

# 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

<b>1</b>	<b>Iterazione 0</b>	<b>1</b>
1.1	Introduzione . . . . .	1
1.2	Analisi dei requisiti . . . . .	1
1.2.1	Analisi del contesto . . . . .	1
1.2.2	Studio di fattibilità . . . . .	2
1.3	Specifiche . . . . .	2
1.3.1	Casi d'uso . . . . .	2
1.3.2	Requisiti funzionali . . . . .	9
1.4	Architettura software . . . . .	11
1.4.1	Component Diagram . . . . .	11
1.4.2	Class Diagram . . . . .	12
1.5	Architettura hardware . . . . .	13
1.5.1	Architettura hardware . . . . .	14
1.5.2	Deployment Diagram . . . . .	14
<b>2</b>	<b>Iterazione 1</b>	<b>16</b>
2.1	Introduzione . . . . .	16
2.2	Back-end e REST API . . . . .	17
2.2.1	Autenticazione . . . . .	17

2.2.2	App AI . . . . .	18
2.2.3	Console di amministrazione . . . . .	20
2.2.4	API endpoints . . . . .	20
2.3	Front-end . . . . .	21
2.3.1	Struttura SPA . . . . .	21
2.3.2	Pagine visitabili . . . . .	22
2.4	Analisi statica . . . . .	23
2.4.1	Back-end . . . . .	23
2.4.2	Front-end . . . . .	23
2.5	Analisi dinamica . . . . .	24
2.5.1	Testing . . . . .	24
2.5.2	Coverage . . . . .	28
<b>Bibliografia</b>		<b>31</b>

# 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;

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

- *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.

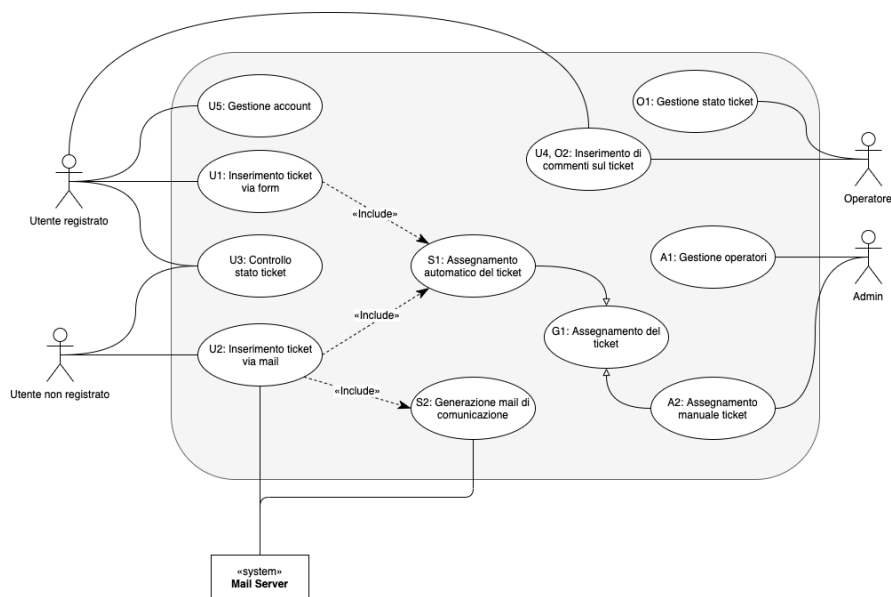


Figura 1.1: Diagramma UML dei casi d'uso

## 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.

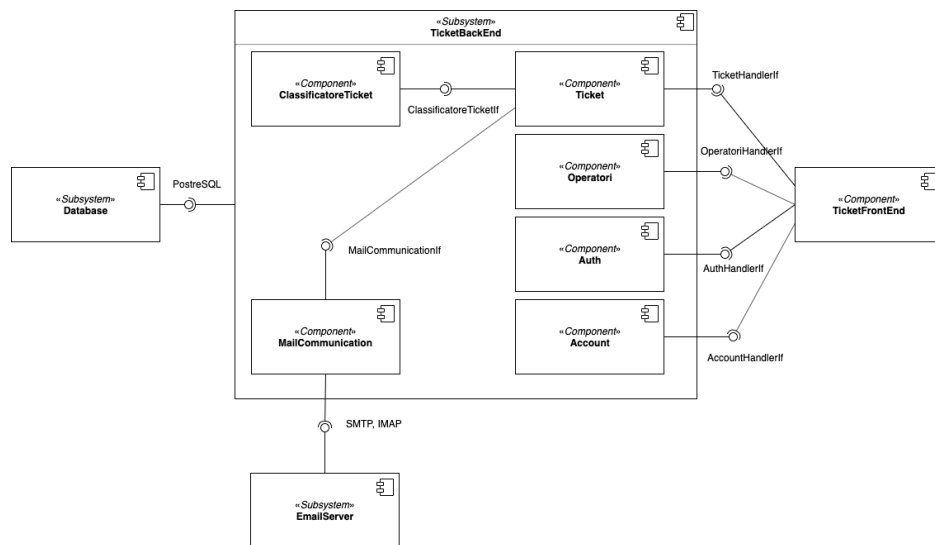


Figura 1.2: Diagramma dei componenti

## 1.4.2 Class Diagram

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

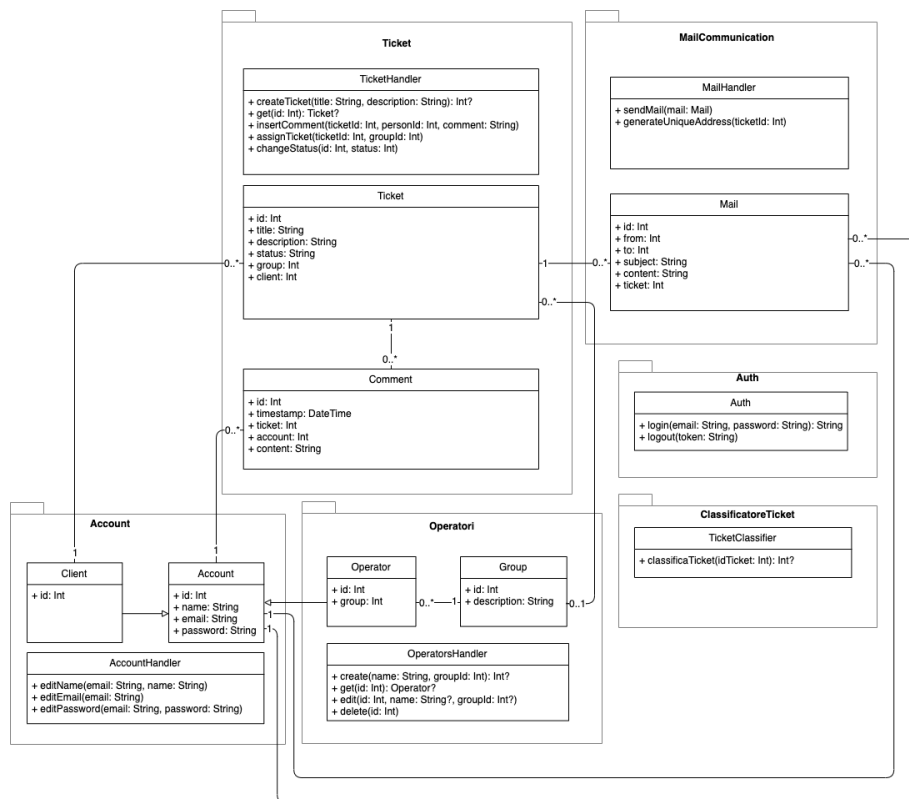


Figura 1.3: Diagramma delle classi

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 architetturale del sistema.

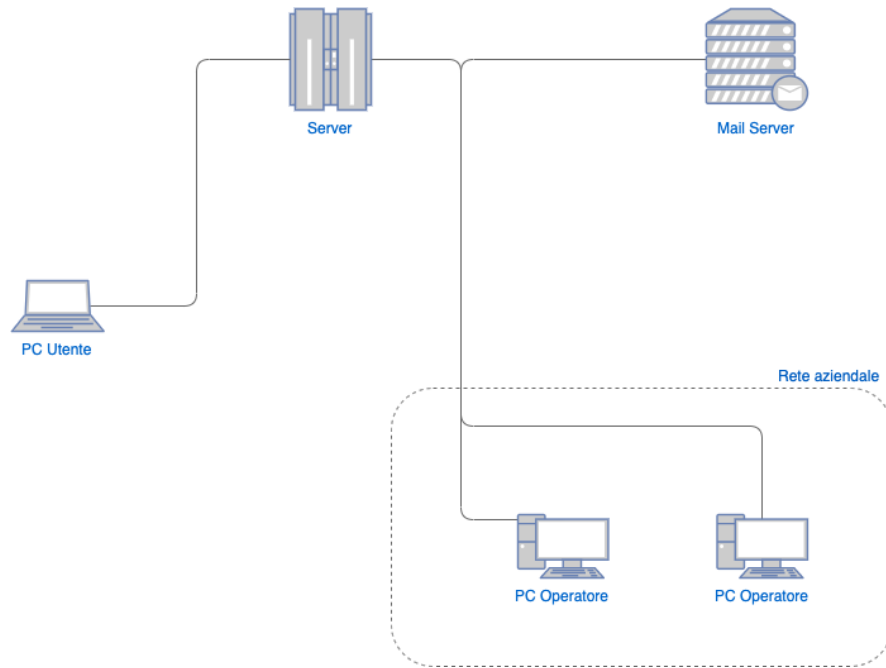


Figura 1.4: Architettura hardware

### 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.

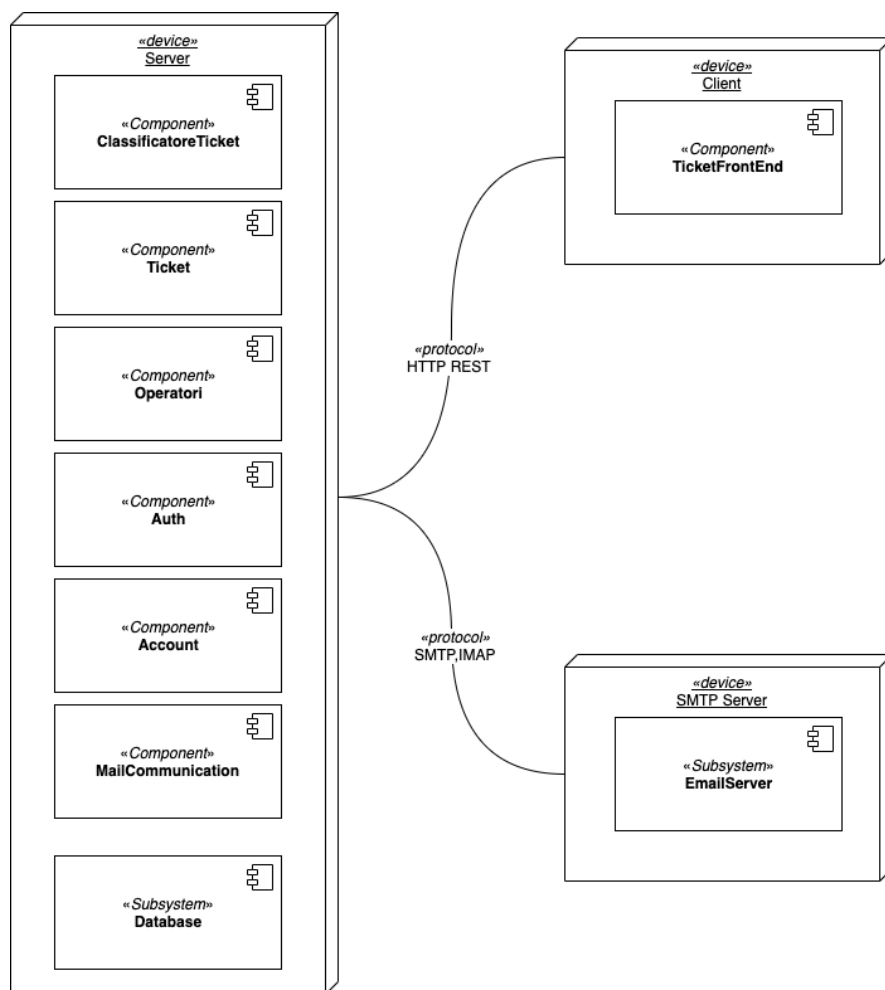


Figura 1.5: Diagramma di deployment

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 della gestione completa<sup>1</sup> di un server in Python e della creazione e mantenimento di API Web.

### 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 ereditarietà<sup>2</sup>.

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

---

<sup>1</sup>Comprendente accesso a database, templating, autorizzazioni, etc.

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

```

@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.

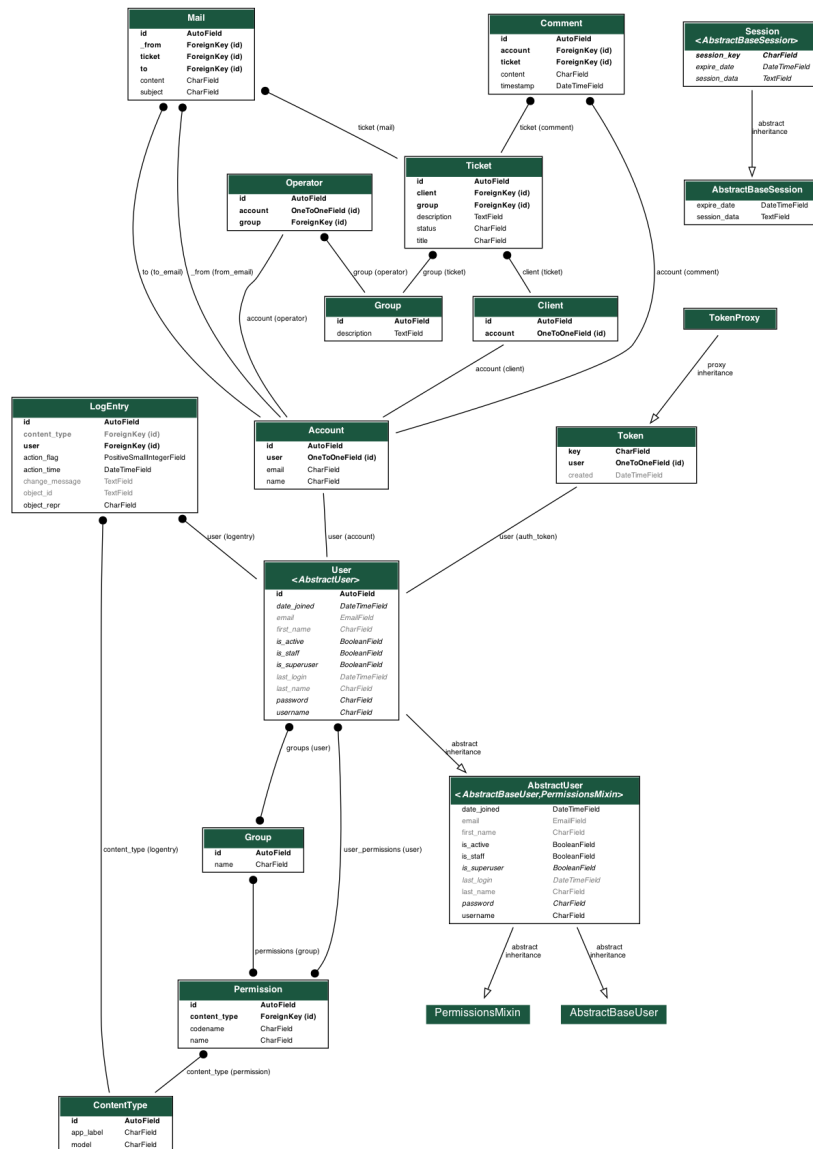


Figura 2.1: Relazioni tra classi di AI

### 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/`.

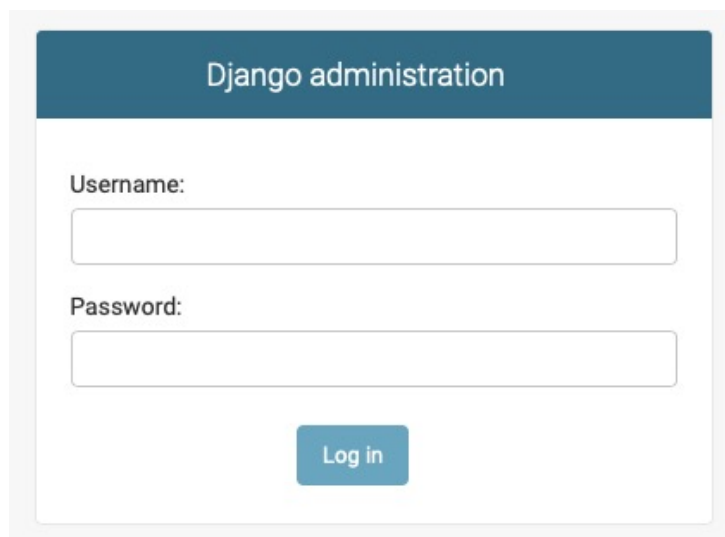


Figura 2.2: Admin console di Django

### 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 cartelle<sup>3</sup>, 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;
- `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.

---

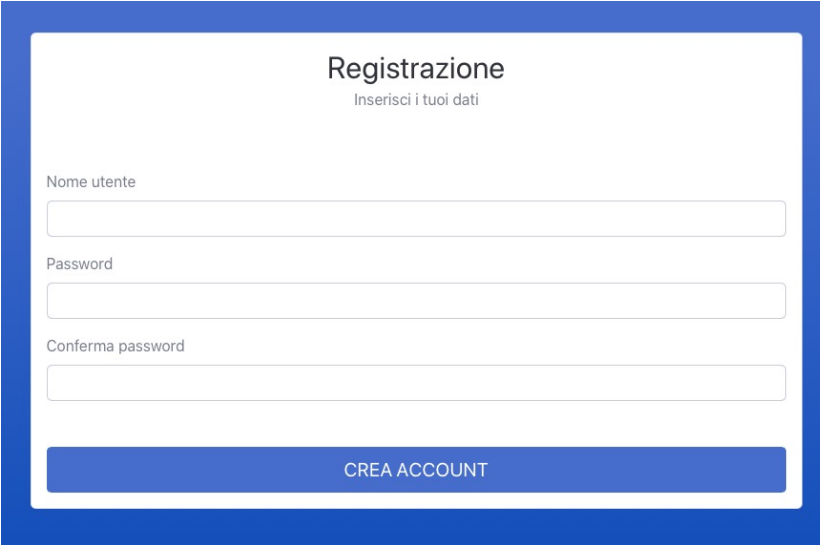
<sup>3</sup>La struttura del progetto non è data da un framework, ma da esperienza personale.

### 2.3.2 Pagine visitabili

Le pagine visitabili sono<sup>4</sup>

- 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.



The image shows a user registration form titled "Registrazione" with the subtitle "Inserisci i tuoi dati". The form is enclosed in a blue border. It contains three input fields: "Nome utente", "Password", and "Conferma password". Below these fields is a blue button labeled "CREA ACCOUNT".

Figura 2.3: Pagina di registrazione

---

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

## 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%
<b>Total</b>	<b>392</b>	<b>33</b>	<b>0</b>	<b>92%</b>

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

File		Statements	Branches	Functions	Lines	
src	<div><div></div></div>	86.96%	20/23	83.33%	5/6	100%
src/components/page	<div><div></div></div>	87.5%	14/16	100%	2/2	80%
src/components/page/menu	<div><div></div></div>	100%	25/25	87.5%	7/8	100%
src/scenes	<div><div></div></div>	100%	3/3	100%	0/0	100%
src/scenes/account	<div><div></div></div>	100%	2/2	100%	0/0	100%
src/scenes/home	<div><div></div></div>	100%	2/2	100%	0/0	100%
src/scenes/login	<div><div></div></div>	88.1%	37/42	87.5%	14/16	100%
src/scenes/signup	<div><div></div></div>	90.63%	58/64	83.87%	26/31	100%
src/scenes/ticket-info	<div><div></div></div>	100%	3/3	100%	0/0	100%
src/scenes/ticket-list	<div><div></div></div>	100%	2/2	100%	0/0	100%
src/scenes/ticket-new	<div><div></div></div>	100%	2/2	100%	0/0	100%

Figura 2.5: Front-end code coverage

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

# 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] *Yarn, a Node package manager*. URL: <https://yarnpkg.com>.
- [5] *The web framework for perfectionists with deadlines*. URL: <https://www.djangoproject.com/>.
- [6] Mauro Pezzè e Michal Young. *Software testing and analysis: process, principles, and techniques*. OCLC: ocn123408179. Hoboken, N.J.: Wiley, 2008. ISBN: 9780471455936.
- [7] William S Vincent. *Django for APIs: Build web APIs with Python and Django*. English. OCLC: 1120851182. 2018. ISBN: 9781093633948.
- [8] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. English. OCLC: 1057907478. 2016. ISBN: 9780136083221 9780132350884.
- [9] 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.