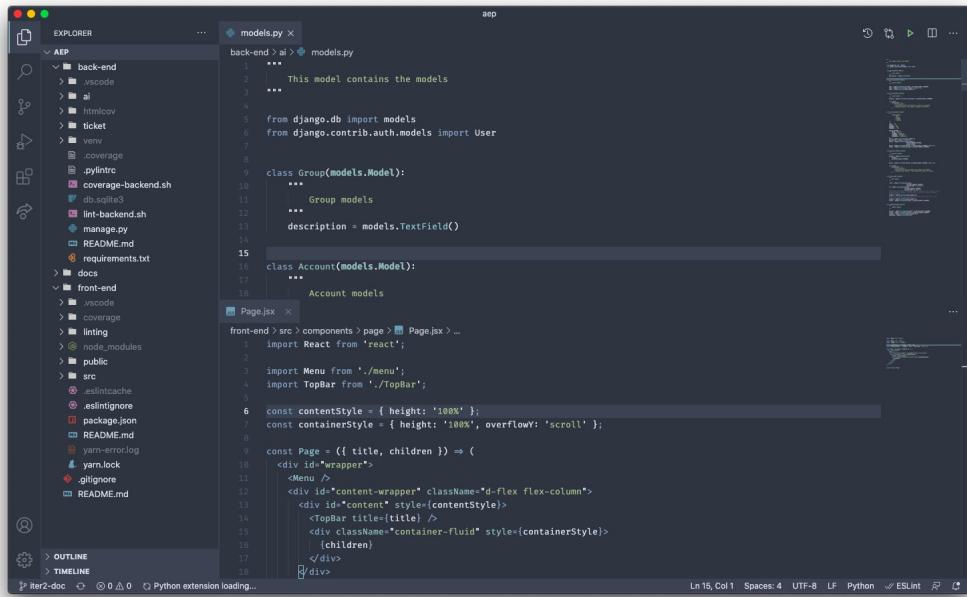
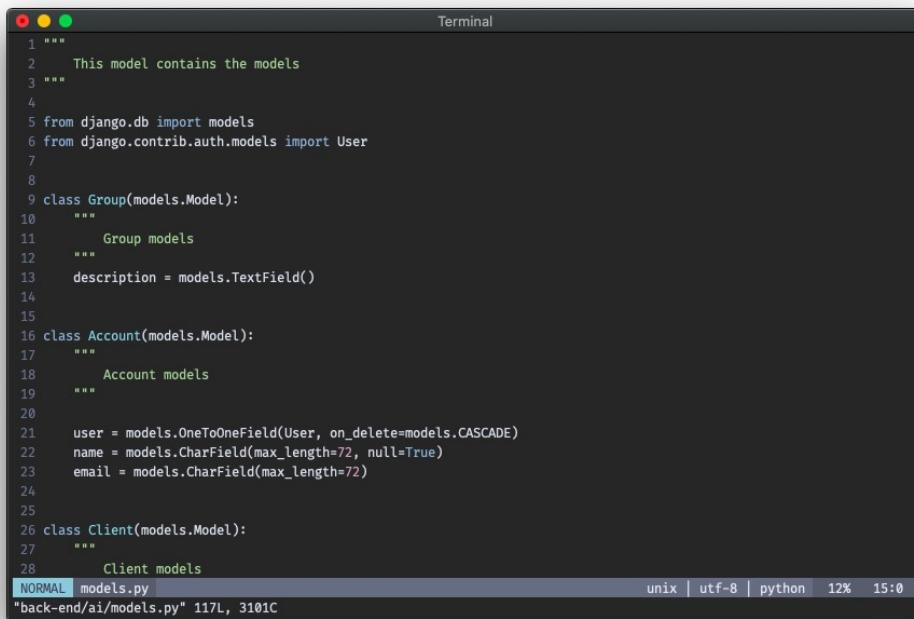


Figura 6.1: Un esempio dell'uso di Visual Studio Code

È anche stato usato **VIM**, un editor di testo scritto originariamente per Unix per poter essere usato da riga di comando, in quei casi in cui era necessario modificare rapidamente un file senza aprire necessariamente un editor complesso come Visual Studio Code. Un esempio è visibile in figura 6.2.



The screenshot shows a terminal window titled "Terminal" displaying Python code for Django models. The code defines three classes: Group, Account, and Client, each with its own docstring and fields. The "models.py" file has 117 lines and 3101 characters. The status bar at the bottom right indicates the file is in NORMAL mode, the encoding is utf-8, it's running under python, and the current time is 15:0.

```
1 """
2     This model contains the models
3 """
4
5 from django.db import models
6 from django.contrib.auth.models import User
7
8
9 class Group(models.Model):
10     """
11         Group models
12     """
13     description = models.TextField()
14
15
16 class Account(models.Model):
17     """
18         Account models
19     """
20
21     user = models.OneToOneField(User, on_delete=models.CASCADE)
22     name = models.CharField(max_length=72, null=True)
23     email = models.CharField(max_length=72)
24
25
26 class Client(models.Model):
27     """
28         Client models
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
```

Figura 6.2: Esempio di uso di VIM

Bibliografia

- [1] *React, a JavaScript library for users interface.* URL: <https://reactjs.org>.
- [2] *Jest, Delightful JavaScript Testing.* URL: <https://jestjs.io>.
- [3] *ESLint, pluggable JavaScript linter.* URL: <https://eslint.org>.
- [4] *Electron, Build cross-platform desktop apps with JavaScript, HTML and CSS.* URL: <https://www.electronjs.org>.
- [5] *Yarn, a Node package manager.* URL: <https://yarnpkg.com>.
- [6] *The web framework for perfectionists with deadlines.* URL: <https://www.djangoproject.com/>.
- [7] *Travis, Test and Deploy with Confidence.* URL: <https://travis-ci.org>.
- [8] Mauro Pezzè e Michal Young. *Software testing and analysis: process, principles, and techniques.* OCLC: ocn123408179. Hoboken, N.J.: Wiley, 2008. ISBN: 9780471455936.
- [9] William S Vincent. *Django for APIs: Build web APIs with Python and Django.* English. OCLC: 1120851182. 2018. ISBN: 9781093633948.
- [10] Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* English. OCLC: 1057907478. 2016. ISBN: 9780136083221 9780132350884.
- [11] Harry Percival. *Test-driven development with Python: obey the testing goat: using Django, Selenium, and JavaScript.* Second edition. OCLC: ocn953432202. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 9781491958704.