Progetto aste:

Gruppo di lavoro:
Tassi Andrea 830541
Grassi Marco 829664
Frigati Francesco 829817
Villa Fabio 829583

Link Github: https://github.com/UnimibSoftEngCourse1920/progetto-aste-2-gruppo-aste-2
Link Travis: https://travis-ci.com/github/UnimibSoftEngCourse1920/progetto-aste-2-gruppo-aste-2
Link Sonarcloud: https://sonarcloud.io/dashboard?id=UnimibSoftEngCourse1920_progetto-aste-2-gruppo-aste-2

Guida all installazione:

Per il corretto funzionamento dell' applicativo è necessario scaricare ed installare PostgreSql.
In alternativa è possibile scaricare l'immagine docker di PostgreSql.
Di default il backend dell' applicativo si aspetta di trovare un db chiamato "databaseaste" installato localmente sulla porta 5433. La porta di default di PostgreSql è la 5432 per cui se già installato è possibile modificare il file Backend/ProgettoAste2/src/main/resources/application.yml specificando la porta 5432, dovendo poi ricreare il file jar tramite Maven.
Una volta installato PostgreSql è necessario importare il file sql del database che si trova sotto Backend/Database. In questa cartella sono presenti due file, dumpTutto.sql e dumpTutto-WIN7.sql. Il secondo file è fornito perchè abbiamo rilevato problemi di compatibilità con Windows 7.
Da terminale (o direttamente da psql se sotto Windows) la procedura è:
- psql -h localhost -p "numeroPorta, per esempio 5433" -U "username, di default postgres con password postgres"
- \i path al file sql, se sotto Windows ricordarsi di invertire il senso degli /
- Il db dovrebbe ora essere stato importato, per controllare provare a collegarsi al db con \c databaseaste

Nel database è già incluso un amministratore con username admin, email admin@gmail.com e password admin. Inoltre è presente una configurazione di default per le aste, se per testare le funzionalità del sistema se ne aggiungesse un altra si consiglia di impostare una durata e numero di timeslot relativamente bassi.

Per la parte di backend è necessario solo eseguire il file jar, assicurarsi di avere installato jre 11 il comando è java -jar ProgettoAste-0.0.1-SNAPSHOT.jar. Il backend dovrebbe essere accessibile all indirizzo localhost:8080.
Il file jar è disponibile nella directory Backend/ProgettoAste2/target/ProgettoAste-0.0.1-SNAPSHOT.jar

Per la parte di front end invece è necessario installare npm. Se sotto Linux è sicuramente presente nel proprio package manager.

Una volta installato npm gli step sono:
- da terminale muoversi nella cartella FrontEnd/asteweb
- `npm install -g @vue/cli`

- npm install vue bootstrap-vue bootstrap
- npm run serve
- il sito dovrebbe partire ed essere accessibile all indirizzo localhost:3242

Per il corretto funzionamento assicurarsi che il database sia attivo e che il backend sia attivo prima del frontend.

Strumenti utilizzati:

Travis CI:

Come consigliato abbiamo usato Travis CI come servizio di continuous integration, questo servizio è stato decisamente molto utile per capire dove e dopo quali modifiche al codice avevamo introdotto problemi che impedivano la build del progetto e/o fallimenti di test.

Sonarcloud:

Indispensabile soprattuto ad inizio progetto per “adattarsi” a stili di scrittura del codice standard e per rimediare ad errori di distrazione.

Draw.io:

Utilizzato per la stesura di tutti I diagrammi di progetto.

Understand:

Usato per ottenere informazioni sulle metriche del codice e per verificare la presenza di Antipattern.

Spring Boot:

La scelta del framework da utilizzare per il backend è ricaduta su Spring Boot per diversi motivi, primo su tutti il fatto che utilizza Java come linguaggio (unico linguaggio che tutti I componenti del gruppo conoscevano). Altri motivi includono la relativa semplicità e chiarezza di utilizzo e la enorme quantità di materiale informativo disponibile sul Web.

Vue.js:

Framework Javascript utilizzato per il frontend, è stato scelto perchè viene ritenuto uno dei framework più semplici per Javascript. Vista la quasi totale inesperienza con il linguaggio il risultato ottenuto risulta essere soddisfacente (se pur non perfetto).

Documentazione:

Diagramma casi d’uso:

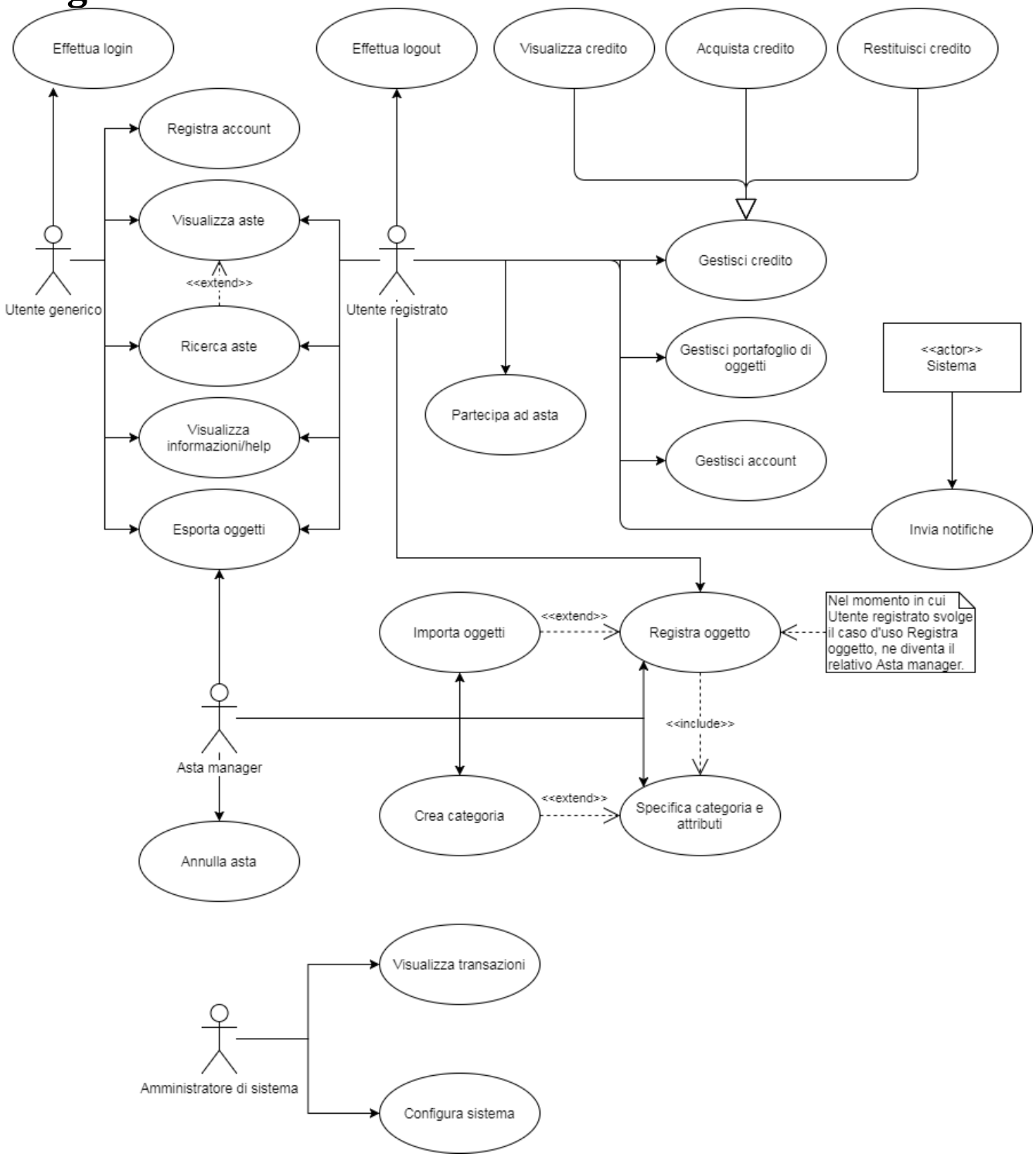


Diagramma di dominio:

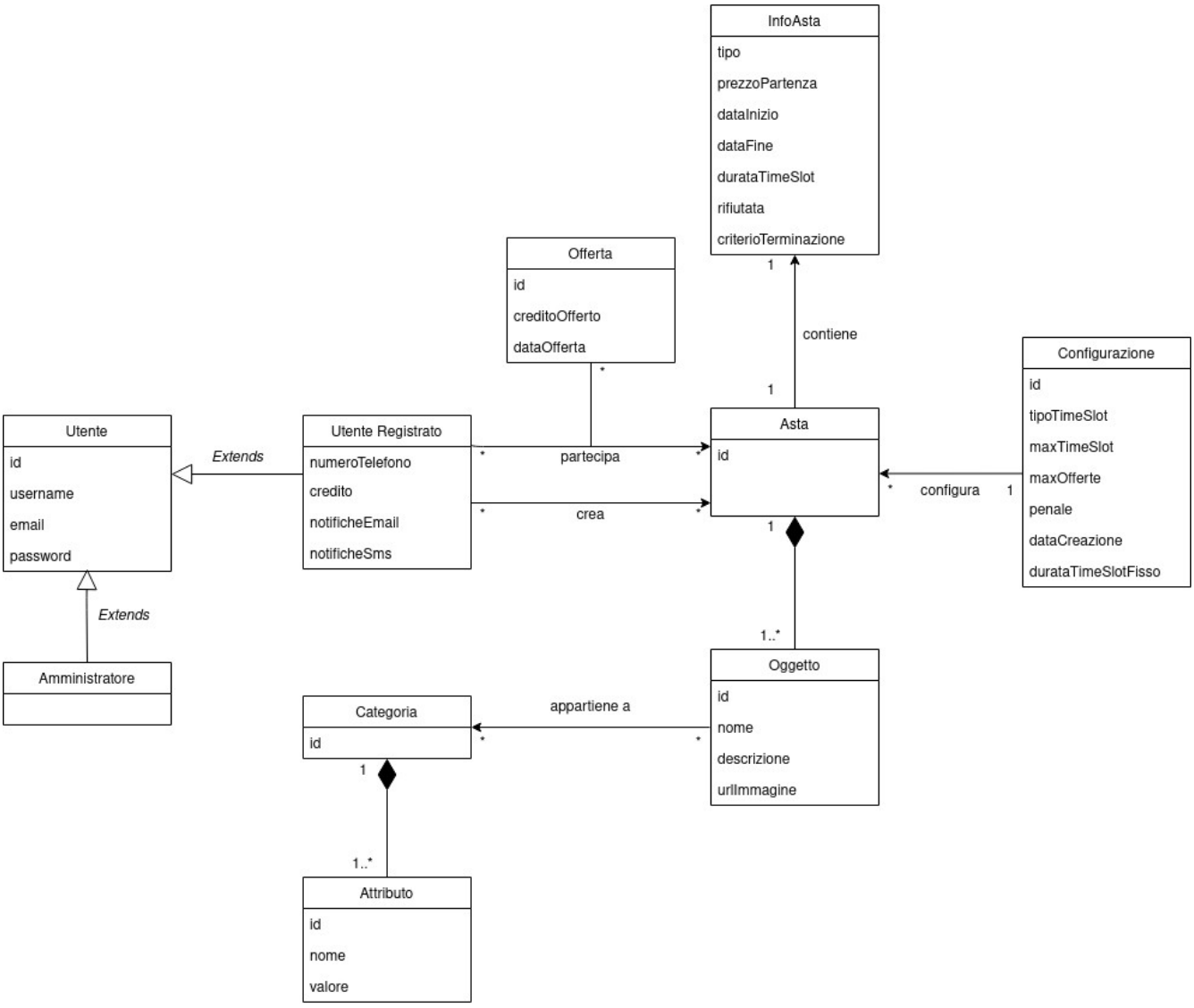
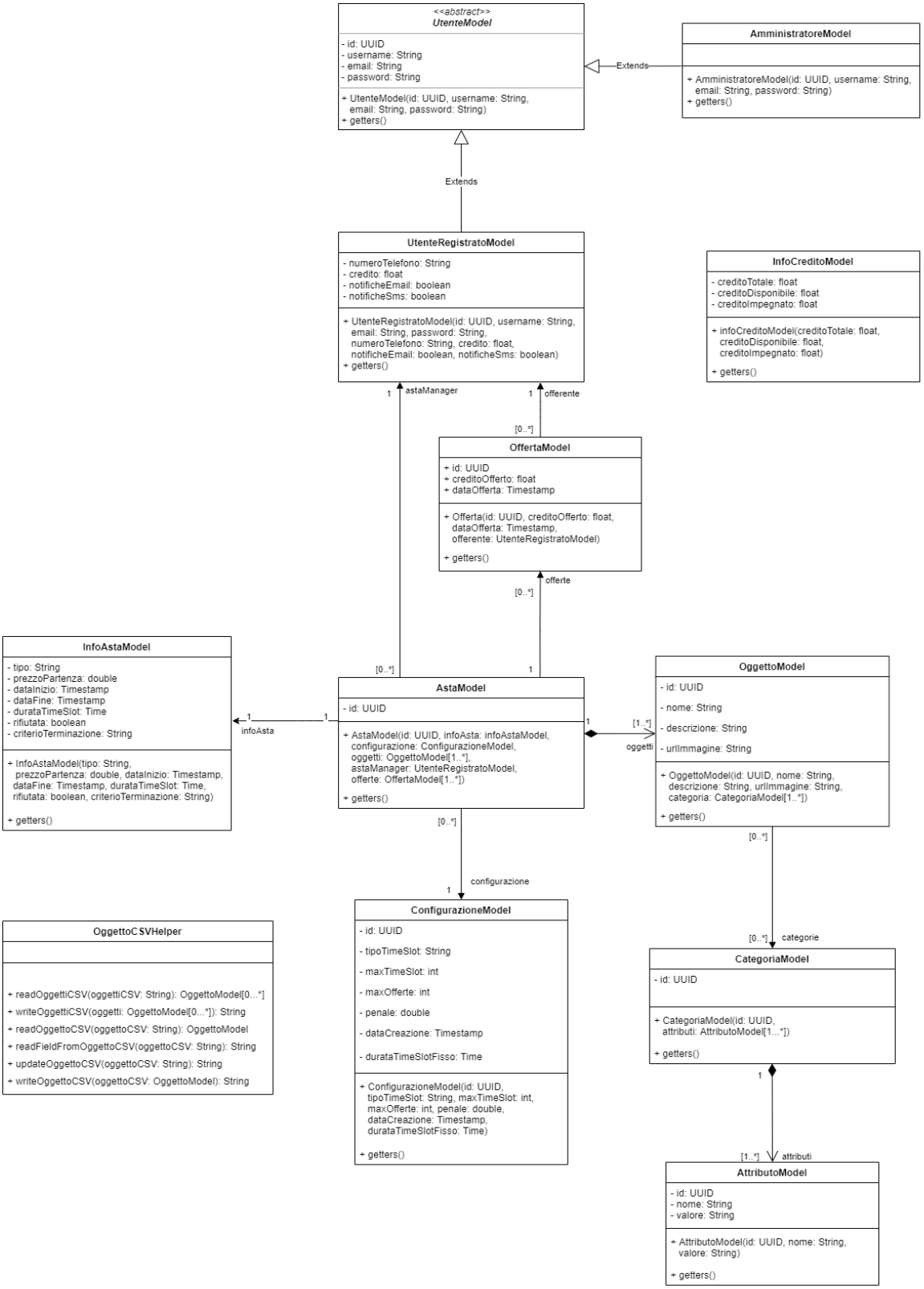
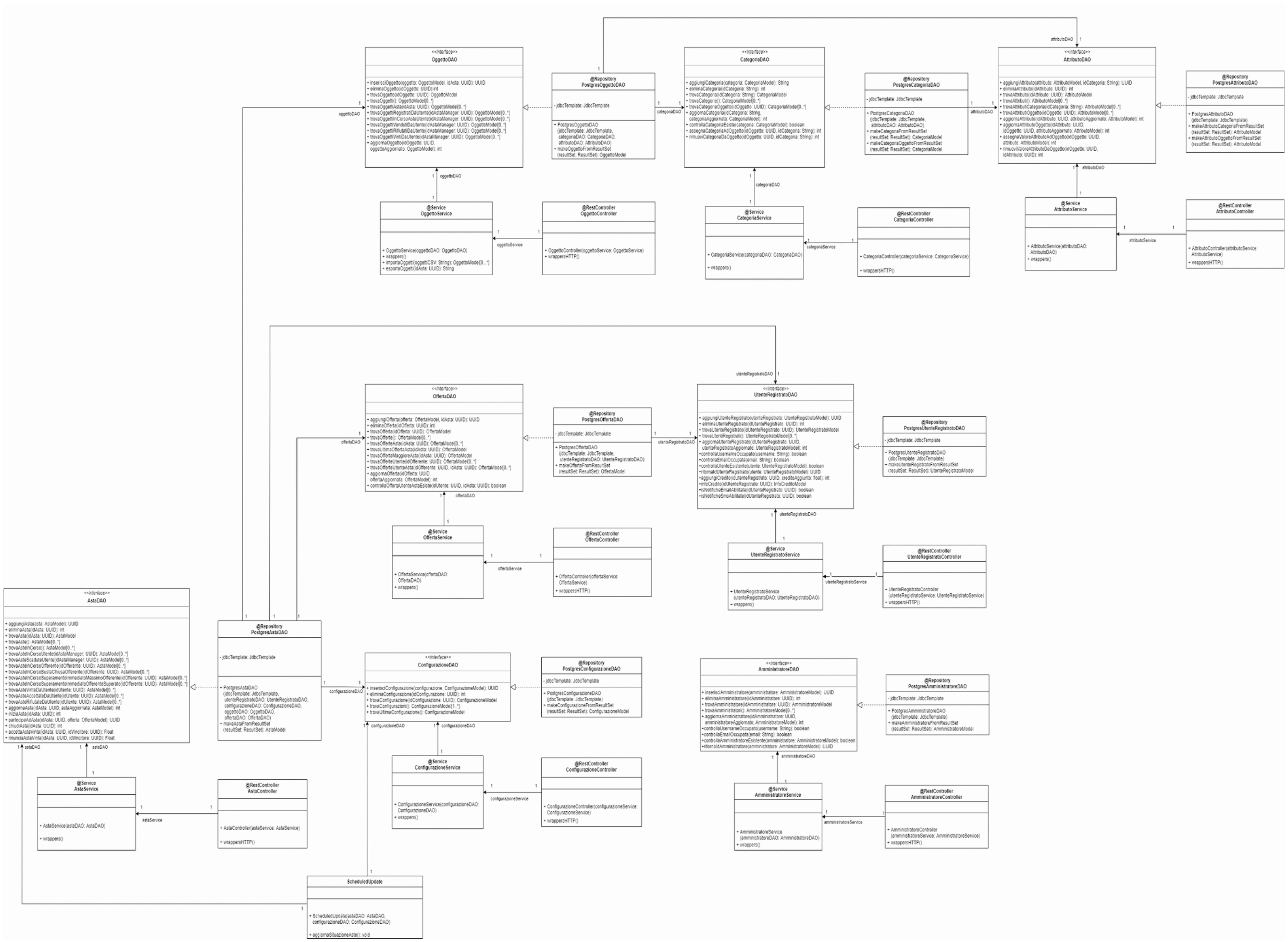


Diagramma di classe:





OggettoDAO
@Repository PostgresOggettoDAO
CategoriaDAO
@Repository PostgresCategoriaDAO
AttributoDAO
@Repository PostgresAttributoDAO
@Service OggettoService
@RestController OggettoController
@Service CategoriaService
@RestController CategoriaController
@Service AttributoService
@RestController AttributoController
OffertaDAO
@Repository PostgresOffertaDAO
UtenteRegistratoDAO
@Repository PostgresUtenteRegistratoDAO
@Service OffertaService
@RestController OffertaController
@Service UtenteRegistratoService
@RestController UtenteRegistratoController
AstaDAO
@Repository PostgresAstaDAO
ConfigurazioneDAO
@Repository PostgresConfigurazioneDAO
AmministratoreDAO
@Repository PostgresAmministratoreDAO
@Service AstaService
@RestController AstaController
@Service ConfigurazioneService
@RestController ConfigurazioneController
@Service AmministratoreService
@RestController AmministratoreController
ScheduledUpdate

I diagrammi probabilmente si leggono meglio direttamente nei file originali (Cartella Documentazione).

Per la struttura delle classi nel backend abbiamo utilizzato la configurazione consigliata da Spring Boot, gli oggetti il cui nome termina con Model sono semplici oggetti (Plain old java objects “POJO”) senza funzionalità e vengono solamente utilizzati come unità dati.
Per ogni tipo di Model considerato dal sistema esiste una relativa classe incaricata di dialogare con il database, queste classi sono marcate dall’ annotazione @Repository. Le funzioni necessario al corretto funzionamento sono dichiarate nella rispettiva interfaccia con nome terminante per “DAO” (Data access object). La corrispondente classe @Repository deve implementare queste funzioni.
Tutte le @Repository di questo progetto sono implementate con l’ottica di interfacciarsi con un database PostgreSql, da cui il prefisso “Postgres”. Tramite le classi Service, marcate dall annotazione @Service che sono fondamentalmente dei Wrappers per le classi @Repository è possibile gestire la Dependeny Injection passando nel costruttore una qualsiasi classe che implementa l’ interfaccia DAO.
Le classi controller invece espongono all’ esterno endpoint rest per richieste http. Anche questa classe va a chiamare I metodi definiti in Service a loro volta definiti nella classe @Repository.

Diagramma database:

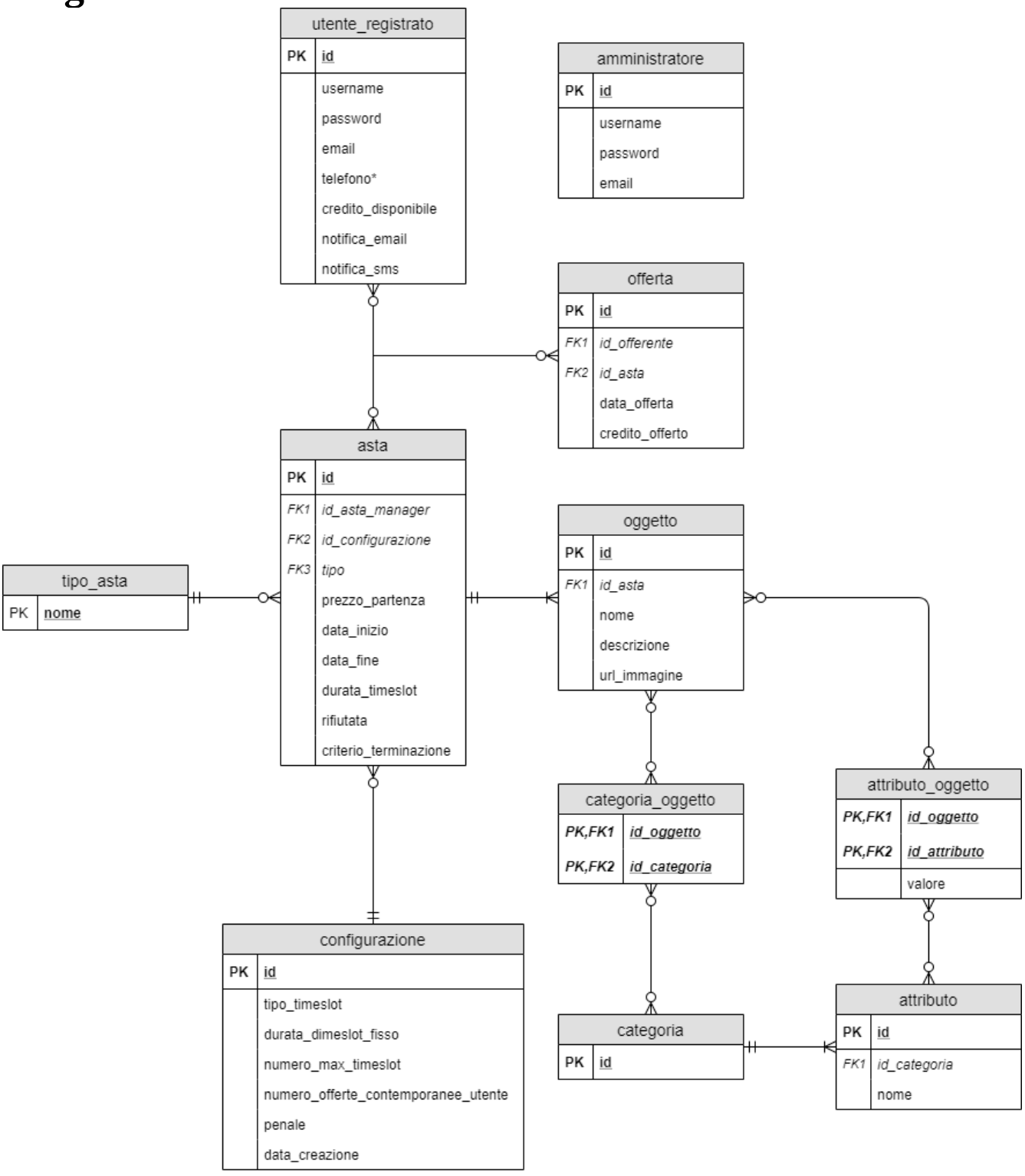


Diagramma architettura software:

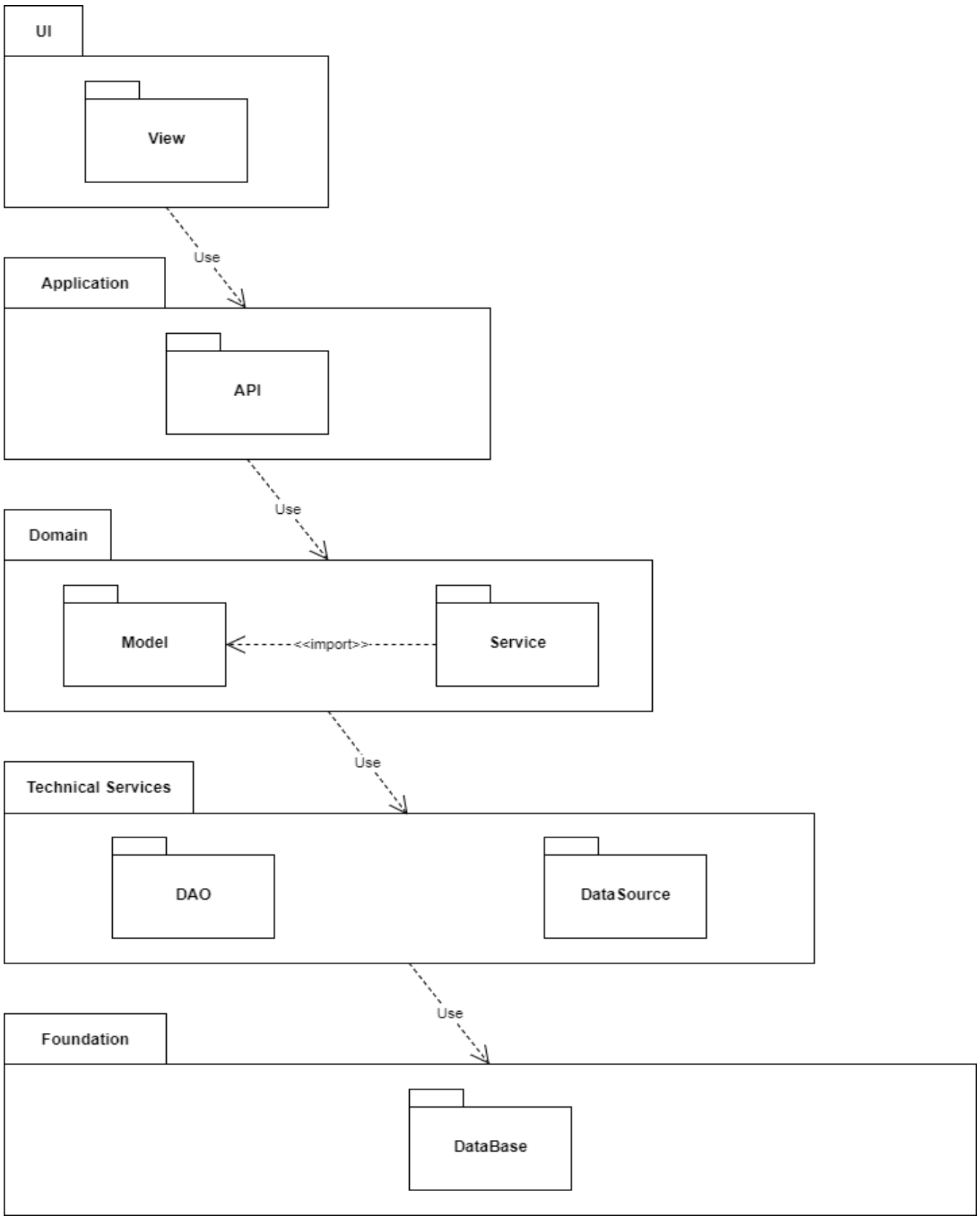
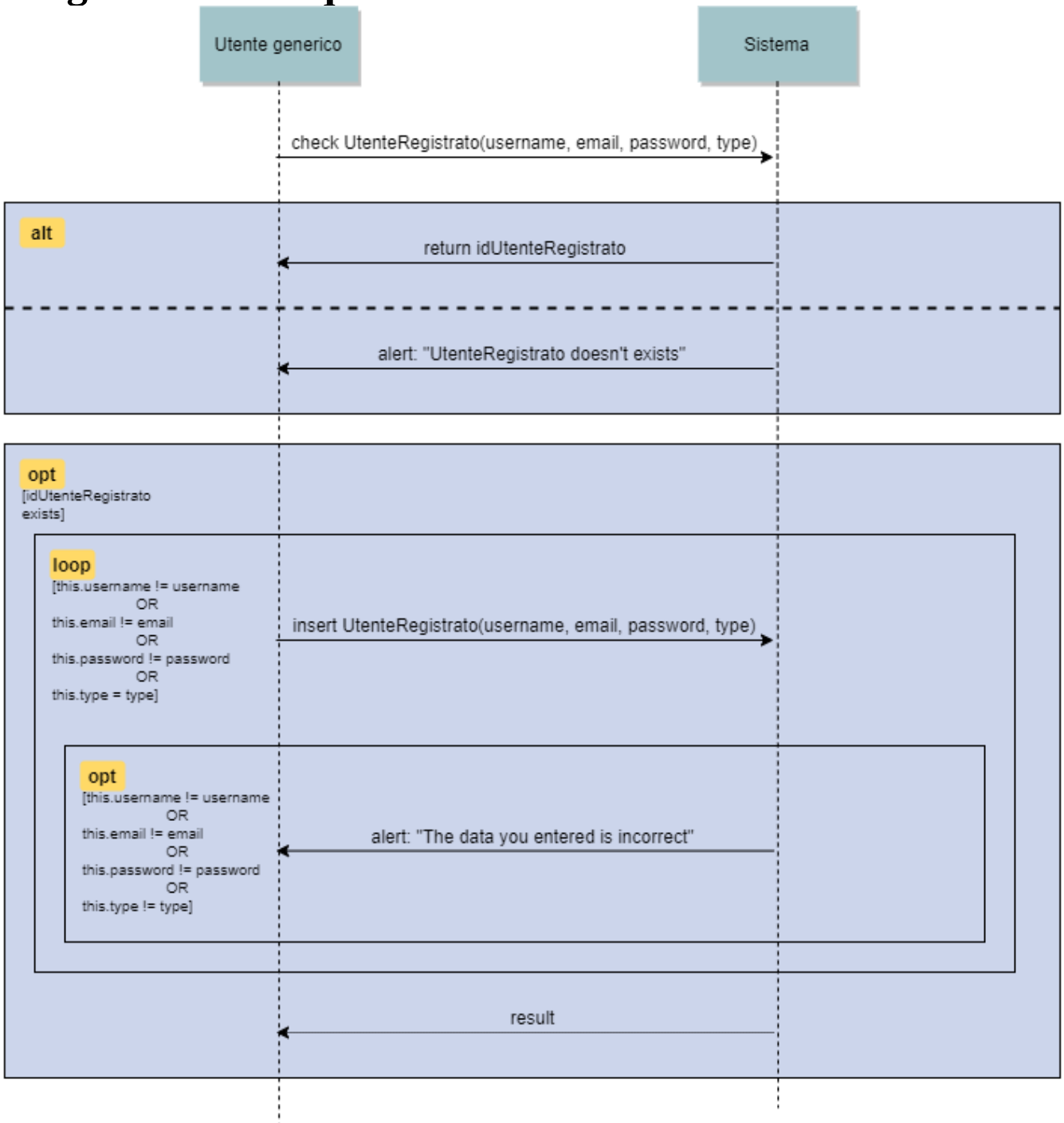


Diagramma di sequenza:



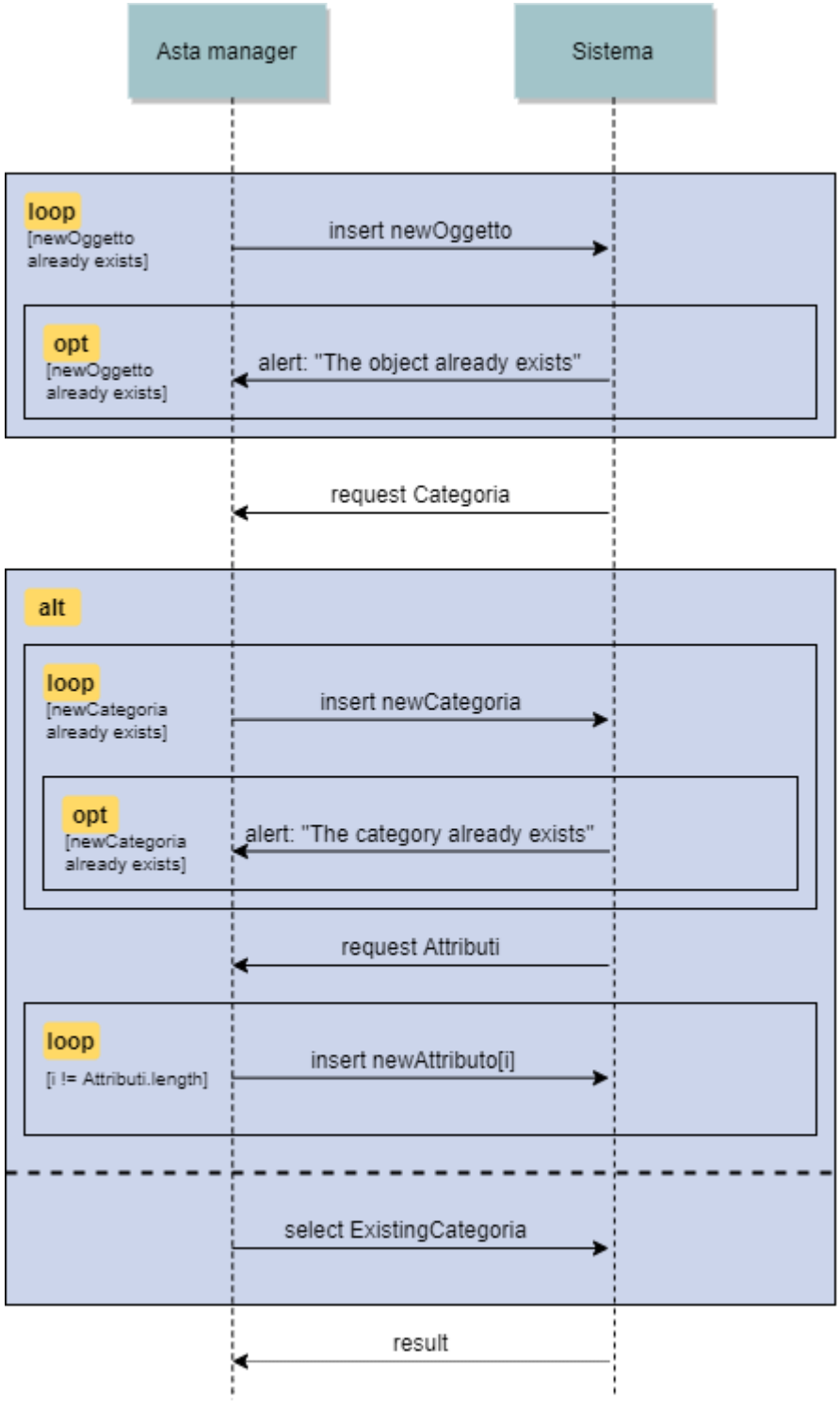
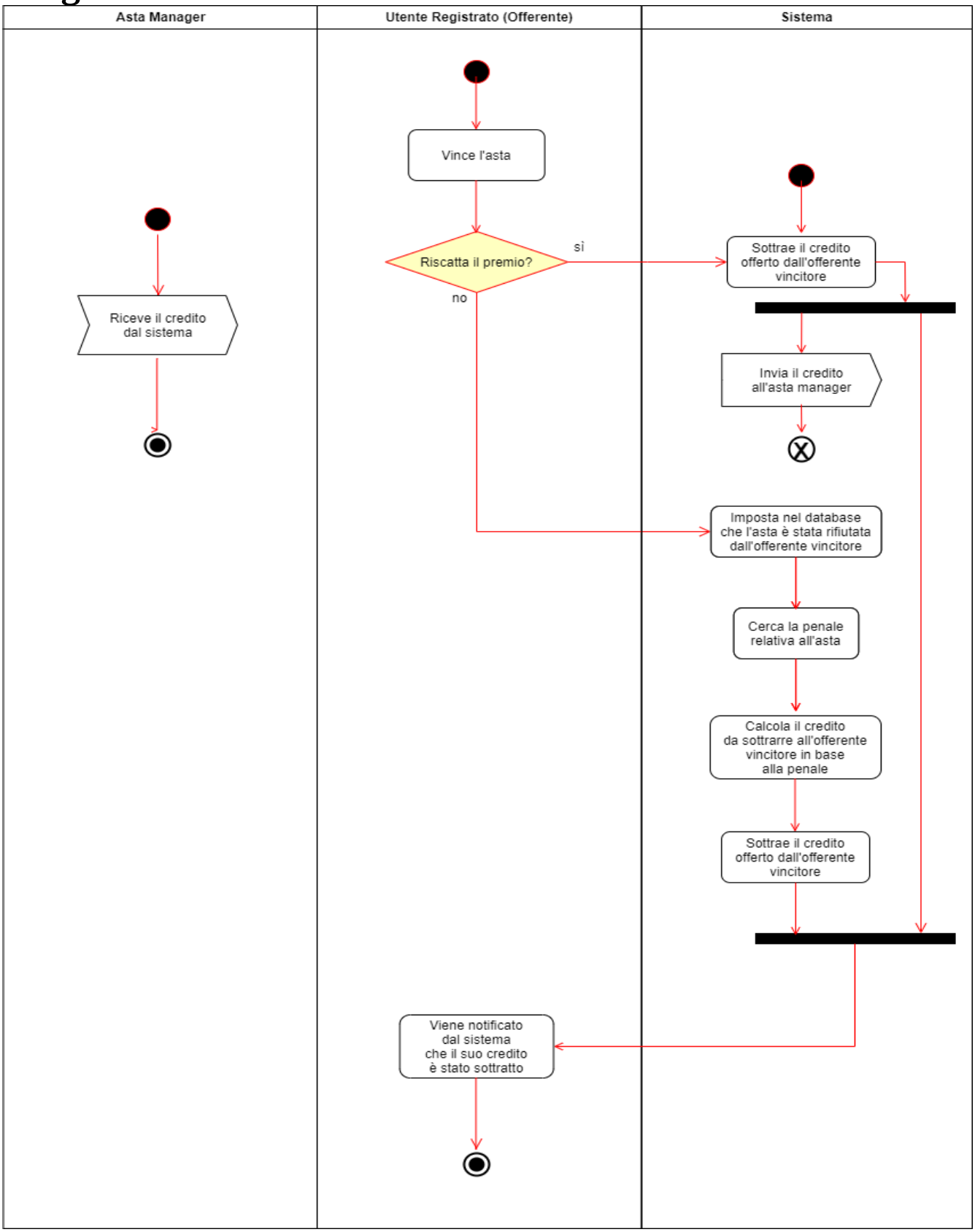
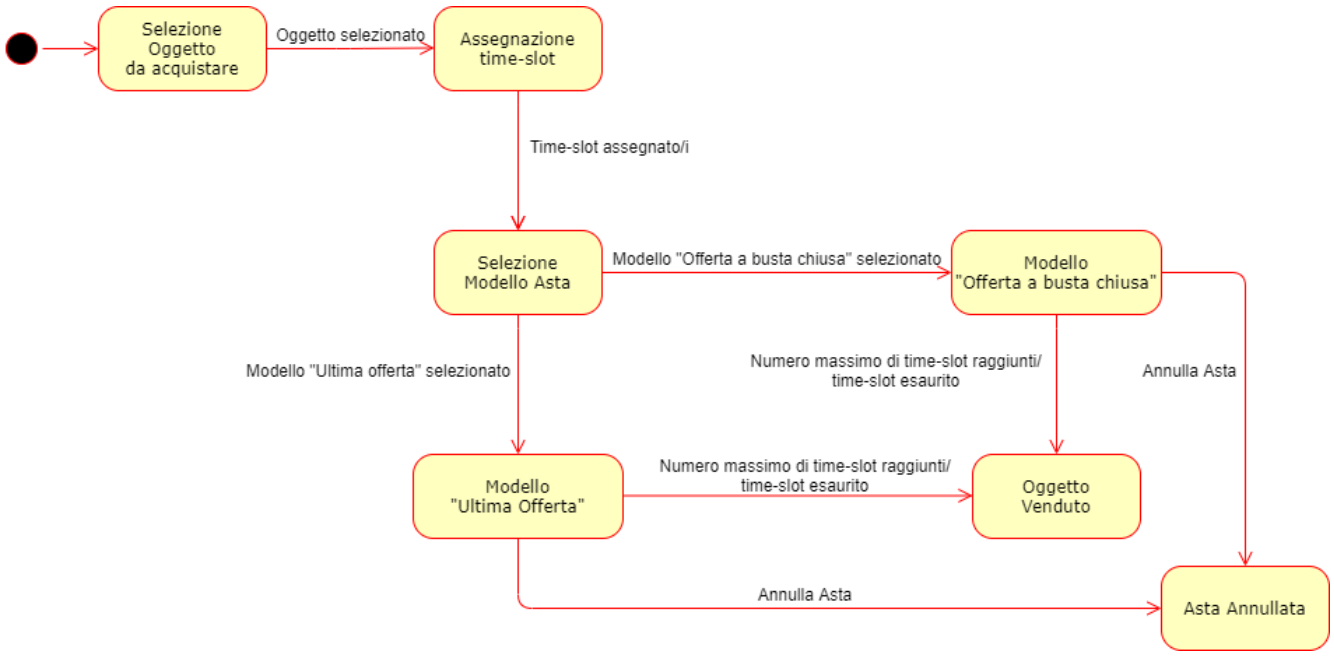
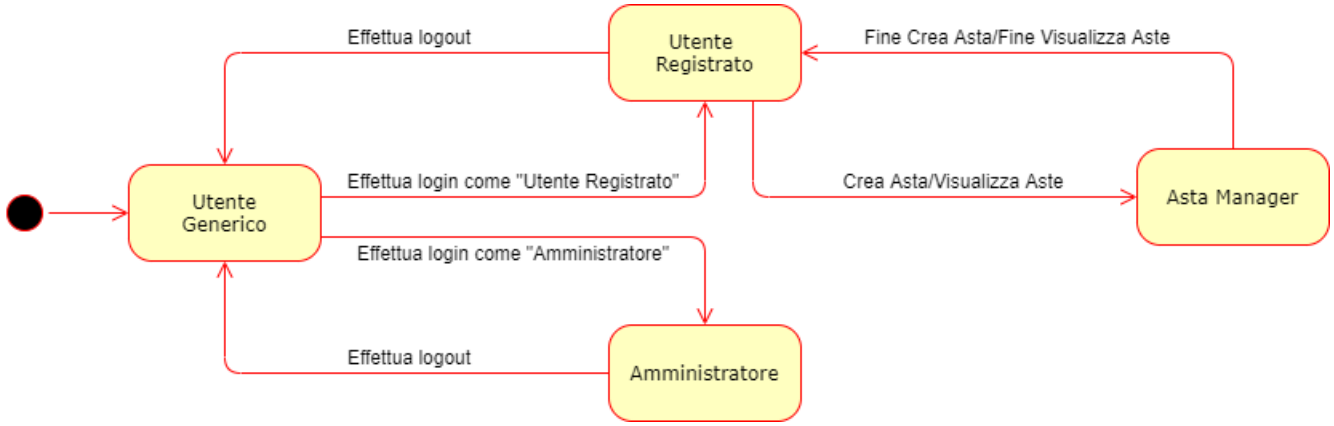

Asta manager
Sistema
loop
[newOggetto already exists]
insert newOggetto
opt
[newOggetto already exists]
alert: "The object already exists"
request Categoria
alt
loop
[newCategoria already exists]
insert newCategoria
opt
[newCategoria already exists]
alert: "The category already exists"
request Attributi
loop
[i != Attributi.length]
insert newAttributo[i]
select ExistingCategoria
result

Diagramma di attività:



Statechart:


Selezione Oggetto da acquistare
Oggetto selezionato
Assegnazione time-slot
Time-slot assegnato/i
Selezione Modello Asta
Modello "Offerta a busta chiusa" selezionato
Modello "Offerta a busta chiusa"
Modello "Ultima offerta" selezionato
Numero massimo di time-slot raggiunti/ time-slot esaurito
Annulla Asta
Modello "Ultima Offerta"
Numero massimo di time-slot raggiunti/ time-slot esaurito
Oggetto Venduto
Annulla Asta
Asta Annullata


Effettua logout
Utente Registrato
Fine Crea Asta/Fine Visualizza Aste
Utente Generico
Effettua login come "Utente Registrato"
Crea Asta/Visualizza Aste
Asta Manager
Effettua login come "Amministratore"
Effettua logout
Amministratore

Design Pattern e Design Principles applicati

Creazionali:

- Singleton: per tutti i componenti che vengono iniettiati automaticamente tramite l'annotazione di SpringBoot @AutoWired.

Strutturali:

- Private Class Data (PIMPL: Private Implementation): per ridurre la visibilità degli attributi delle classi (per esempio, la classe InfoAstaModel per incapsulare gli attributi di AstaModel).

Comportamentali:

- Strategy: per tutte le classi DAO e PostgresDAO, permettendo diverse implementazioni del Data Access Object a seconda della tipologia di database impiegata.

Architetturali:

- Architettura Multi-Tier: in particolare, un'architettura three-tier (Presentation tier, Logic tier and Data tier).
- Layers: architettura basata su layers (View-Controller-Service-DAO-Database).
- Service Layer: per organizzare i servizi in una serie di classi Service nel layer della logica applicativa, in modo che i metodi di una stessa classe Service offrano un più piccolo set di servizi della stessa tipologia.
- MVC (Model-View-Controller): le classi del tier Model sono tutte le classi Model e quelle annotate con @Service, le classi del tier View sono quelle della parte front-end, le classi del tier Controller sono tutte quelle annotate con @RestController.
- Data Access Object (DAO): per la gestione della persistenza e per stratificare ed isolare l'accesso al data layer (database) tramite query (poste all'interno dei metodi della classe).
- Repository: ambiente in cui vengono gestiti i dati persistenti attraverso l'implementazione del DBMS PostgreSQL.

Metodologia:

- Responsibility (GRASP):
  - Information Expert: la responsabilità è assegnata alle classi che possiedono tutte le informazioni essenziali (le diverse classi DAO per ciascun tipo di classe Model del dominio).
  - Creator: le classi vengono istanziate nelle classi DAO o nelle classi Service.
  - Controller: classi Controller.
  - Low Coupling: per supportare basso impatto in una classe dei cambiamenti in altre classi e alto potenziale di riuso (anche la dipendenza tra classi è bassa, tranne per quanto riguarda le classi DAO e PostgresDAO, ma ciò è dovuto alla particolare implementazione con PostgreSQL; ciononostante, l'impatto dei cambiamenti in altre classi rimane basso e il potenziale di riuso rimane alto).
  - High Coesion: a supporto del Low Coupling, per rendere coese le responsabilità delle classi (infatti, per quanto riguarda le classi PostgresDAO, i dati che generalmente dovrebbero essere recuperati da altre classi PostgresDAO vengono recuperati invocando proprio i metodi di queste ultime, lasciando alle singole classi il pieno e solo controllo dei dati della propria tabella relazionale (del database) a cui fanno riferimento).
  - Polymorphism: UtenteModel, UtenteRegistratoModel e AmministratoreModel.
  - Pure Fabrication: OggettoCSVHelper.

    - Indirection: classi Controller → classi Service → classi DAO.
    - Protected Variations: interfacce DAO → implementazioni PostgresDAO.
- Make it run, make it right, make it fast, make it small: per sfruttare una strategia di sviluppo iterative per risolvere problem di business con soluzioni semplici e veloci.

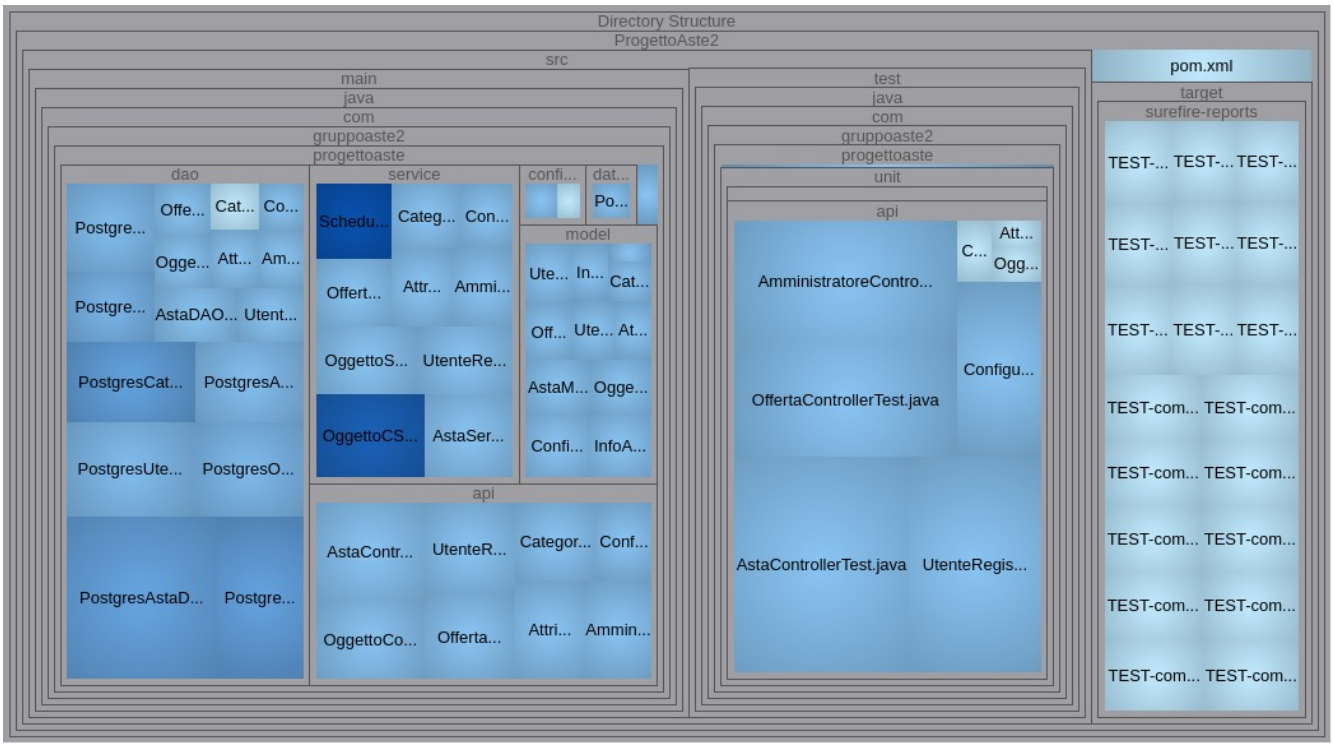
Altri Pattern:

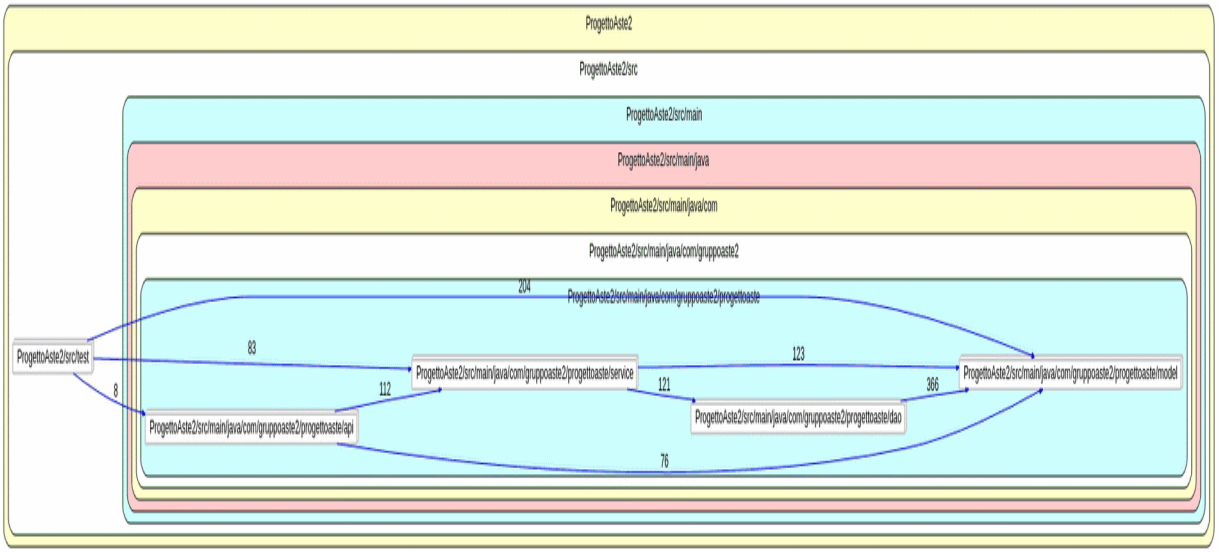
- Dependency Injection: per semplificare lo sviluppo e migliorare la testabilità. I componenti dipendenti sono le classi PostgresDAO, gli injector sono le relative classi Service (che creano, a richiesta, le istanze delle classi che implementano le dependency interfaces, ovvero le classi DAO) e le interface contracts, ovvero le dichiarazioni delle dipendenze del componente, avvengono attraverso le annotazioni @Qualifier nelle classi injector (Service).
- Inversion of Control (IoC): un componente di livello applicativo riceve il controllo da un componente appartenente a un libreria riusabile. Pattern attuato tramite la Dependency Injection.
- Mock Object: per simulare in modo controllato il comportamento dei layer sottostanti alle classi Controller in ambito di testing.
- Null Object: utilizzo della classe java Optional per istanziare gli oggetti di ritorno di un metodo senza essere sicuri dell'esistenza effettiva di un loro riferimento. In questo modo, se l'oggetto restituito da un metodo ha riferimento null, il corpo del suo oggetto Optional è vuoto, ma l'oggetto Optional in sé ha un riferimento e non è più null, permettendo di eseguire operazioni su di esso e di controllare facilmente se è vuoto.
- Servant: per dare funzionalità ad un gruppo di classi (le classi Model) senza definirle in ognuna di esse, bensì nelle classi a loro supporto (le classi Service).

Design Principles:

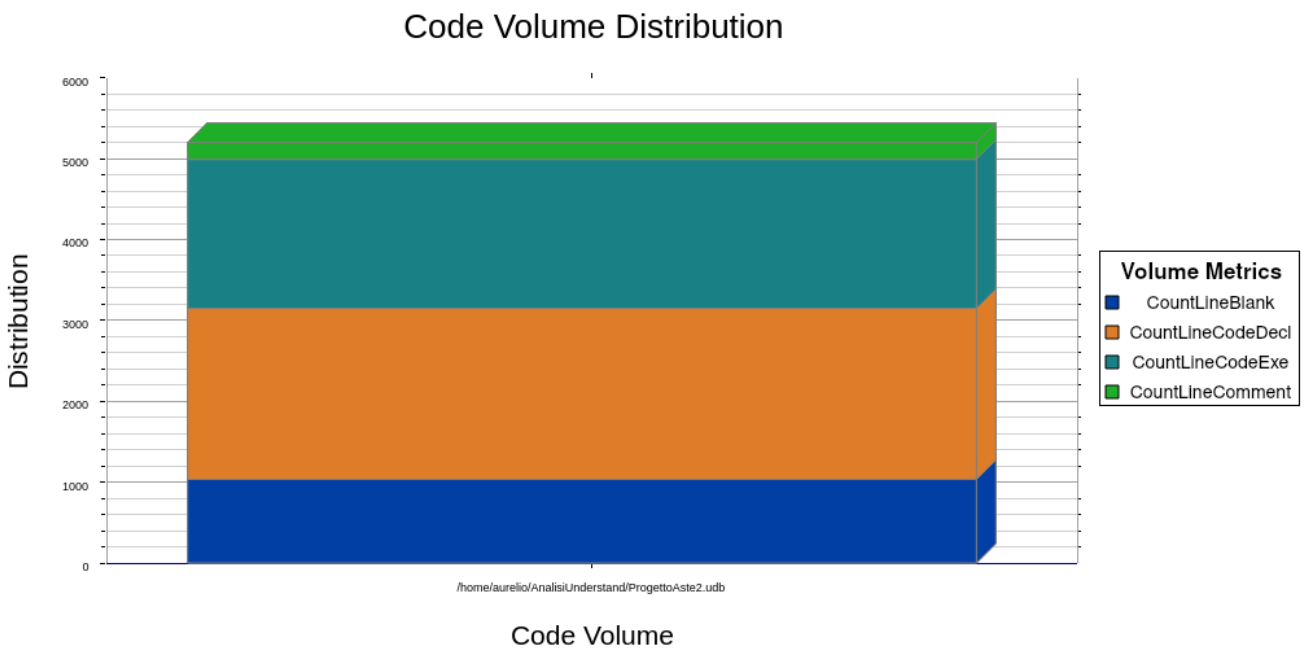
- Open-Box Extensibility: per permettere al codice di essere esteso, modificando direttamente il codice sorgente.

Analisi Understand:

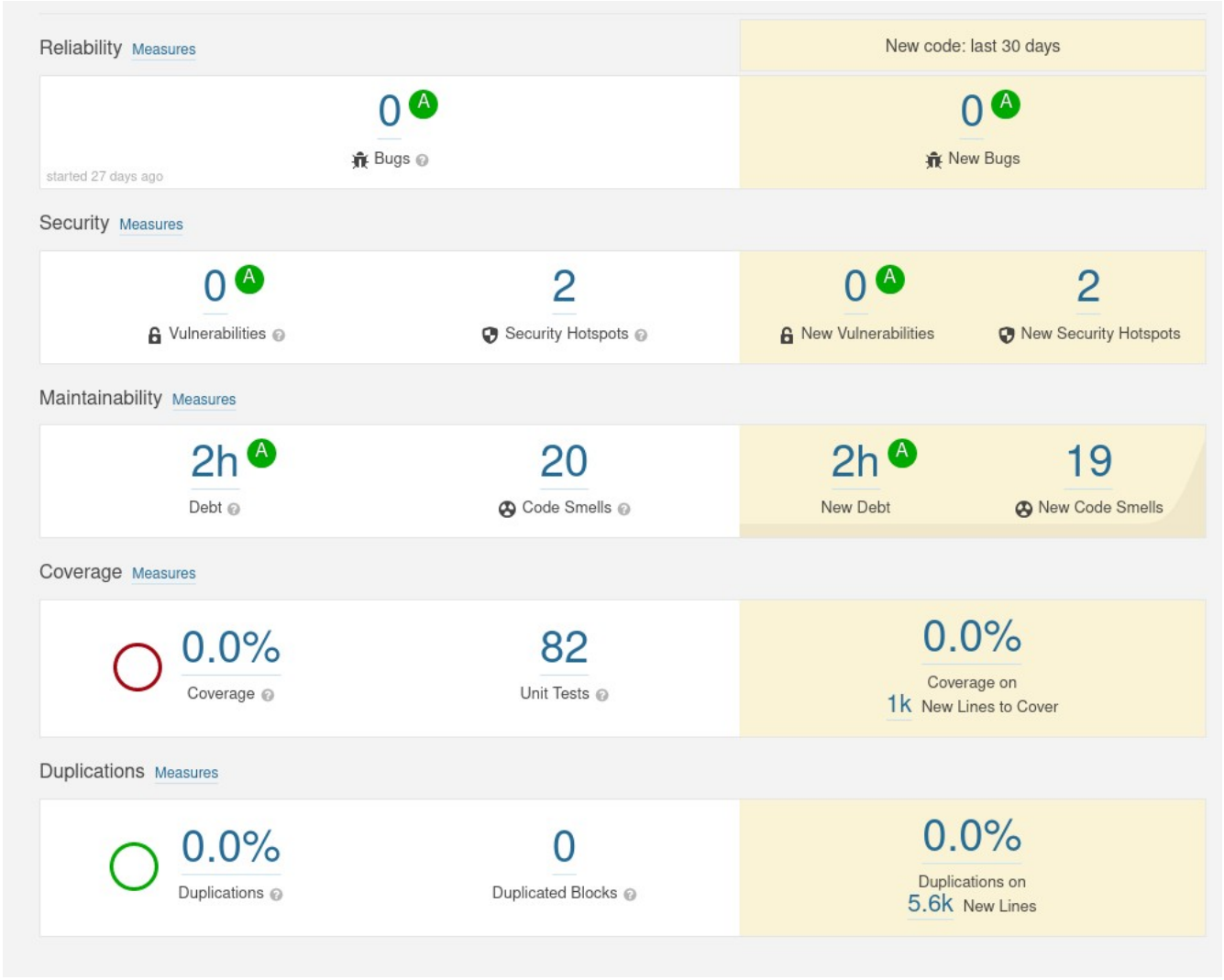




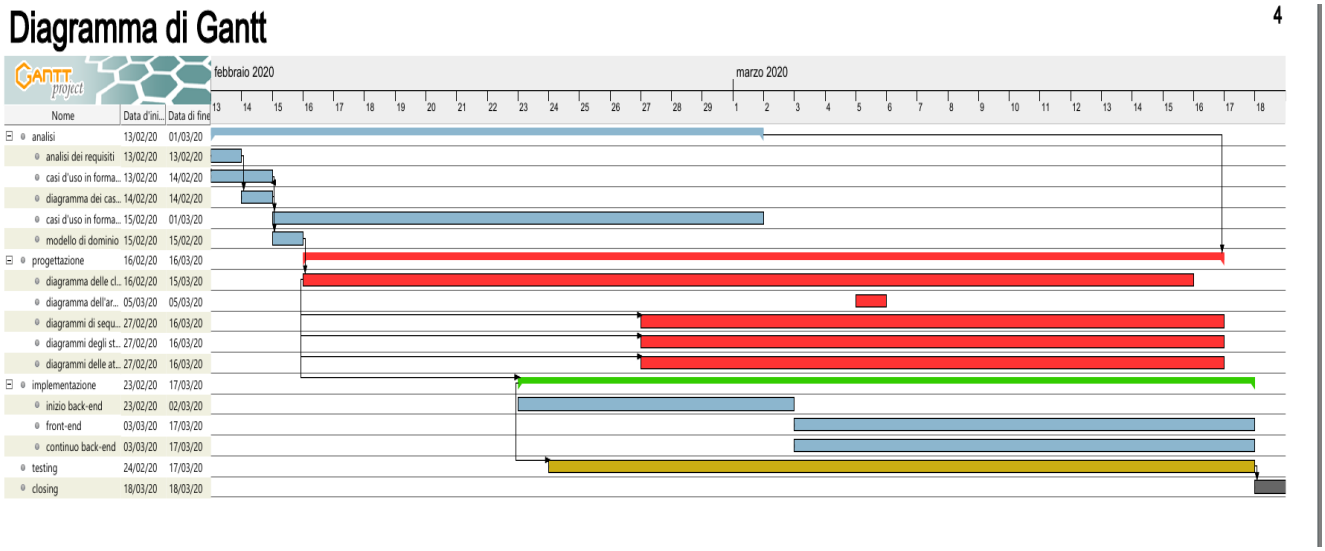
Si nota che la maggior parte delle dipendeze si trovano tra le classi di test e le relative classi che vengono testate e dalle classi che utilizzano le unità dati Model per operare.



Analisi Sonarcloud:



Diagrammi Gantt:



Considerazioni sul progetto:

Il progetto è stato estremamente utile per capire meglio le dinamiche del lavoro in team su un progetto software e di tutti I problemi e vantaggi che scaturiscono da questo lavoro collaborativo.
Come si può notare dalla differenza dei diagrammi Gantt le tempistiche e I ruoli inizialmente assegnati non sono stati rispettati, complici l'inesperienza del team, la situazione di emergenza ed un periodo di malattia di un membro del gruppo (Grassi Marco).
I ruoli inizialmente stabiliti prevedevano come responsabile del Frontend Frigati Francesco e come collaboratore Tassi Andrea mentre il Backend aveva come responsabile Grassi Marco e Villa Fabio.

A causa di inesperienza si sono un po' sottovalutate le difficoltà dei compiti assegnati e questo ha portato a periodi di relativamente scarsa produttività e partecipazione.

Adottando però una procedura di lavoro Agile si è cercato di "riportare sui binari" il progetto assegnando funzionalità più piccole ma più velocemente raggiungibili. Il team di lavoro è stato abbastanza flessibile permettendo a tutti di occuparsi anche solo per un breve periodo di tutte le componenti dell' applicativo.

Alcune funzioni risultano purtroppo non implementate:

- La gestione delle notifiche, la cui implementazione era possibile tramite Websocket non è stata implementata per motivi di tempo per garantire lo sviluppo di funzioni più vitali.

- L'esportazione/importazione di oggetti in formato csv non è possibile da front end, risultano però pronte e funzionanti (sono state testate) le rispettive funzioni nel backend.

- Per la chiusura delle aste non vengono presi in considerazione I criteri di terminazione, tutte le aste vengono trattate come aste la cui terminazione è definita dall’ esaurimento di tutti I time slot.

- Stesso discorso per I tipi di offerta, tutte le offerte vengono trattate come offerte per aste a superamento immediato.

La struttura per colmare a queste lacune è già presente nel backend, con una migliore gestione del tempo allocato al progetto sarebberò potute essere implementate.

A fine progetti I ruoli svolti sono risultati:

-Tassi Andrea: Inizialmente incaricato alla documentazione, poi successivamente passato al Backend dove ha svolto il ruolo di responsabile dei Test e ha implementato considerevoli parti del backend e molte funzioni vitali.

-Villa Fabio: Inizialmente ha contribuito al backend implementando diverse classi poi successivamente passato al frontend dove ha implementato più pagine.

-Grassi Marco: Implementato insieme ad Andrea Tassi gran parte del Backend, responsabile della creazione del database PostgreSql e della scelta degli strumenti tecnologici utilizzati. Successivamente passato al frontend dove ha sviluppato molteplici pagine.

-Frigati Francesco: Inizialmente responsabile per la raccolta di informazioni su come sviluppare un frontend è passato a sviluppare parte dei test nel backend e successivamente alla stesura di molti diagrammi facenti parte della documentazione.

Ci riteniamo piuttosto soddisfatti del lavoro svolto e delle competenze ottenute dal lavoro svolto su questo progetto