Object Design Document

AlcoList

Riferimento	
Versione	0.5
Data	
Destinatario	Prof. Gravino
Presentato da	Baldi Maria Rosaria Conte Melania Di Zenzo Carmine Federico Zaccardi Mario
Approvato da	

Sommario

Team Members

Nome	Acronimo	Informazioni di contatto
Baldi Maria Rosaria	BMR	m.baldi24@studenti.unisa.it
Conte Melania	CM	m.conte49@studenti.unisa.it
Di Zenzo Carmine Federico	DCF	c.dizenzo2@studenti.unisa.it
Zaccardi Mario	ZM	m.zaccardi@studenti.unisa.it

Revision History

Data	Versione	Descrizione	Autori
11/01/2023	0.1	Prima stesura	BMR, CM, DCF, ZM
12/01/2023	0.2	Linee guida per la documentazione, definizioni e acronimi	BMR, CM, DCF, ZM
13/01/2023	0.3	Package	BMR, CM, DCF, ZM
14/01/2023	0.4	Class Diagram	BMR, CM, DCF, ZM
16/01/2023	0.5	Design Pattern	BMR, DCF

1 Introduzione

L'Object Design Document illustra i diversi dettagli legati alla fase implementativa del Sistema AlcoList. In particolare, descrive gli object design goals, i trade-off di progettazione definiti dagli sviluppatori, le linee guida da seguire per le interfacce dei sottosistemi, i design pattern utilizzati, la decomposizione dei sottosistemi in packages e classi e, infine, la specifica delle interfacce delle classi.

1.1 Object Design Goals

Gli obiettivi di object design posti per il Sistema sono:

- **Astrazione**: le interfacce devono essere intuitive e di un alto livello, così da garantire un'implementazione corretta e comprensibile;
- **Modularità**: le unità del Sistema devono essere organizzate in moduli facilmente collegati;
- **Riusabilità**: il riuso del codice deve essere prioritario e verrà fornito attraverso l'ereditarietà e i design pattern.

1.2 Object Design Trade-off

Tempo di rilascio Vs funzionalità

Per rispettare le scadenze del Progetto potrebbe essere necessaria l'implementazione parziale di alcune delle funzionalità richieste.

Tempo di rilascio Vs Correttezza

Per rispettare le scadenze del Progetto, l'architettura sviluppata sarà soltanto quella relativa all'ambiente DEV (development), tralasciando momentaneamente lo sviluppo dell'ambiente PROD (production).

Portabilità Vs Efficienza

Il Sistema è progettato in modo che Browser diversi e dispositivi con risoluzioni diverse (smartphone, laptop, tablet, etc) possano visualizzare correttamente le pagine web del sito AlcoList sfruttando al meglio lo spazio del display. Questo implica che il livello di efficienza garantito non sia lo stesso per ogni dispositivo, poiché una tale adattabilità richiederebbe un carico maggiore da gestire.

Velocità Vs memoria

Per garantire tempi di risposta rapidi, si è preferito utilizzare query che risultano più veloci a discapito dello spazio che occupano in memoria; in particolar modo viene introdotta ulteriore ridondanza di dati

Costruire Vs Comprare

Sebbene utilizzare software già realizzato da altri permetta l'utilizzo di funzionalità già complete o una minore quantità di lavoro per gli sviluppatori, si è deciso di realizzare la maggior parte del Sistema partendo da zero, utilizzando componenti esterne soltanto in alcuni casi. Il motivo per cui è stata presa questa decisione riguarda l'aumento dei costi e l'impegno necessario per integrare le componenti già realizzate con quelle costruite dagli sviluppatori.

Nella seguente tabella, il Design Goal in **grassetto** indica il design goal prioritario.

Trade-Off	
Tempi di rilascio	Funzionalità
Portabilità	Efficienza
Velocità	Memoria
Costruire	Comprare

1.3 Componenti off-the shelf

Il Sistema utilizzerà i seguenti componenti off-the-shelf:

- **Apache Tomcat**: un Web Server con annesso application container per applicazioni scritte in Java;
- **Spring Boot:** strumento che rende semplice lo sviluppo di applicazioni Web e microservizi basato su framework Spring.
- **JPQL:** linguaggio di query che consente di definire query di database in base al modello di entità.
- **Docker Swarm:** strumento di orchestrazione del contenitore che esegue l'applicazione Docker, configurato per essere unito in un cluster.
- **Kubernetes:** sistema open-source di orchestrazione e gestione di container che consente di automatizzare processi manuali necessari per il deployment, la gestione e la scalabilità delle applicazioni containerizzate.
- **JSON.simple:** toolkit Java per codificare o decodificare il testo JSON.
- **Springfox-Swagger2:** suite di strumenti per sviluppatori API utilizzato per descrivere e documentare le API RESTful.
- **JUNIT**: framework di unit testing per il linguaggio Java.
- **Mockito**: framework per realizzare oggetti Mock partendo da un'interfaccia o una classe e che permette di eseguire diverse tipologie di test.

Per l'implementazione della sezione front-end sono stati utilizzati i seguenti framework:

- **bootstrap-5.2.3**: framework di sviluppo front-end open source che contiene modelli di progettazione basati su HTML e CSS.
- **jquery-3.6.1**: libreria JavaScript che consente la manipolazione e la gestione di documenti HTML.
- **boxicons-2.1.1**: libreria open source di icone.
- **chart.js-2.5.0:** libreria per la creazione di grafici interattivi e animati.
- **Selenium IDE 3.17.2:** software open source per automatizzare le pagine web.

1.4 Linee guida per la documentazione di Interfacce

Il progetto AlcoList include una serie di convenzioni e regole a cui gli sviluppatori fanno riferimento durante la progettazione del Sistema.
Di seguito sono riportati un link alle convenzioni utilizzate per definire le linee guida su:

- HTML: https://www.w3schools.com/html/html5_syntax.asp
- JavaScript: https://www.w3schools.com/js/js_syntax.asp

Inoltre, il progetto AlcoList è realizzato con due IDE di sviluppo differenti:

- Eclipse IDE for Enterprise Java and Web Developers 2022-09 per il backend
- IntelliJ IDEA 2021.3 per lo sviluppo del frontend

Inoltre, il Progetto è suddiviso:

- Il progetto è suddiviso in due macro-componenti: AL-frontend e AL-backend.
- Il nome di una classe deve rispettare il seguente formato: **NomeClasse**.
- Il nome di un metodo o di una variabile di istanza deve rispettare la notazione Camel Case.
- Un intero metodo, compreso di intestazione ed istruzioni, è preceduto e seguito da una riga vuota.
- I commenti, laddove necessari, avranno formato **// commento** se si estendono su una sola riga, altrimenti se un commento si estende su più righe presenta il formato **/* commento */**.

- Ogni classe e ogni metodo devono essere corredate da commenti che rispettano lo standard utilizzato da Javadoc per la produzione di documentazione in formato HTML.

- Ogni Entity deve contenere un almeno un costruttore e i metodi getter e setter.

- Il package Service contiene tutte le classi che si occupano della logica di business del Sistema e agisce da interlocutore tra le classi contenute nei sub-package Controller e Repository.

- Il package webapp contiene dei sub-package in cui sono organizzati tutti i file che si occupano dell'interfaccia utente (JSP, JS, immagini e fonts).

- Il package webapp contiene un sub-package "users" che contiene i file JSP smistati per ruolo dell'utente.

- Il package webapp contiene un sub-package "error" che contiene i file JSP inerenti ai vari errori di comunicazione col server (404, 401, 500).

- I file CSS si trovano nel sub-package chiamato "style".

- I file JavaScript si trovano nel sub-package chiamato "script".

- Le immagini **(*.SVG)** e l'icona **(*.ICO)** del sito si trovano nel sub-package chiamato "img".

- I font utilizzati si trovano nel sub-package "fonts".

- I vari file rispettano il seguente formato: **nomeFile**.

- I commenti nei file JavaScript, laddove necessari, avranno formato **//commento** se si estendono su una sola riga, altrimenti se un commento si estende su più righe presenta il formato **/*commento*/**.

- I commenti nei file JSP, laddove necessari, avranno formato **<!--commento-->** sia se si estendono su una che su più righe.

- Le librerie utilizzate (come JQuery o Bootstrap) vengono importate tramite link **(CDN)** nelle pagine che le utilizzano.

1.5 Definizioni ed acronimi

In questa sezione descriveremo i termini che sono stati utilizzati all'interno del Documento stesso divisi in tre sezioni principali: definizioni, acronimi ed abbreviazioni.

1. **Definizioni**:
   - **Package**: raggruppamento di classi, interfacce, file correlati o altri package;
   - **Design pattern**: template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità;
   - **Interfaccia**: insieme di signature delle operazioni offerte dalla classe;
   - **Camel Case**: è la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione nel mezzo della frase inizi con una lettera maiuscola, senza spazi o punteggiatura intermedi;
   - **Javadoc**: sistema di documentazione offerto da Java, che viene generato sottoforma di pagina web in modo da rendere la documentazione accessibile e facilmente leggibile.
   - **Maven**: strumento di gestione di progetti software ospitato da Apache Software Foundation.
   - **Container**: unità software standard che impacchetta il codice e tutte le sue dipendenze in modo che l'applicazione venga eseguita in modo rapido e affidabile da un ambiente informatico all'altro.
2. **Acronimi**:
   - **SDD**: System Design Document
   - **RAD**: Requirements Analysis Document
   - **CRUD**: Create Read Update Delete

1.6 Riferimenti

Per stilare la presente documentazione, si è preso come riferimento le slide fornite dal Docente del corso di Ingegneria del Software, Carmine Gravino, inserite nella sezione "M4" della piattaforma di e-learning della facoltà di Informatica. Inoltre, è stato consultato il libro di testo "Object-Oriented Software Engineering Using UML, Patterns and Java: Third Edition, di Bernd Bruegge ed Allen H. Dutoit" e, infine, si è consultata la documentazione relativa al RAD e SDD.

2 Package

In questa sezione viene mostrata la suddivisione del Sistema in package, in base a quanto definito nel documento di System Design.

2.1 Interface

- Package **Users**

  infoCocktail: mostra le info del cocktail selezionato;
  listaCocktail: mostra una lista con tutti i cocktail presenti in piattaforma;
  menu: mostra i cocktail presenti in menù;
  profile: mostra le info dell'utente loggato;

  - Package **Bartender**

    aggiungiCocktail: mostra un form di aggiunta di un cocktail;
    aggiungiIngredientiCocktail: si innesca dopo "aggiungiCocktail" e mostra un form di aggiunta degli ingredienti al cocktail;
    aggiungiProdotto: mostra un form di aggiunta di un prodotto;
    dashboard: pagina iniziale che mostra le ordinazioni presenti al banco (quelle da realizzare);
    drinkList: mostra la lista dei cocktail da preparare per la singola ordinazione;
    magazzino: mostra la lista di tutti i prodotti presenti in magazzino;
    modificaCocktail: mostra un form di modifica di un cocktail;
    modificaProdotto: mostra un form di modifica di un prodotto.

- Package **Manager**

  dashboard: mostra un'overview dell'andamento del Bar, informazioni sul personale e di vendita.

  addUser: mostra un form di aggiunta di un nuovo utente;

  editUser: mostra un form di modifica di un utente:

  infoUser: mostra i dettagli del singolo utente selezionato;

  listaUser: mostra la lista di tutti gli utenti presenti in database.

- Package **Waiter**

  selectTable: mostra la lista dei tavoli (con i relativi stati) selezionabili per effettuare le comande;

  addOrdination: si innesca dopo "selectTable" e mostra la lista dei cocktail da poter aggiungere alla comanda (quelli presenti in menù);

  editOrdination: mostra un form di modifica della comanda.

- Package **Errori**
  - **Error404**: viene visualizzata in caso di errore 404;
  - **Error401**: viene visualizzata in caso di errore 401;
  - **Error500**: viene visualizzata in caso di errore 500.

navBar: mostra il menù di navigazione laterale (caricata in tutte le schermate)

index: mostra la schermata di login.

2.2 ApplicationLogic

Questo package contiene i seguenti sub-package e le seguenti classi:

- Package **Controller**
  - **UserAccountController**
  - **ProductController**
  - **IngredientController**
  - **CocktailController**
  - **TablesController**
  - **OrdinationController**
  - **StatisticsController**

I controller elencati si occupano della gestione di richieste dal web.

- Package **Service**
  - **UserAccountService**
  - **ProductService**
  - **IngredientService**
  - **CocktailService**
  - **TablesService**
  - **OrdinationService**
  - **StatisticsService**
  - **EmailService**

I service elencati contengono la logica applicativa e l'implementazione delle funzionalità transazionali.

- Package **DTO**
  - **UserAccountDTO**
  - **UserAccountResultDTO**
  - **ProductDTO**
  - **ProductResultDTO**
  - **IngredientDTO**
  - **CocktailDTO**
  - **CocktailResultDTO**
  - **OrdinationDTO**
  - **OrdinationResultDTO**
  - **OrderedCocktailDTO**
  - **TablesDTO**
  - **TablesResultDTO**
  - **MessageDTO**

I DTO elencati sono utilizzati per il passaggio di dati all'interno di richieste.

- Package **Utils**
  - **MailSender:** classe utilizzata per l'invio di una mail;
  - **FlavourEnum:** classe Enum utilizzata per indicare i possibili gusti che un cocktail può assumere;
  - **OrdinationStatusEnum**: classe Enum utilizzata per indicare i possibili stati che un'ordinazione può assumere;

2.3 Storage

Questo package contiene i seguenti sub-package e le seguenti classi:

- Package **Repository**
  - **UserAccountRepository**
  - **ProductRepository**
  - **IngredientRepository**
  - **CocktailRepository**
  - **TablesRepository**
  - **OrdinationRepository**
  - **OrderedCocktailRepository**
  - **MessageRepository**

I repository elencati sono classi che contengono l'implementazione dei metodi CRUD.

- Package **Entity**
  - **UserAccount:** classe che modella un dipendente che lavora all'interno del bar;
  - **Product:** classe che modella un prodotto utilizzato all'interno del bar;
  - **Ingredient:** classe che modella un ingrediente utilizzato all'interno del cocktail;
  - **Cocktail:** classe che modella un cocktail ordinabile all'interno del bar;
  - **Tables:** classe che modella tavoli all'interno del bar;
  - **Ordination:** classe che modella un'ordinazione;
  - **OrderedCocktail:** classe che modella una collezione di cocktail ordinati;
  - **Message:** classe che modella un messaggio collegato ad un'ordinazione;

3 Class Interfaces

Di seguito, vengono elencate le interfacce delle classi previste dal Sistema, che si trovano nel package "Controller" e "Repository". Per una questione di leggibilità e chiarezza del documento e per evitare di aggiungere informazioni ridondanti, non verranno mostrate le interfacce delle classi Repository (che forniscono i metodi CRUD per l'entità di cui si fanno carico), le classi Service che contengono le implementazioni delle funzionalità richieste dai Controller e i DTO che contengono i dati di scambio.

Per motivi di leggibilità si è deciso di sfruttare swagger-2 per documentare le API implementate. Sono state rese accessibili al seguente link: http://localhost:8090/swagger-ui.html quando è in esecuzione.

Classi nel Package Controller

Nome Classe	UserAccountController
Descrizione	Questa classe è il tramite per invocare le API collegate ad uno UserAccount.
Metodi	+ add (UserAccountDTO, HttpServletRequest): ResponseEntity<?> + delete (String , HttpServletRequest): ResponseEntity<?> + update (UserAccountDTO, HttpServletRequest): ResponseEntity<?> + login (UserAccountDTO, HttpServletRequest): ResponseEntity<?> + getByUuid (String, HttpServletRequest): ResponseEntity<?> + searchByFields (UserAccountDTO): ResponseEntity<?> + getAll (): ResponseEntity<?>
Invariante di classe	/

Nome Classe	ProductController
Descrizione	Questa classe è il tramite per invocare le API collegate ad un Product.
Metodi	+ add (ProductDTO, HttpServletRequest): ResponseEntity<?> + delete (String , HttpServletRequest): ResponseEntity<?> + update (ProductDTO, HttpServletRequest): ResponseEntity<?> + getByUuid (String, HttpServletRequest): ResponseEntity<?> + searchByFields (ProductDTO): ResponseEntity<?>
Invariante di classe	/

Nome Classe	IngredientController
Descrizione	Questa classe è il tramite per invocare le API collegate ad un Ingredient.
Metodi	+ add (IngredientDTO, HttpServletRequest): ResponseEntity<?> + delete (String , HttpServletRequest): ResponseEntity<?> + update (IngredientDTO, HttpServletRequest): ResponseEntity<?> + getByUuid (String, HttpServletRequest): ResponseEntity<?>
Invariante di classe	/

Nome Classe	CocktailController
Descrizione	Questa classe è il tramite per invocare le API collegate ad un Cocktail.
Metodi	+ add (CocktailDTO, HttpServletRequest): ResponseEntity<?> + delete (String , HttpServletRequest): ResponseEntity<?> + update (CocktailDTO, HttpServletRequest): ResponseEntity<?> + getByUuid (String, HttpServletRequest): ResponseEntity<?> + getIngredients(String): ResponseEntity<?> + searchByFields (CocktailDTO): ResponseEntity<?> + getMenu (CocktailDTO, HttpServletRequest): ResponseEntity<?> + getMenuMenu (CocktailDTO, HttpServletRequest): ResponseEntity<?>
Invariante di classe	/

Nome Classe	TablesController
Descrizione	Questa classe è il tramite per invocare le API collegate ad un tavolo.
Metodi	+ add (TableDTO, HttpServletRequest): ResponseEntity<?> + delete (String , HttpServletRequest): ResponseEntity<?> + update (TableDTO, HttpServletRequest): ResponseEntity<?> + getByUuid (String, HttpServletRequest): ResponseEntity<?> + getAll(): ResponseEntity<?> + getTableByOrderUuid(String): ResponseEntity<?>
Invariante di classe	/

Nome Classe	OrdinationController
Descrizione	Questa classe è il tramite per invocare le API collegate ad un ordinazione.
Metodi	+ create (OrdinationDTO, HttpServletRequest): ResponseEntity<?> + get(String, HttpServletRequest): ResponseEntity<?> + addCocktail(OrderedCocktailDTO, HttpServletRequest): ResponseEntity<?> + RemoveCocktail(OrderedCocktailDTO, HttpServletRequest): ResponseEntity<?> + updateStatus (OrdinationStatusEnum, MessageDTO, HttpServletRequest): ResponseEntity<?> + searchByFields (OrdinationDTO, HttpServletRequest): ResponseEntity<?> getOrdinationFromTable(String, HttpServletRequest)):

	ResponseEntity<?>
Invariante di classe	/

Nome Classe	**StatisticsController**
Descrizione	Questa classe è il tramite per invocare le API collegate alle statistiche.
Metodi	+ getByRoleUsers(String): ResponseEntity<?> + getCreatedBy(String): ResponseEntity<?> + getDeliveredBy(String): ResponseEntity<?> + getExecutedBy(String): ResponseEntity<?> + getBestSelling(Integer, HttpServletRequest)<?> + getBestSellingByFlavour(HttpServletRequest)<?>
Invariante di classe	/

Classi nel Package Repository

Nome Classe	**UserAccountRepository**
Descrizione	Questa classe che permette di recuperare informazioni inerenti allo UserAccount.
Metodi	+ findByEmail(String): List<UserAccount> + findByUuid(String): UserAccount
Invariante di classe	/

Nome Classe	**ProductRepository**
Descrizione	Questa classe che permette di recuperare informazioni inerenti ad un Prodotto.
Metodi	+ findByUuid(String uuid):Product + findByNameContainsIgnoreCase(String):List<Product> + findByNameContainsIgnoreCase(Pageable, String:List<Product> + countByNameContainingIgnoreCase(String name):int + findByCategoryContainsIgnoreCase(String):List<Product> + findByCategoryContainsIgnoreCase(Pageable, String):List<Product> + countByCategoryContainingIgnoreCase(String):int + findByNameAndCategoryContainsIgnoreCase(String,

	String):List<Product> +findByNameAndCategoryContainsIgnoreCase(Pageable,String,String):List<Product> + countByNameAndCategoryContainingIgnoreCase(String,String):int
Invariante di classe	/

Nome Classe	**IngredientRepository**
Descrizione	Questa classe che permette di recuperare informazioni inerenti ad un Ingrediente.
Metodi	+ findByUuid(String uuid):Ingredient + findByCocktailUuidAndProductUuid(String,String):Ingredient + deactivateFinischedIngredients(String):void
Invariante di classe	/

Nome Classe	**CocktailRepository**
Descrizione	Questa classe che permette di recuperare informazioni inerenti ad un Cocktail.
Metodi	+ findByUuid (String):Cocktail + findByNameContainsIgnoreCase(String):List<Cocktail> + findByNameContainsIgnoreCase(Pageable,String):List<Cocktail> + countByNameContainingIgnoreCase(String):int + findByFlavourContainsIgnoreCase(String):List<Cocktail> + findByFlavourContainsIgnoreCase(Pageable,String):List<Cocktail> + countByFlavourContainsIgnoreCase(String flavour):int + findByIsAlcoholic(boolean alcoholic):List<Cocktail> + findByIsAlcoholic(Pageable pageable, boolean alcoholic):List<Cocktail> + countByIsAlcoholic(boolean alcoholic):int +findByNameAndFlavourContainsIgnoreCase(String,String):List<Cocktail> +findByNameAndFlavourContainsIgnoreCase(Pageable,String,String):List<Cocktail> + countByNameAndFlavourContainingIgnoreCase(String,String):int + findByNameAndIsAlcoholic(String,boolean):List<Cocktail> + findByNameAndIsAlcoholic(Pageable,String,boolean):List<Cocktail> + countByNameAndIsAlcoholic(String,boolean):int; + findByFlavourAndIsAlcoholic(String flavour, boolean alcoholic):List<Cocktail> + findByFlavourAndIsAlcoholic(Pageable pageable, String flavour, boolean alcoholic):List<Cocktail> + countByFlavourAndIsAlcoholic(String flavour, boolean alcoholic):int + findByNameAndFlavourAndIsAlcoholic(String name, String flavour,

	boolean alcoholic):List<Cocktail> + List<Cocktail> findByNameAndFlavourAndIsAlcoholic(Pageable,String,String,boolean):List<Cocktail> + countByNameAndFlavourAndIsAlcoholic(String,String,boolean):int + findByInMenu(Pageable,boolean):List<Cocktail> + countByInMenu(boolean):int + findByIsIBA(Pageable pageable, boolean isIba):List<Cocktail> + countByIsIBA(boolean isIba):int + findAllByOrderBySoldDesc(Pageable pageable):List<Cocktail> + findTop5ByOrderBySoldDesc():List<Cocktail> + findBestSellingByFlavour():List<Integer>
Invariante di classe	/

Nome Classe	**TablesRepository**
Descrizione	Questa classe che permette di recuperare informazioni inerenti ad un Tavolo.
Metodi	+ findByUuid(String):Tables + findBynumber(Integer):Tables + findByOrderUuid(String):Tables + findByIsFree(Boolean):List<Tables> + findByIsFree(Pageable,Boolean):List<Tables> + countByIsFree(Boolean):int
Invariante di classe	/

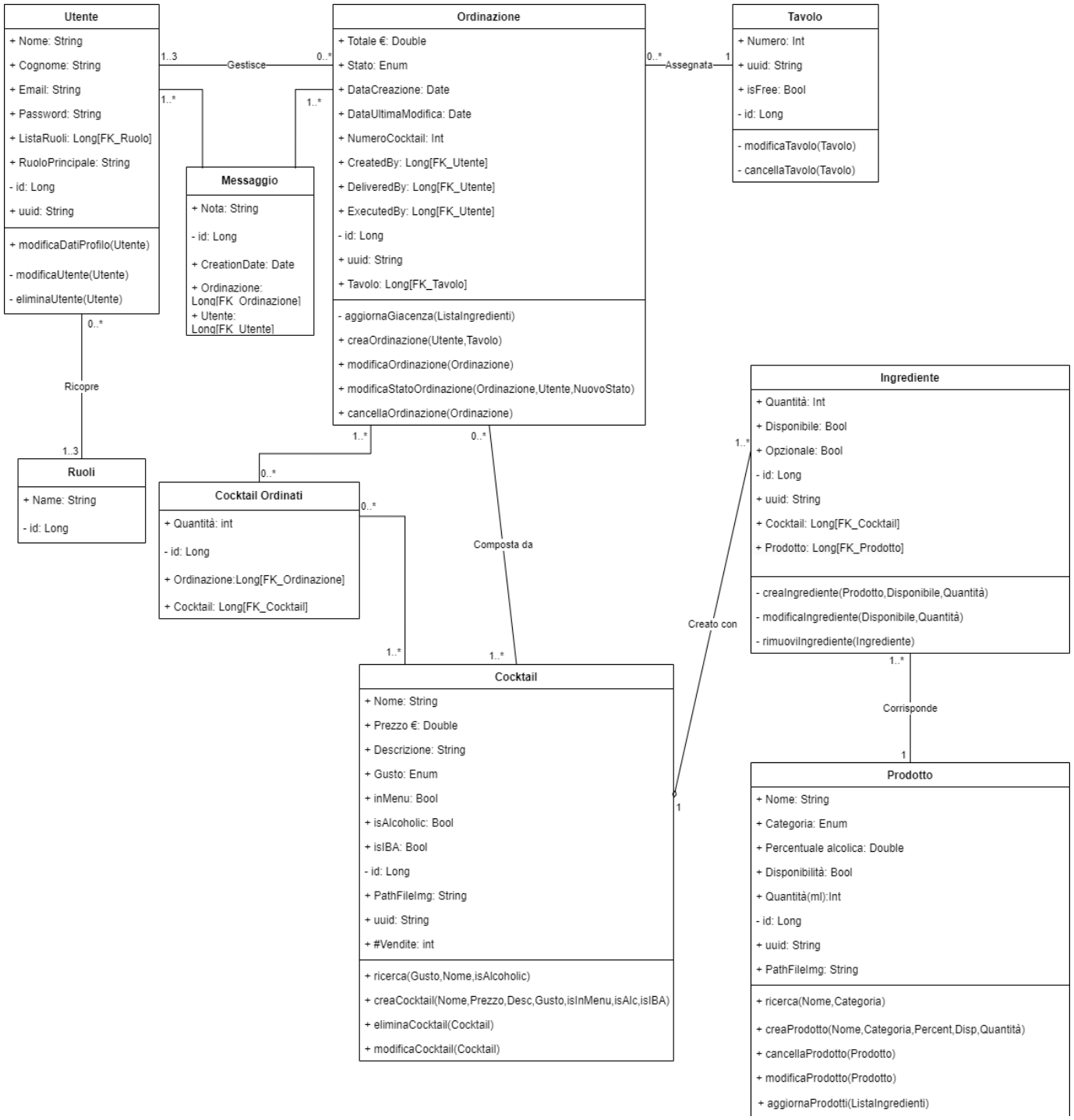
Nome Classe	**OrdinationRepository**
Descrizione	Questa classe che permette di recuperare informazioni inerenti ad un'Ordinazione.
Metodi	+ findByUuid(String):Ordination + findByCreatedBy(String):List<Ordination> + findByCreatedBy(Pageable,String):List<Ordination> + countByCreatedBy(String):int + findByDeliveredBy(String):List<Ordination> + findByDeliveredBy(Pageable,String):List<Ordination> + countByDeliveredBy(String):int + findByExecutedBy(String):List<Ordination> + findByExecutedBy(Pageable,String):List<Ordination> + countByExecutedBy(String):int + findByStatus(OrdinationStatusEnum):List<Ordination> + findByStatus(Pageable,OrdinationStatusEnum):List<Ordination> + countByStatus(OrdinationStatusEnum status):int +findOpenOrdinationForTableUuid(OrdinationStatusEnum,String):List<O

	rdination>
Invariante di classe	/

Nome Classe	**OrderedCocktailRepository**
Descrizione	Questa classe che permette di recuperare informazioni inerenti ai cocktail ordinati.
Metodi	+ findByOrderUuidAndCocktailUuid(String,String): OrderedCocktail
Invariante di classe	/

Nome Classe	**MessageRepository**
Descrizione	Questa classe che permette di recuperare informazioni inerenti ad un Messaggio su un'ordinazione.
Metodi	Composta da soli metodi basilari save e delete.
Invariante di classe	/

4 Class Diagram



5 Design Pattern

In questa sezione del presente documento si andranno a descrivere e dettagliare i design pattern utilizzati nello sviluppo della Web Application AlcoList.

Singleton

Il Singleton è un design pattern creazionale. Spring Boot utilizza il principio Inversion of Control (IoC), cioè processo in base al quale gli oggetti definiscono le loro dipendenze, ovvero gli altri oggetti con cui lavorano, solo tramite argomenti del costruttore, argomenti a un metodo factory o proprietà che vengono impostate sull'istanza dell'oggetto dopo che è stata costruita o restituita da un metodo factory. Il contenitore, quindi, inietta tali dipendenze quando crea il bean. Questo processo è fondamentalmente l'inverso, da cui il nome Inversion of Control (IoC), del bean stesso che controlla l'istanza o la posizione delle sue dipendenze utilizzando la costruzione diretta delle classi.
AlcoList deve garantire operazioni transazionali sull'ordinazione dei clienti, in modo da essere sicuri di avere la disponibilità dei prodotti e che due o più ordinazioni non concorrano per lo stesso prodotto.
Il sistema per garantire la transazionalità delle operazioni, crea un unico Singleton per gli oggetti Service. Sfruttando il principio dei Singleton in Spring Boot si ha una gestione delle dipendenze più efficiente poiché non verranno istanziate classi utilizzate precedentemente.

DTO

Il DTO (data transfer object) è un design pattern architetturale usato per trasferire dati tra sottosistemi di un'applicazione software. I DTO sono spesso usati in congiunzione con gli oggetti di accesso ai dati (Repository) per recuperare i suddetti da una base di dati con lo scopo di i ridurre il numero di chiamate ai metodi permettendo di passare molteplici parametri in una chiamata.
AlcoList deve garantire uno standard di comunicazione tra i vari layer del progetto riducendo il numero di parametri.
Per garantire questo standard, si definiscono implicitamente classi DTO condivise tra i vari layer per garantirne la comunicazione.

6 Glossario

In questa sezione descriveremo i termini tecnici che sono stati utilizzati all'interno del Documento stesso.

- **GitHub**: servizio di hosting per progetti software;
- **Web Application**: applicazione accessibile via Web per mezzo di una rete Internet.
- **CDN**: è una rete di server che distribuisce contenuti da un server di "origine" in tutto il mondo memorizzando nella cache i contenuti vicino a dove ogni utente finale accede a Internet tramite un dispositivo abilitato per il web.