



**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**

CORSO DI LAUREA MAGISTRALE DI INGEGNERIA INFORMATICA

Dipartimento di Ingegneria Gestionale, dell'Informazione e della
Produzione



ServeEasy

Progetto del corso di
Progettazione, Algoritmi e Computabilità

Prof.ssa
Patrizia Scandurra

Candidati:
Giorgio Chirico
1068142

Isaac Maffeis
1041473

Guyphard Ndombasi **Salvatore Salamone**
1092015 1096149

Anno accademico 2023-2024

Indice

Elenco delle figure	5
Elenco dei codici	10
1 Iterazione 0	11
1.1 Introduzione	11
1.2 Requisiti funzionali	12
1.2.1 Use case stories: Amministratore	12
1.2.2 Use case stories: Cliente	13
1.2.3 Use case stories: Cuoco	13
1.2.4 Use case stories: Cassiere	14
1.2.5 Priorità dei casi d'uso	14
1.2.6 Use case diagram	15
1.3 Requisiti non funzionali	17
1.3.1 Performance	17
1.3.2 Integrabilità	17
1.3.3 Modificabilità	17
1.3.4 Testabilità	17
1.3.5 Sicurezza	17
1.4 Topologia	18
1.5 Toolchain	19
1.5.1 Modellazione	19
1.5.2 Stack applicativo	19
1.5.3 Deployment	19
1.5.4 Gestore repository	19
1.5.5 Continuous Integration	19
1.5.6 Analisi statica	19
1.5.7 Analisi dinamica	20
1.5.8 Documentazione e organizzazione del team	20
1.5.9 Modello di sviluppo	20
2 Iterazione 1	21
2.1 Introduzione	21
2.2 Use Cases	22
2.2.1 Gruppo Sistema	22

2.2.2	Gruppo cliente	23
2.2.3	Gruppo cuoco	23
2.3	Component Diagram	24
2.3.1	Sistema ServeEasy	24
2.3.2	Gestione Comanda	25
2.3.3	Gestione Cliente	30
2.3.4	Gestione Cucina	33
2.4	Database	36
2.4.1	Modello Entità-Relazione	36
2.4.2	Modello logico	37
2.5	Deployment Diagram	39
2.6	Interface Class Diagram	40
2.6.1	Interfacce Gestione Comanda	40
2.6.2	Interfacce Gestione Cucina	41
2.6.3	Interfacce Gestione Cliente	42
2.7	Organizzazione del lavoro	43
2.7.1	Organizzazione del processo di sviluppo	43
2.7.2	Organizzazione dell'area di lavoro	44
2.8	Costruzione dello scheletro	47
2.8.1	Realizzazione database MariaDB	47
2.8.2	Definizione Interfacce con principi SOLID	47
2.8.3	Adattatore Kafka	50
2.8.4	Adattatore JPA	61
2.8.5	Continuous Integration	68
2.8.6	Continuous Delivery	70
2.8.7	DTO	72
2.8.8	Interfaccia di Test	76
2.8.9	Definizione Domain	87
2.9	Deployment dei microservizi con Docker	89
2.9.1	Healthcheck	89
2.9.2	Setup della rete di microservizi	89
2.9.3	Attivazione della rete di microservizi	90
2.10	Documentazione delle API	91
2.11	Analisi statica CheckStyle	109
2.11.1	Introduzione	109
2.11.2	Report Gestione Comanda	110
2.11.3	Report Gestione Cliente	112
2.11.4	Report Gestione Cucina	114

2.11.5 Specifica	116
2.11.6 Checkstyle	118
2.11.7 Generazione Grafi	123
2.12 Analisi statica SonarQube	126
2.12.1 SonarQube	126
2.12.2 Analisi report SonarQube	128
3 Iterazione 2	135
3.1 Introduzione	135
3.2 Organizzazione	136
3.2.1 Ordine	136
3.2.2 Cucina	136
3.2.3 Postazione	136
3.3 Funzione di priorità	137
3.3.1 Parametri	137
3.3.2 Pesi	138
3.3.3 Funzione matematica	140
3.4 Pseudocodice	141
3.4.1 Assegna valore di priorità:	141
3.4.2 Funzione di aggiornamento:	142
3.4.3 x1 ingrediente principale:	142
3.4.4 x2 tempo di preparazione:	143
3.4.5 x3 urgenza del cliente:	143
3.4.6 x4 numero ordine effettuato:	144
3.4.7 x5 tempo in attesa:	145
3.4.8 Gestore Code:	145
3.5 Struttura dati	147
3.5.1 Indexed priority queue	147
3.5.2 Coda (queue)	148
3.5.3 Diagramma di flusso	150
4 Iterazione 3	151
4.1 Introduzione	151
4.2 Gateway	152
4.2.1 Component Diagram	152
4.2.2 Deployment Diagram	152
4.3 Algoritmo	154
4.3.1 brainstorming	154

4.3.2	Adattamento struttura dati	155
4.3.3	Implementazione	155
4.3.4	Simulazione	157
4.4	Analisi dei Dati	161
4.4.1	Prestazioni algoritmo	161
4.4.2	Analisi dei thread	167

Elenco delle figure

1.1	Diagramma dei casi d'uso	16
1.2	Topologia del sistema	18
2.1	Casi d'uso presi in considerazione nell'iterazione 1	22
2.2	Component diagram - ServeEasy	24
2.3	Component diagram - System	24
2.4	Architettura esagonale per il microservizio Gestione comanda	26
2.5	Component diagram - Gestione Comanda	27
2.6	Component diagram - Gestione Comanda - Infrasructure	27
2.7	Component diagram - Gestione Comanda - Domain	28
2.8	Component diagram - Gestione Comanda - Interface	29
2.9	Component diagram - Gestione Cliente	30
2.10	Component diagram - Gestione Cliente - Infrastructure	31
2.11	Component diagram - Gestione Cliente - Domain	32
2.12	Component diagram - Gestione Cliente - Interface	32
2.13	Component diagram - Gestione Cucina	33
2.14	Component diagram - Gestione Cucina - Infrastructure	33
2.15	Component diagram - Gestione Cucina - Domain	34
2.16	Component diagram - Gestione Cucina - Interface	35
2.17	Modello Entità-relazione	36
2.18	Modello Logico	37
2.19	Deployment Diagram	39
2.20	Interface class diagram - Gestione Comanda	40
2.21	Interface class diagram - Gestione Cucina	41
2.22	Interface class diagram - Gestione Cliente	42
2.23	Organizzazione del lavoro cloud e locale, CI/CD e deployment	44
2.24	Ambiente di lavoro con IntelliJ IDEA e Docker Compose	45
2.25	source tree del progetto GestioneComanda	45
2.26	interfaccia grafica di PHPMyAdmin eseguito da container	47
2.27	Esempio funzionamento kafdrop	55
2.28	CI merge a pull-request	70
2.29	CI main check	70
2.30	Component diagram - Gestione Comanda - Interface con Test	77
2.31	Esempio applicativo Postman	87
2.32	Startup della rete, inizializzazione code in GestioneCucina	90

2.33 Sommario di gestione comanda	110
2.34 Rules generate da gestione comanda	110
2.35 Grafico di tutte le regole	111
2.36 Grafico delle info	111
2.37 Grafico degli warnings	111
2.38 Grafico degli errori	111
2.39 Sommario di gestione cliente	112
2.40 Rules generate da gestione cliente	112
2.41 Grafico di tutte le regole	113
2.42 Grafico delle info	113
2.43 Grafico degli warnings	113
2.44 Grafico degli errori	113
2.45 Sommario di gestione cucina	114
2.46 Rules generate da gestione cucina	114
2.47 Grafico di tutte le regole	115
2.48 Grafico delle info	115
2.49 Grafico degli warnings	115
2.50 Grafico degli errori	115
2.51 Schermata installazione plugin	119
2.52 Grafico warnings di gestione comanda	125
2.53 Analisys list gestione comanda	128
2.54 Coverage variata nel corso delle modifiche testate	130
2.55 diagramma di bole sulla secutity	130
2.56 diagramma di bole sui risk	131
2.57 overview issues	131
2.58 issues specifico	131
2.59 view singolo issue	132
2.60 gestione cucina overview	132
2.61 esempio caso uncoverage	132
2.62 java rules overview sonarqube	133
3.1 Strutture dati dell'algoritmo	147
3.2 Diagramma di flusso	150
4.1 Component Diagram con Gateway	152
4.2 Deployment Diagram con gateway	153
4.3 logico-struttura dell'algoritmo da implementare	154
4.4 associazione dizionario ed heap	155

4.5	Funzionamento Thread Cliente e Producer	157
4.6	Funzionamento Thread Checker	158
4.7	Funzionamento Thread Consumer e Producer	159
4.8	Funzionamento Thread Consumer	159
4.9	Funzionamento Thread Cuoco	160
4.10	File csv di report	162
4.11	Analisi Tempo di attesa in base al numero degli ordini	162
4.12	Distribuzione del Tempo di Attesa di preparazione degli Ordini	163
4.13	Distribuzione del Tempo di Attesa in coda degli Ordini	165
4.14	Distribuzione del Tempo di Attesa in coda degli Ordini per Numero progressivo	165
4.15	Analisi dei parametri in base al tempo totale in attesa	166
4.16	workflow dalla raccolta al display dei dati	167
4.17	grafo di Gantt	168
4.18	grafo cartesiano	168

Elenco dei codici

2.1	Query del database in SQL	38
2.2	port-binding del servizio <i>broker</i> alla voce <i>ports</i> (porta_host:porta_container)	46
2.3	Interfaccia MessagePort	48
2.4	Interfaccia DataPort	48
2.5	Interfaccia NotifyOrderEvent	49
2.6	Setup del docker-compose.yaml per l'adattatore Kafka	50
2.7	Aggiornamento dipendenze nel pom.xml per includere spring-kafka	51
2.8	Aggiornamento del file ‘application.yml‘ per il producer Kafka	51
2.9	Classe di configurazione KafkaConfig.java	52
2.10	Classe di configurazione JsonConfig.java	52
2.11	Classe del producer kafka CucinaPubProducer.java	53
2.12	Operazione di post sul topic kafka	54
2.13	Operazione di get sul topic kafka	54
2.14	Aggiornamento del docker-compose.yaml perl per Kafdrop	54
2.15	Aggiornamento dipendenze nel pom.xml per includere spring-kafka-test .	55
2.16	Aggiornamento del file ‘application.properties‘ di test per il producer kafka	55
2.17	Test di integrazione per il producer CucinaPubProducer	55
2.18	Aggiornamento del file ‘application.yml‘ per il consumer Kafka	57
2.19	Classe del consumer kafka SubClienteAdapter.java	57
2.20	Aggiornamento del file ‘application.properties‘ di test per il consumer Kafka	59
2.21	Test di integrazione per il consumer SubClienteAdapter	59
2.22	Aggiornamento dipendenze nel pom.xml per includere spring-data-jpa .	61
2.23	Aggiornamento del file ‘application.yml‘ per Spring Data JPA	62
2.24	Classe entità OrdineEntity.java	62
2.25	Classe repository OrdineRepository.java	64
2.26	Classe adattatore JPA JPADBAdapter.java	64
2.27	Aggiornamento del file pom.xml per la dipendenza di H2	66
2.28	Aggiornamento del file application.properties per i test con H2	66
2.29	Classe adattatore JPA JPADBAdapter.java	66
2.30	Creazione del file maven.yml	69
2.31	modifiche file maven.yml per CI/CD	71
2.32	Dockerfile di GestioneComanda	71
2.33	Aggiornamento dipendenze nel pom.xml per includere model-mapper .	72
2.34	Classe di configurazione MapperConfig.java	72
2.35	Classe DTO per l'entità ordine OrderDTO.java	73

2.36 Interfaccia Mapper.java	74
2.37 Classe OrdineMapper.java implementazione di ModelMapper.java	74
2.38 Test di integrazione OrdineMapperTests.java	75
2.39 Aggiornamento dipendenze nel pom.xml per includere spring-web	77
2.40 Interfaccia TestAPI.java	78
2.41 Classe REST Controller di test TestController.java	81
2.42 Classe Test per il REST Controller di test TestControllerTests.java	85
2.43 classe GestionePrioritàOrdini.java nel dominio di GestioneComanda	88
2.44 GestioneCucina con healthcheck, dipendenze e variabili sovrascritte	89
2.45 plugin Maven con Maven Checkstyle Plugin	119
2.46 Personalizzazioni regole di Sun Checkstyle	119
2.47 Avvio controllo checkstyle	122
2.48 Script Python - aggiunta checkstyle-result.xml	123
2.49 Script Python - parsing checkstyle-result.xml	123
2.50 Script Python - Plot pie chart	123
2.51 Python - installare i pacchetti necessari	124
2.52 Avvio script python	124
2.53 file checkstyle_warning_severity_counts.csv warning di gestione comanda	124
2.54 Implementazioni pom.xml	126
2.55 Avvio sonarqube	127

Iterazione 0

1.1 Introduzione

Il sistema che si intende realizzare per il caso di studio è un software gestionale per ottimizzare la gestione delle comande di un ristorante, migliorando l'esperienza dei clienti, la produttività della cucina e l'efficacia della cassa. Il sistema si baserà sull'utilizzo di tablet, che permettono ai commensali di ordinare i piatti desiderati, inserendo eventuali note, visualizzando lo stato degli ordini e richiedere il conto in modo semplice e veloce. La cucina riceve le comande tramite una dashboard dedicata, che le ordina secondo un algoritmo di priorità basato su diversi parametri, come il tempo trascorso dall'ordinazione, la volontà del cliente, la durata di preparazione del piatto e altri fattori. La cucina può anche notificare il completamento di un ordine, che verrà visualizzato sul tablet del tavolo corrispondente. L'operatore di cassa sarà in grado di visualizzare il sommario degli ordini e stampare a schermo una ricevuta al cliente. L'amministratore del ristorante può personalizzare la configurazione delle sale e dei menu, registrare i tavoli e gli account, e visualizzare delle statistiche sulle ordinazioni effettuate. Il sistema offre anche delle funzionalità opzionali, come la possibilità di far arrivare i piatti tutti insieme al tavolo, di allegare note agli ordini in preparazione, chiedere il conto al tavolo. Il sistema si propone quindi di rendere più agile e soddisfacente il servizio di ristorazione, sfruttando le potenzialità della tecnologia e gli alti rendimenti di un algoritmo apposito.

1.2 Requisiti funzionali

I requisiti funzionali sono stati esplicitati mediante le *use case stories*, considerando come attori coinvolti nel sistema:

- Amministratore;
- Cliente;
- Cuoco;
- Cassiere.

1.2.1 Use case stories: Amministratore

L'amministratore è un responsabile di sala, col compito di configurare il software nelle fasi di setup dell'attività.

CONFIGURAZIONE DISPOSITIVI SALA

- Come amministratore, voglio poter registrare i dispositivi destinati ai tavoli dei clienti per consentire ai commensali di accedere al sistema;
- Come amministratore, voglio poter registrare i dispositivi destinati alla cucina per permettere alla cucina di gestire gli ordini;
- Come amministratore, voglio poter registrare un dispositivo destinato al cassiere affinché sia possibile elencare al cliente la comanda che ha ordinato.

LOGIN/LOGOUT

- Come amministratore, voglio poter effettuare il log-in/log-out dal sistema.

CONFIGURAZIONE MENÙ

- Come amministratore, voglio poter effettuare una gestione del menù per visualizzare/modificare/aggiungere/eliminare portate;
- Come amministratore, voglio poter aggiungere/rimuovere/modificare gli ingredienti assegnati ad una portata per dettagliare la composizione.

1.2.2 Use case stories: Cliente

Il cliente può essere di due tipi: il cliente al tavolo, che usufruisce del dispositivo posto a disposizione dal ristorante, e il cliente da asporto, che comunica la sua ordinazione al ristorante tramite un portale sulla rete.

AUTENTICAZIONE

- Come cliente, voglio che il sistema riconosca i miei ordini così che possa elaborare le informazioni relative alla mia comanda;

VISUALIZZARE MENÙ

- Come cliente, voglio poter visualizzare il menù per decidere quale pietanza ordinare.

EFFETTUARE UN'ORDINAZIONE

- Come cliente, voglio effettuare un'ordinazione per ottenere una o più pietanze;
- Come cliente, voglio effettuare un'ordinazione personalizzando la pietanza desiderata per, ad esempio, togliere ingredienti non desiderati.

VISUALIZZARE STATO ORDINI

- Come cliente, voglio poter visualizzare lo stato di preparazione dei miei ordini per poter avere un feedback dalla cucina.

ANNULLARE UN ORDINE

- Come cliente, voglio annullare l'ordinazione di un piatto.

MODIFICARE UN ORDINE

- Come cliente al tavolo, voglio poter modificare un ordine già mandato verso la cucina per, ad esempio, precisare ingredienti da togliere, qualora l'ordine non fosse già in preparazione;
- Come cliente al tavolo, voglio poter modificare un ordine già mandato verso la cucina per, ad esempio, esigere il piatto prima (ad es., se il cliente ritiene di star aspettando troppo) o posticipare la sua preparazione.

1.2.3 Use case stories: Cuoco

Il terzo attore coinvolto è il cuoco che prepara le ordinazioni col supporto del sistema.

GESTIONE PREPARAZIONE ORDINI

- come cuoco, voglio poter gestire gli ordini effettuati dai clienti per poter eventualmente gestire la priorità di essi;
- come cuoco, voglio poter modificare lo stato di un piatto per avvertire il sistema di un'avvenuta preparazione.

VISUALIZZAZIONE LISTA ORDINI

- Come cuoco, voglio poter verificare lo stato degli ordini richiesti.

1.2.4 Use case stories: Cassiere

Il cassiere legge la comanda del cliente al fine di elencare le pietanze da lui ordinate, dettagliare informazioni annesse e calcolarne il conto.

VISUALIZZARE COMANDA

- Come cassiere, voglio visualizzare la comanda delle ordinazioni relativa a un determinato cliente per generare il conto.

GENERAZIONE CONTO

- Come cassiere, voglio poter generare il conto per un determinato cliente, per concludere la sua sessione nel sistema.

1.2.5 Priorità dei casi d'uso

Per ottimizzare il processo di sviluppo, si è deciso di categorizzare le specifiche funzionali in tabelle con tre livelli di priorità: elevata, media e bassa. Nello specifico il primo livello è assegnato alla Tabella 1.1 a cui sono attribuiti i casi d'uso essenziali per il funzionamento dell'applicazione, i casi d'uso relativi alle funzionalità aggiuntive non critiche sono stati attribuiti alla Tabella 1.2 a priorità media, mentre il livello a bassa priorità che accoglie requisiti funzionali opzionali previsti per versioni successive alla Tabella 1.3 .

PRIORITÀ ELEVATA

Codice	Titolo
UC1	Gestione comanda
UC2	Effettuare un'ordinazione
UC3	Visualizzare menù
UC4	Autenticazione
UC5	Visualizzare lista ordini
UC6	Gestione preparazione ordini

Tabella 1.1: Casi d'uso ad elevata priorità

PRIORITÀ MEDIA

Codice	Titolo
UC7	Configurazione dispositivi sala
UC8	Gestione dispositivi
UC9	Login amministratore
UC10	Logout amministratore
UC11	Configurazione menù
UC12	Gestione dati menù

Tabella 1.2: Casi d'uso a media priorità

PRIORITÀ BASSA

Codice	Titolo
UC13	Modifica un ordine
UC14	Annnullare un ordine
UC15	Visualizza stato delle ordinazioni
UC16	Generazione conto
UC17	Visualizzare comanda

Tabella 1.3: Casi d'uso a bassa priorità

1.2.6 Use case diagram

Dalla descrizione delle *use case stories*, è stato creato il diagramma UML dei casi d'uso in Figura 1.1, il quale è composto da 4 attori (Amministratore, Cuoco, Cassiere e Cliente) che tramite ereditarietà viene ridefinito in Cliente al tavolo oppure Cliente che effettua

ordinazioni d'asporto) e 6 viste (vista amministratore, vista cucina, vista cassiere, vista cliente, vista cliente al tavolo e sistema).

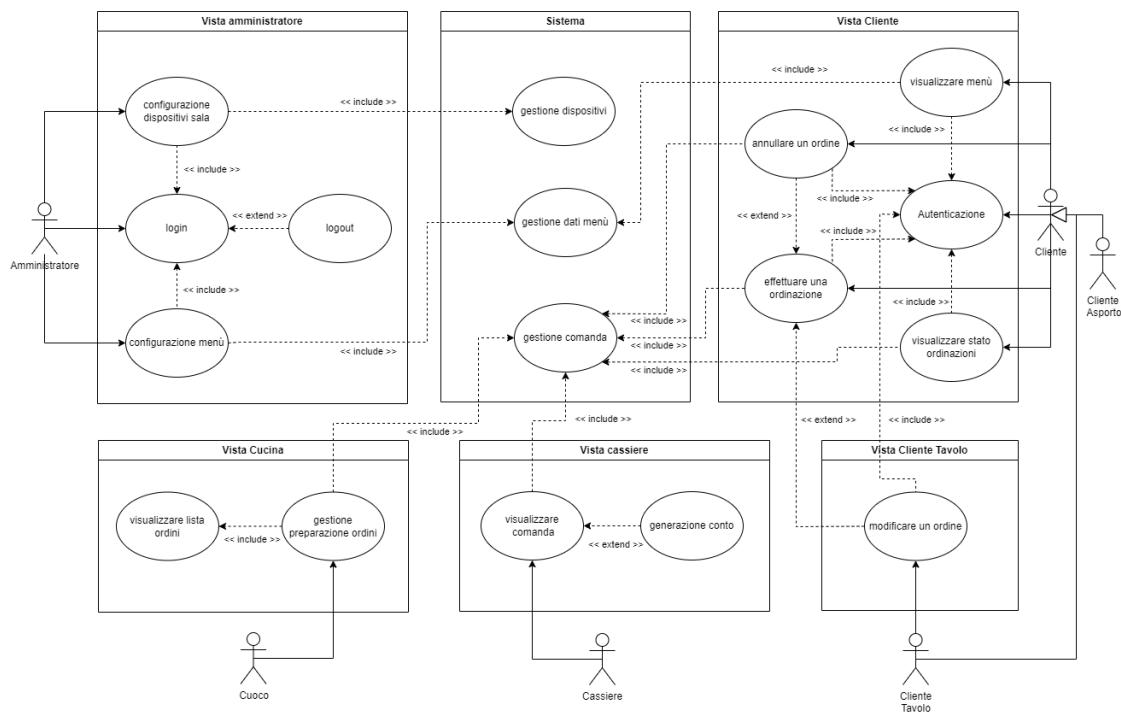


Figura 1.1: Diagramma dei casi d'uso

1.3 Requisiti non funzionali

Il progetto verrà sviluppato tenendo considerazione delle performance, integrabilità, modificabilità, testabilità e sicurezza dei componenti.

1.3.1 Performance

L'algoritmo di priorità impiegato dalla cucina per la selezione degli ordini deve fornire risultati in un tempo utile. Allo stesso tempo, gli utenti dell'applicativo web devono poter accedere e aggiornare le informazioni in un tempo accettabile.

1.3.2 Integrabilità

Ogni componente di sistema deve collaborare con gli altri componenti in modo da garantire le funzionalità previste dal sistema. Questa caratteristica è essenziale per garantire il corretto funzionamento e la coerenza dell'intero sistema.

1.3.3 Modificabilità

Il software deve facilitare l'aggiunta di nuovi componenti e funzionalità.

1.3.4 Testabilità

Ogni componente deve poter permettere la progettazione, implementazione ed esecuzione di test efficaci, in modo da garantire una massima copertura di requisiti e funzionalità.

1.3.5 Sicurezza

Il sistema deve integrare meccanismi di autenticazione ed autorizzazione degli attori, in modo da garantire la gestione delle identità, oltre alla protezione dei dati e delle API da accessi non autorizzati. Risulta dunque necessaria una distinzione dei ruoli con cui gli attori accedono al sistema.

1.4 Topologia

Per il progetto è stata adottata una topologia three-tier al fine di separare in tre livelli distinti la presentazione dei dati, la gestione dell'applicativo e la mappatura dei dati sui dispositivi di archiviazione. Come si può vedere dalla Figura 1.2 il servizio è esposto tramite un web server, al quale i dispositivi clienti accedono, tramite richiesta HTTP/REST, per mezzo di una API unificata, con funzionalità di gateway. Il web-server usufruirà di database relazionali per lo storage (Data Layer), mentre sarà supportato da un database in-memory H2 (Application Layer) per avvantaggiarsi di una ridondanza dati, allo scopo di aumentare le performance lato client.

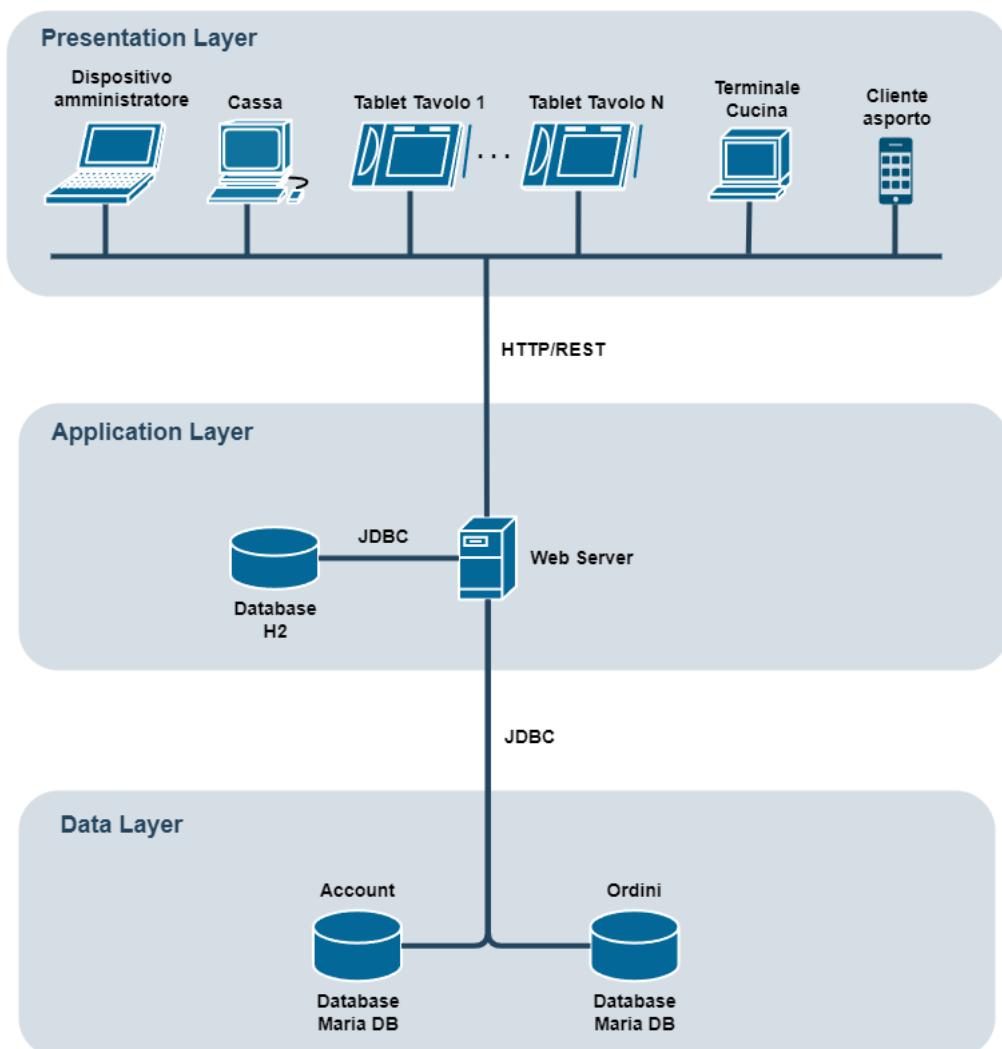


Figura 1.2: Topologia del sistema

1.5 Toolchain

Di seguito è presentata la toolchain utilizzata per lo sviluppo del progetto software

1.5.1 Modellazione

- draw.io: casi d'uso, topologia, schemi;

1.5.2 Stack applicativo

- Java Spring Boot 3.2.4: back-end;
- MariaDB: database per l'archiviazione;
- H2: database in-memory;
- Apache Kafka: piattaforma di streaming dati distribuita per la gestione degli eventi;
- Python: data analysis;

1.5.3 Deployment

- Docker: piattaforma per container virtuali;
- Docker Compose: gestione app multi-container;

1.5.4 Gestore repository

- Git: controllo versione per codice sorgente;
- GitHub: piattaforma hosting e collaborativa per progetti Git;

1.5.5 Continuous Integration

- Maven: gestore di progetti e dipendenze Java;
- GitHub Action: piattaforma di automazione per repository GitHub;

1.5.6 Analisi statica

- Checkstyle: visualizzazione di alto livello di metriche qualitative del codice;

1.5.7 Analisi dinamica

- Postman: strumento per testare API e servizi;
- JUnit5 (Jupiter): framework per test unitari Java, integrato in Spring Boot;

1.5.8 Documentazione e organizzazione del team

- Google Drive: servizio cloud per archiviazione;
- Documenti condivisi di Google: per elaborare la documentazione in modo condiviso;
- L^AT_EX: generazione documentazione;
- Overleaf: editor online per linguaggio L^AT_EX;
- IntelliJ Code With Me: collaborazione in tempo reale su codice;
- Google colab: piattaforma cloud di programmazione collaborativa per data analysis;
- Microsoft Teams: per organizzazione e meeting;

1.5.9 Modello di sviluppo

Il modello adottato segue la filosofia AGILE, con enfasi sui seguenti aspetti-chiave:

- pair programming, per favorire creatività e controllo del lavoro prodotto;
- orientamento al risultato, con enfasi maggiore sulla generazione di codice funzionante e componenti completi prima della relativa documentazione;
- rapidità di risposta ai cambiamenti;
- collaborazione attiva col cliente, al fine di incontrare le sue necessità, garantire trasparenza e fornire feedback tempestivo sul lavoro di progetto;
- Proattività nell'identificazione e mitigazione dei rischi.

Iterazione 1

2.1 Introduzione

Nella Iterazione 1 si sono presi i casi d'uso a più alta priorità e ci si è focalizzati allo sviluppo della architettura software e del database. Si è adottato un approccio di good design, puntando a un sistema software di alta qualità, mantenibile e scalabile, con componenti modulari e codice chiaro. Parallelamente, si è perseguito il principio di coesione funzionale, assicurando che funzioni correlate fossero raggruppate per formare moduli coesi, migliorando così manutenibilità e testabilità del sistema. E' stato quindi eseguito un lavoro di analisi e decomposizione del problema seguendo delle euristiche di early design, riunendo gli use cases in gruppi a cui potessero essere associati dei subsystem ben delineati. L'applicazione delle euristiche assume che la progettazione della soluzione si baserà su un'architettura a microservizi.

2.2 Use Cases

Si sono presi in considerazione i seguenti casi d'uso, ossia quelli a priorità più elevata della Tabella 1.1

Codice	Titolo
UC1	Gestione comanda
UC2	Effettuare un'ordinazione
UC3	Visualizzare menù
UC4	Autenticazione
UC5	Visualizzare lista ordini
UC6	Gestione preparazione ordini

Tabella 2.1: Casi d'uso presi in considerazione nell'iterazione 1

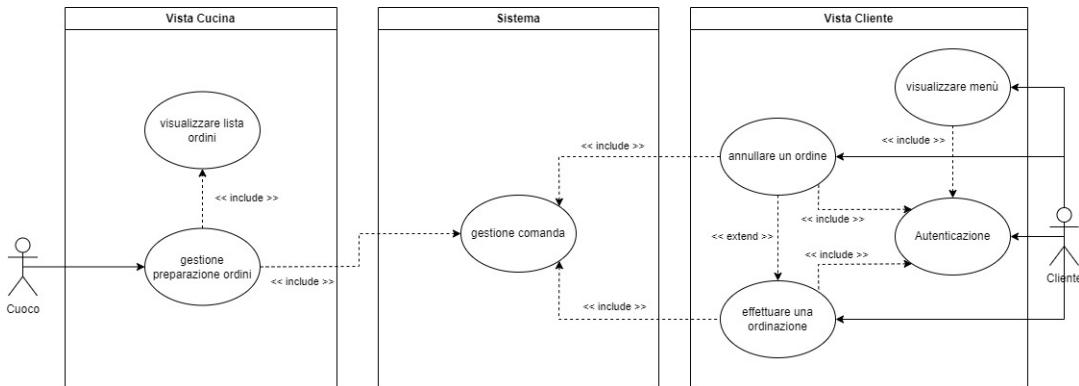


Figura 2.1: Casi d'uso presi in considerazione nell'iterazione 1

Per facilitare una migliore organizzazione e comprensione del sistema, i casi d'uso vengono raggruppati nel seguente modo:

2.2.1 Gruppo Sistema

UC-1 “Gestione Comanda”:

- UC-1.1 : gestione priorità ordine
ogni ordine è caratterizzato da una priorità
- UC-1.2 : gestione coda ordini
ogni ordine è inserito in una coda ordini
- UC-1.3 : assegnazione ordini comanda
ogni ordine deve essere associato ad una comanda

- UC-1.4 : assegnazione comanda cliente
ogni comanda deve essere associata ad un cliente

2.2.2 Gruppo cliente

UC-2 “effettuare un’ordinazione”:

- UC-2.1 : effettuare un ordine personalizzato
il cliente può effettuare un ordine escludendo un ingrediente o descrivendo una variazione del piatto

UC-3 “Visualizzare menu”:

- UC-3.1 : visualizzare piatto
- UC-3.2 : visualizzare informazioni piatto
il cliente deve poter leggere breve descrizione, ingredienti, prezzo

UC-4 “Autenticazione”:

- UC-4.1 : identificazione sessione cliente
al momento del pasto e solo per il pasto, il cliente deve poter distinguere la propria comanda

2.2.3 Gruppo cuoco

UC-5 “visualizzare lista ordini”:

- UC-5.1 : visualizzazione ordini per postazione
il cuoco deve visualizzare gli ordini destinati alla sua postazione

UC-6 “gestione preparazione ordini”:

- UC-6.1 : notifica preparazione ordine
il cuoco deve segnalare la presa in carico dell’ordine
- UC-6.2 : notifica completamento ordine
il cuoco deve segnalare il completamento dell’ordine così da passare al successivo
- UC-6.3 : gestione priorità postazione
il cuoco può modificare la priorità di un certo ingrediente così da ridurre la pressione su una certa postazione o, viceversa, per aumentarne il traffico. In tal modo può manualmente agire sulla gestione del traffico verso la cucina.

2.3 Component Diagram

2.3.1 Sistema ServeEasy

single component del sistema

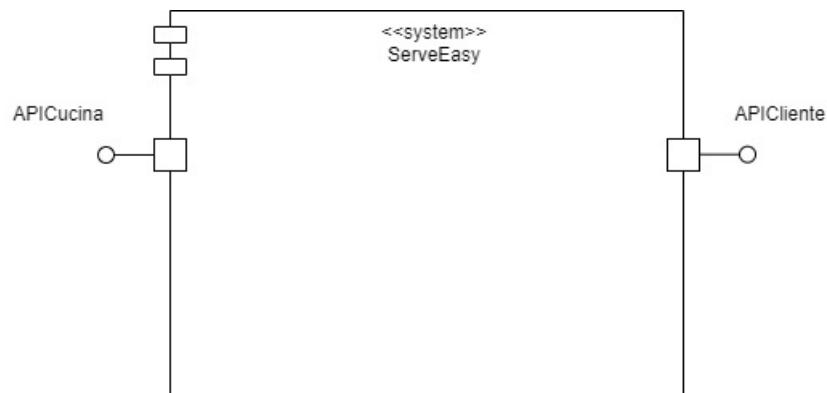


Figura 2.2: Component diagram - ServeEasy

Visualizzazione iniziale della soluzione come un componente unico che espone due API, dedicate rispettivamente alla cucina ed ai clienti. Si procede con uno sviluppo top-down.

primo zoom-in sul sistema

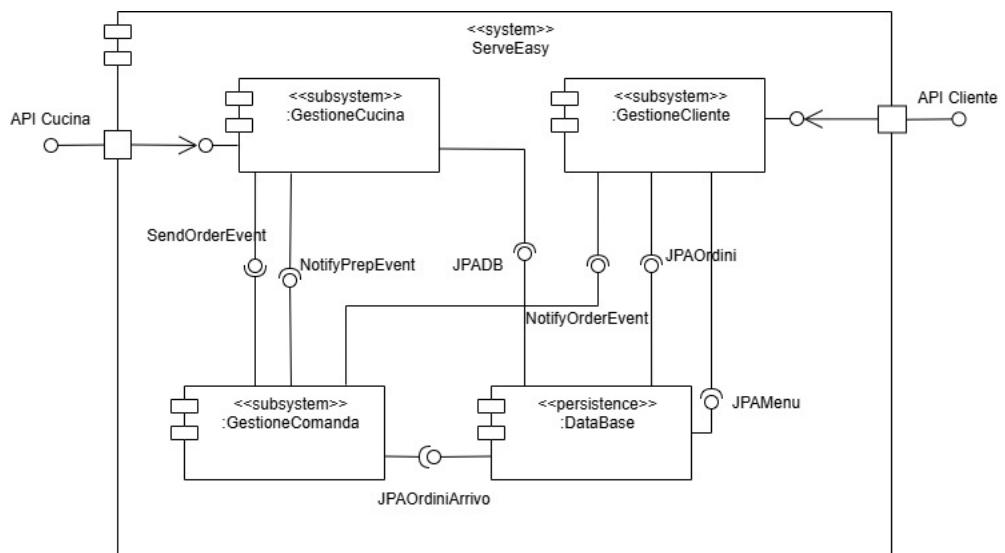


Figura 2.3: Component diagram - System

Al primo zoom-in si identificano i servizi che andranno a comporre l'architettura della soluzione:

- **GestioneComanda:** risolve gli use case del gruppo “sistema”, rappresenta il cuore del sistema ed incorpora la logica di backend fondamentale per la gestione regolarizzata degli ordini da cliente a cucina, attraverso politiche di schedulazione a priorità progettate ed implementate con un algoritmo ad-hoc.
- **GestioneCliente:** risolve gli use case del gruppo “cliente”, espone le funzionalità destinate ai dispositivi di tavolo ed al portale web per clienti d’asporto. Ha dunque il compito di gestire gli aspetti del servizio legati alle interazioni del cliente col sistema, come la visualizzazione del menu, la creazione degli ordini ed il raggruppamento degli ordini in una comanda relativa.
- **GestioneCucina:** risolve gli use case del gruppo “cucina”, espone le chiamate destinate ai dispositivi di cucina. Questo servizio conterrà un sistema a code, dove l’ordine in arrivo verrà classificato ed inserito in base al suo ingrediente principale. Gli ordini verranno gestiti dalle postazioni della cucina seguendo una politica FIFO.

Per la memorizzazione persistente dei dati cruciali per l’attività come piatti, ordini e comande, è stato inserito un componente database. All’interno del sistema ServeEasy, i componenti comunicano tra loro attraverso una comunicazione ad eventi, asincrona. Si è deciso di attuare una politica pub-sub per la gestione delle comunicazioni interne, costituite da scambi di notifiche e DTO tra i microservizi designati.

2.3.2 Gestione Comanda

Componenti esagonali

Il design dei microservizi seguirà l’architettura esagonale: un dominio, denominato “Domain”, nucleo della logica di servizio, sarà racchiuso tra due gusci denominati “Interface” e “Infrastructure”, i quali avranno il compito di astrarre la gestione dati, rendendola opaca al dominio. La logica di base del microservizio seguirà lo schema port-adapter, dove il dominio comunica con i gusci attraverso delle interfacce dette porte (il cui nome nel progetto è caratterizzato dal suffisso “Port”), mentre i gusci hanno il compito di implementare l’effettivo componente di trasmissione (guscio Infrastructure) e/o ricezione (guscio Interface), detto adattatore (sarà identificabile da suffisso “Adapter”). Nello specifico il **Domain** definisce gli oggetti, le entità e le operazioni che sono pertinenti al problema che il microservizio gestisce. Gli **Interface adapters** fungono da ponte tra il mondo esterno e il core del sistema, consentendo al microservizio di comunicare con altre applicazioni, servizi o dispositivi esterni in modo indipendente dall’implementazione interna del sistema stesso, mentre gli **Infrastructure adapters** fungono da ponte tra il core del sistema e l’infrastruttura esterna, gestendo le chiamate e le operazioni necessarie per accedere e

utilizzare le risorse infrastrutturali. Tali caratteristiche avvantaggiano l'intercambiabilità dei singoli componenti di sistema a costo di un aumento della complessità.

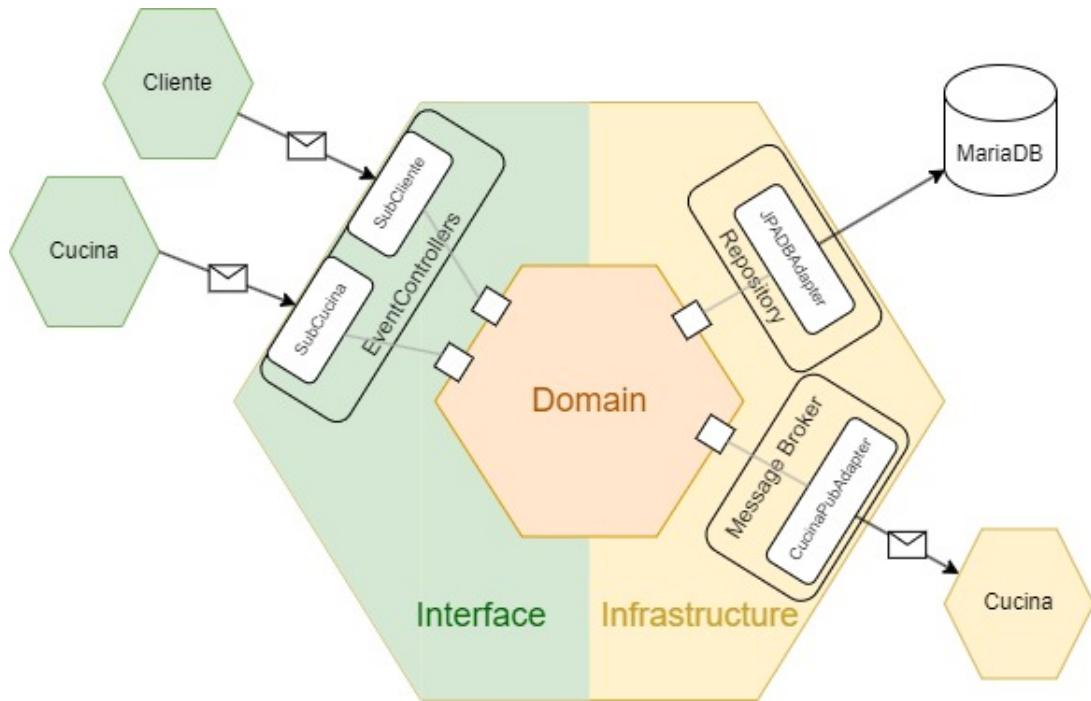
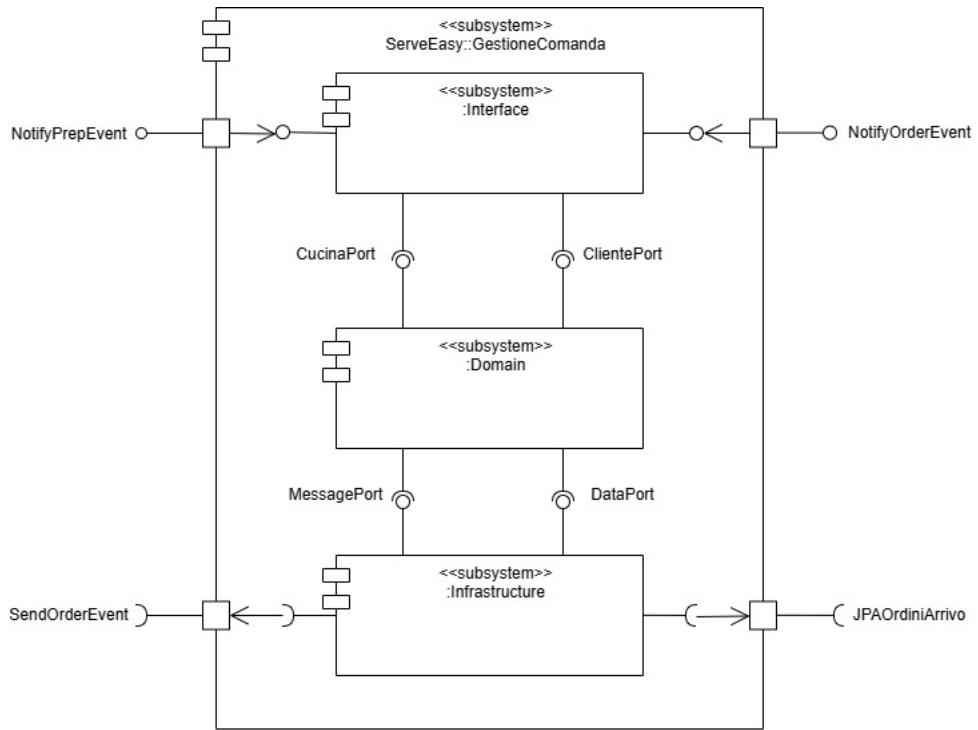
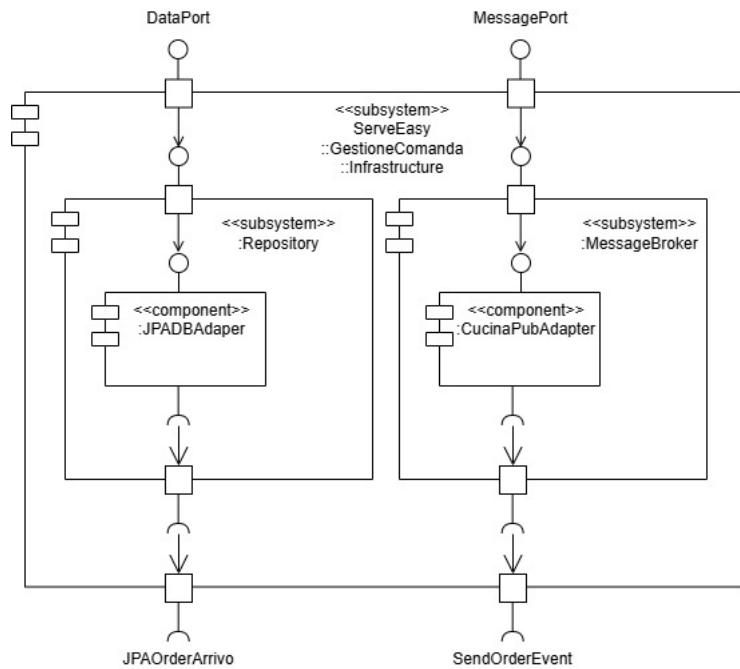


Figura 2.4: Architettura esagonale per il microservizio Gestione comanda

zoom-in gestione comanda**Figura 2.5:** Component diagram - Gestione Comanda**zoom-in infrastruttura di gestione comanda****Figura 2.6:** Component diagram - Gestione Comanda - Infrastruttura

- Repository: JPADBAdapter per la comunicazione con il database;
- MessageBroker: CucinaPubAdapter per l'invio di messaggi sul topic verso il microservizio della cucina.

zoom-in domain di gestione comanda

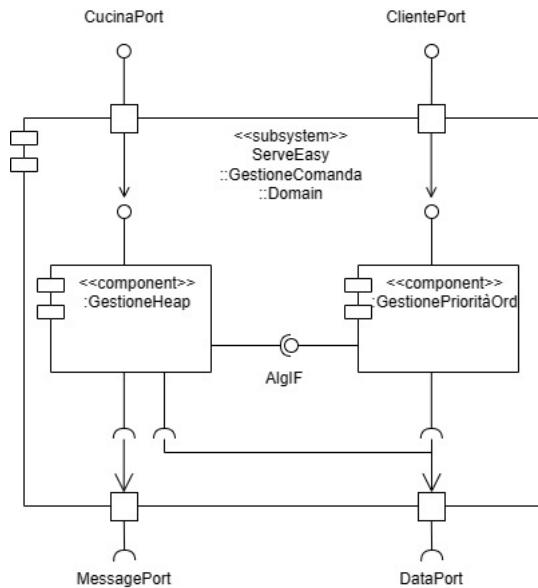
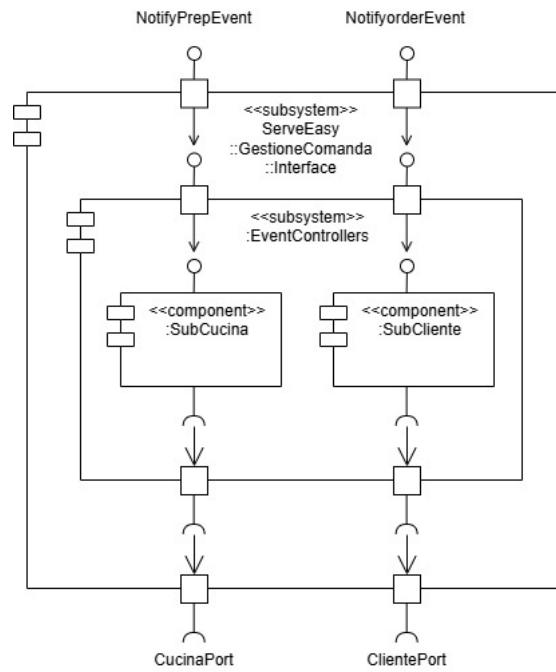


Figura 2.7: Component diagram - Gestione Comanda - Domain

zoom-in interface di gestione comanda**Figura 2.8:** Component diagram - Gestione Comanda - Interface

- EventControllers: SubCucina e SubCliente, permettono la ricezione di messaggi tramite message broker dagli altri microservizi.

2.3.3 Gestione Cliente

zoom-in gestione cliente

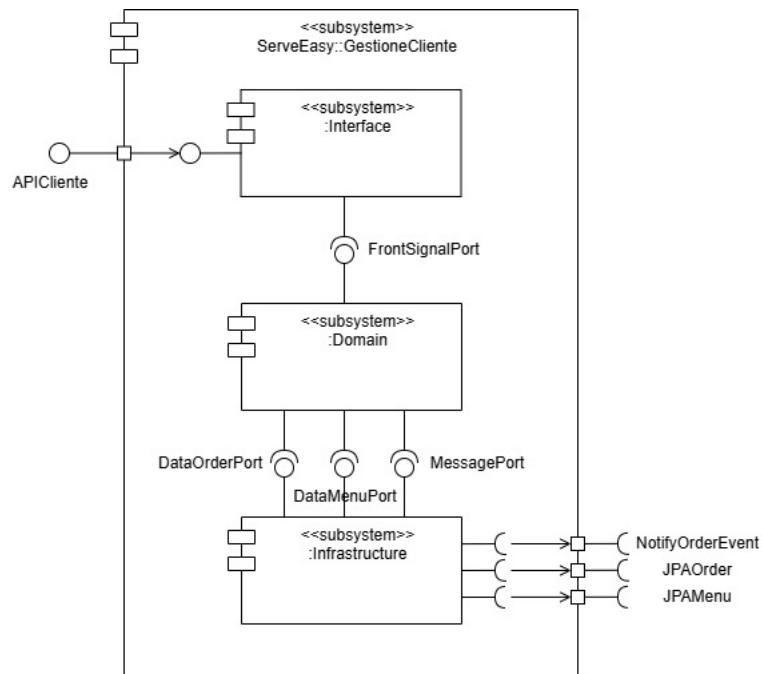


Figura 2.9: Component diagram - Gestione Cliente

zoom-in infrastructure di gestione cliente

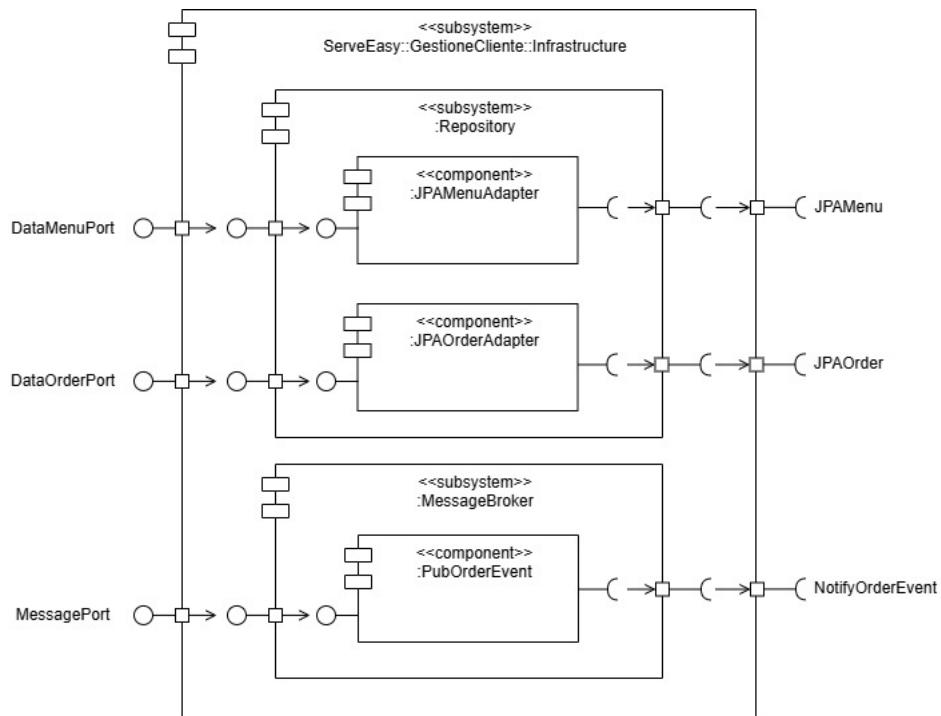


Figura 2.10: Component diagram - Gestione Cliente - Infrastructure

- Repository: `JPAOrderAdapter` e `JPAMenuAdapter` per la comunicazione con il database;
- MessageBroker: `PubOrderEvent` per l'invio di messaggi sul topic verso il microservizio della cucina.

zoom-in domain di gestione cliente

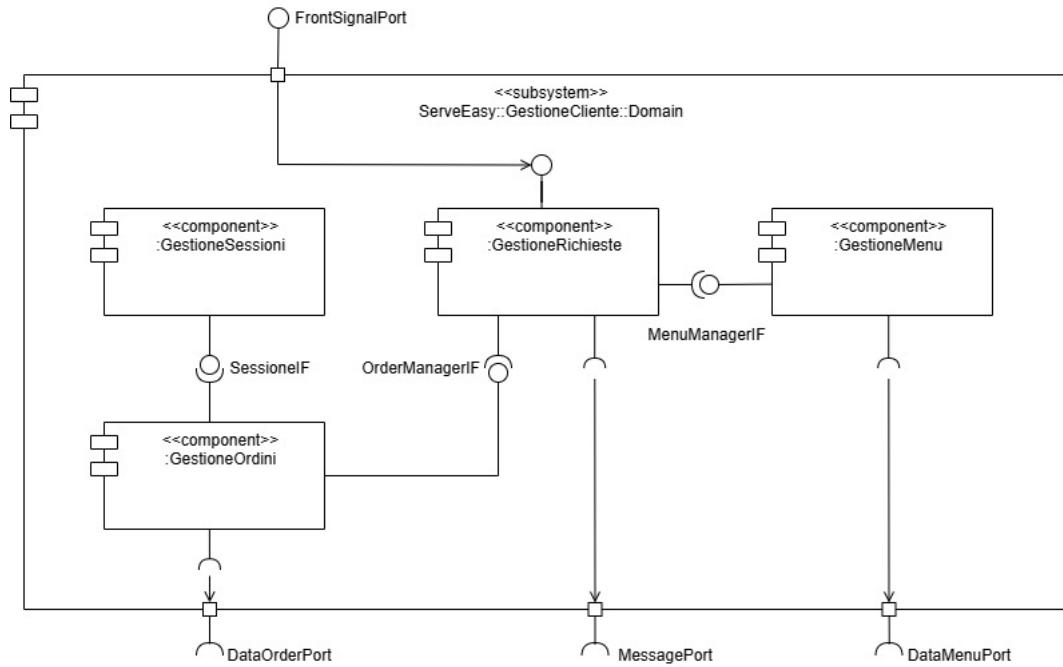


Figura 2.11: Component diagram - Gestione Cliente - Domain

zoom-in interface di gestione cliente

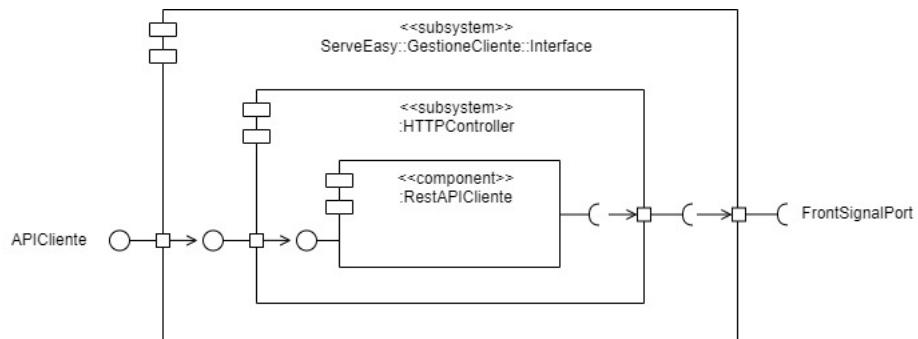


Figura 2.12: Component diagram - Gestione Cliente - Interface

- `HTTPControllers`: `RestApiClient`, permette di esporre API verso l'esterno.

2.3.4 Gestione Cucina

zoom-in gestione cucina

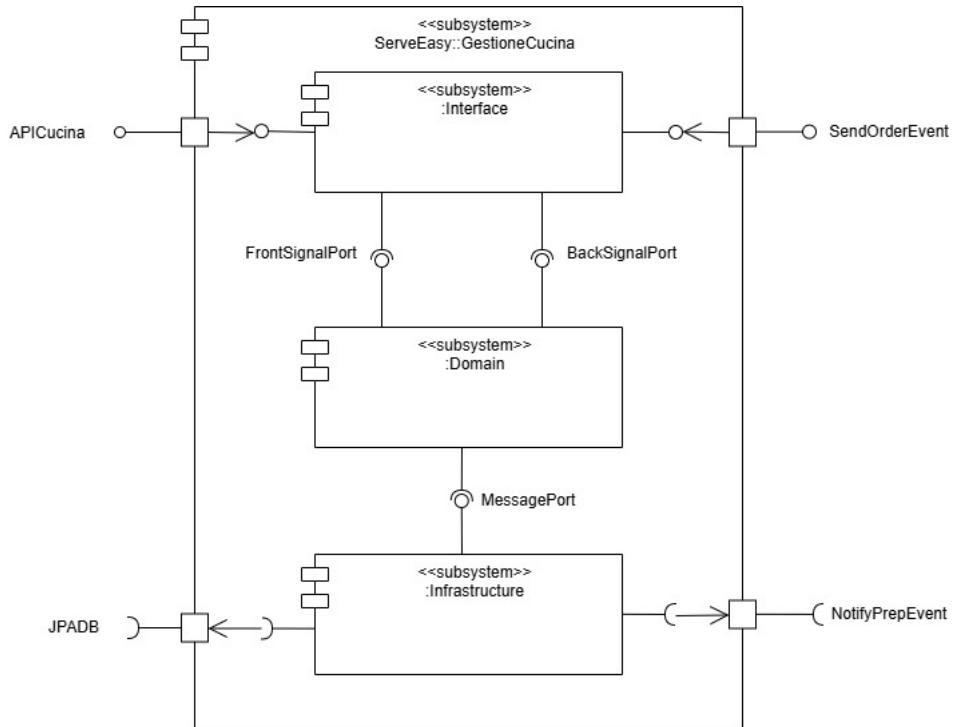


Figura 2.13: Component diagram - Gestione Cucina

zoom-in infrastruttura di gestione cucina

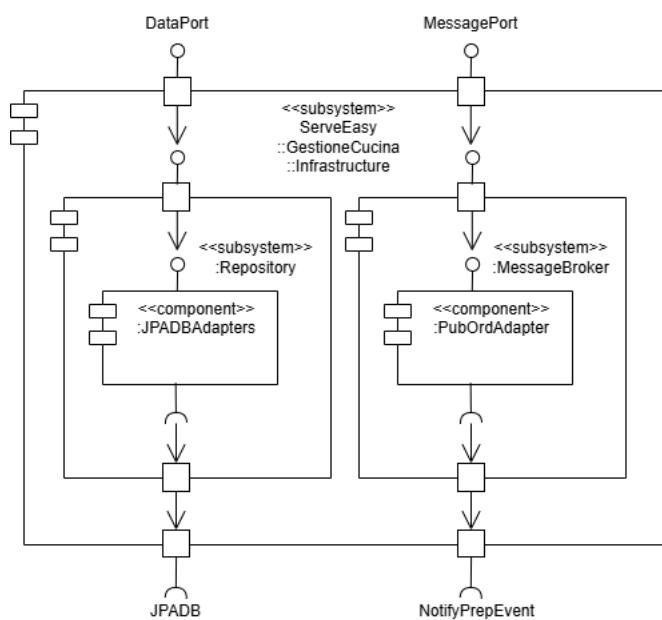


Figura 2.14: Component diagram - Gestione Cucina - Infrastructure

- Repository: JPADBAdapter per la comunicazione con il database;
- MessageBroker: PubOrderAdapter per l'invio di messaggi sul topic verso il micro-servizio di GestioneComanda.

zoom-in domain di gestione cucina

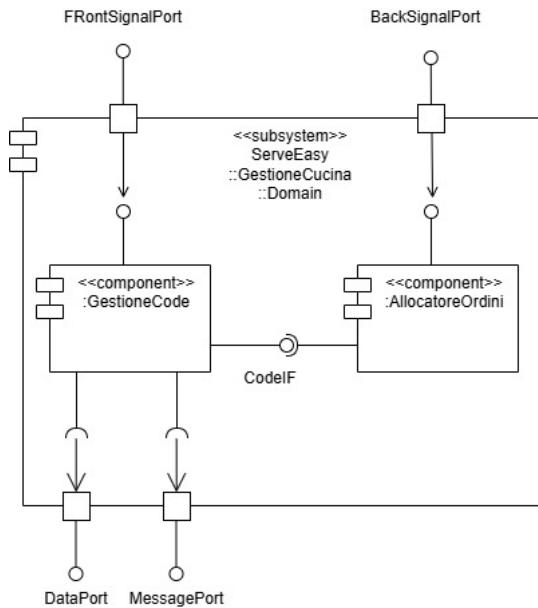
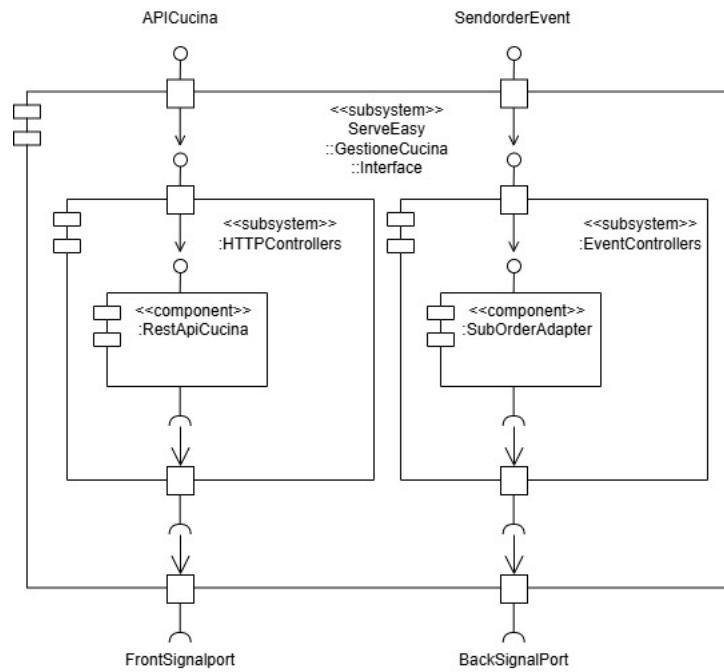


Figura 2.15: Component diagram - Gestion Cucina - Domain

zoom-in interface di gestione cucina**Figura 2.16:** Component diagram - Gestione Cucina - Interface

- EventControllers: SubOrderAdapter, permette la ricezione di messaggi tramite message broker dal microservizio GestioneComanda;
- HTTPControllers: RestApiCucina, permette di esporre API verso l'esterno.

2.4 Database

Per facilitare l'identificazione delle entità coinvolte nel database si è utilizzato un modello entità-relazione che fornisce una rappresentazione grafica chiara e intuitiva della struttura dei dati. Questo modello aiuta a visualizzare le entità (oggetti o concetti del mondo reale), le relazioni (le associazioni tra le entità) e gli attributi (le proprietà o le caratteristiche delle entità e delle relazioni).

2.4.1 Modello Entità-Relazione

Nel seguente diagramma entità-relazione in Figura 2.17, osserviamo che le comande possono essere costituite da più ordini effettuati dai clienti. Tali clienti sono suddivisi in due categorie: clienti d'asporto identificati tramite numero di telefono e clienti al tavolo identificati tramite numero del tavolo. I piatti, consultabili tramite un menù, sono caratterizzati da un ingrediente principale. Una volta ordinato un piatto dal menù, questo viene inserito al'interno di un ordine identificato da un codice progressivo per cliente, e viene successivamente inserito nella comanda del rispettivo cliente. La comanda sarà quindi utilizzata per identificare il cliente e contiene i piatti ordinati oltre che il totale dello scontrino con il corrispettivo codice di pagamento.

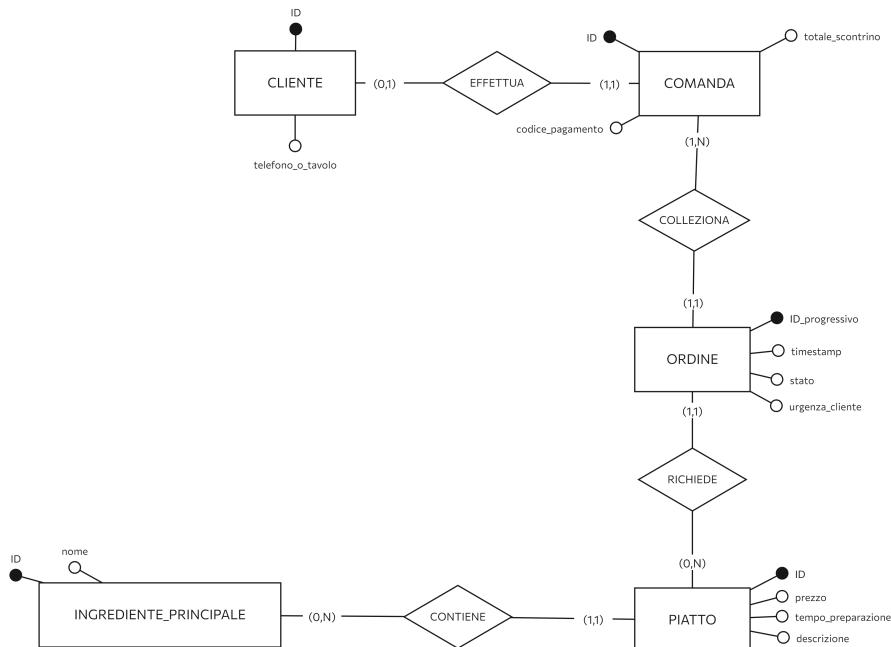


Figura 2.17: Modello Entità-relazione

2.4.2 Modello logico

Tramite il modello logico viene rappresentata in modo astratto la struttura dei dati così da facilitare la progettazione del database, definendo come i dati sono organizzati e come le entità interagiscono tra loro. Rappresentazione della struttura dei dati all'interno del database. L'attributo di cliente::asporto_o_tavolo è stato pensato come un boolean in quanto il cliente può essere di due tipi:

- se asporto_o_tavolo = 0, allora l'ID sarà il codice identificativo di un tavolo;
- se asporto_o_tavolo = 1, allora l'ID sarà un numero di telefono;

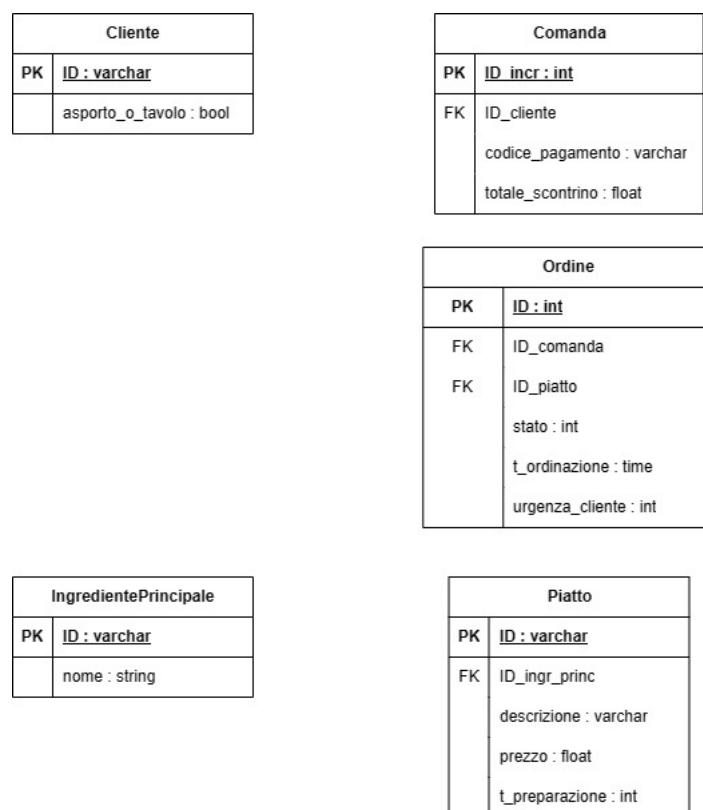


Figura 2.18: Modello Logico

Il modello logico è implementato con le seguenti query al database:

```

1 CREATE TABLE IF NOT EXISTS Cliente(
2     ID varchar(10) PRIMARY KEY,
3     t_o_a boolean NOT NULL
4 );
5
6 CREATE TABLE IF NOT EXISTS Comanda (
7     ID int(10) AUTO_INCREMENT ,
8     ID_cliente varchar(10) NOT NULL ,
9     codice_pagamento varchar(255) DEFAULT NULL ,
10    totale_scontrino float DEFAULT 0.0 ,
11    PRIMARY KEY (ID),
12    FOREIGN KEY (ID_cliente) REFERENCES Cliente(ID)
13 );
14
15 CREATE TABLE IF NOT EXISTS IngredientePrincipale(
16     ID varchar(20) primary key ,
17     nome varchar(20) not NULL
18 );
19
20 CREATE TABLE IF NOT EXISTS Piatto(
21     ID varchar(20) NOT NULL PRIMARY KEY ,
22     ID_ingr_princ varchar(20) NOT NULL ,
23     descrizione varchar(50) ,
24     prezzo float(6) NOT NULL ,
25     t_preparazione int ,
26     FOREIGN KEY (ID_ingr_princ) REFERENCES IngredientePrincipale(ID)
27 );
28
29 CREATE TABLE IF NOT EXISTS Ordine(
30     ID int(10) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
31     ID_comanda int(10) NOT NULL ,
32     ID_piatto varchar(20) NOT NULL ,
33     stato int(1) DEFAULT 0 , -- 0=creato , 1=in coda , 2=in preparazione ,
34     3=completato
35     t_ordinazione TIMESTAMP DEFAULT CURRENT_TIMESTAMP ,
36     urgenza_cliente int(2) DEFAULT 0 , -- priorita' del cliente: 1=
37     massima , -1=minima
38     FOREIGN KEY (ID_comanda) REFERENCES Comanda(ID) ,
39     FOREIGN KEY (ID_piatto) REFERENCES Piatto(ID) ,
40     CHECK (stato >= 0 AND stato <= 3 )
41 );

```

Codice 2.1: Query del database in SQL

2.5 Deployment Diagram

Il diagramma mostra le istanze usate nella rete di container Docker adottata per creare l'applicazione e le loro relazioni. In particolare, si è scelto di implementare un'architettura event-driven con Kafka e Zookeeper su singolo host server.

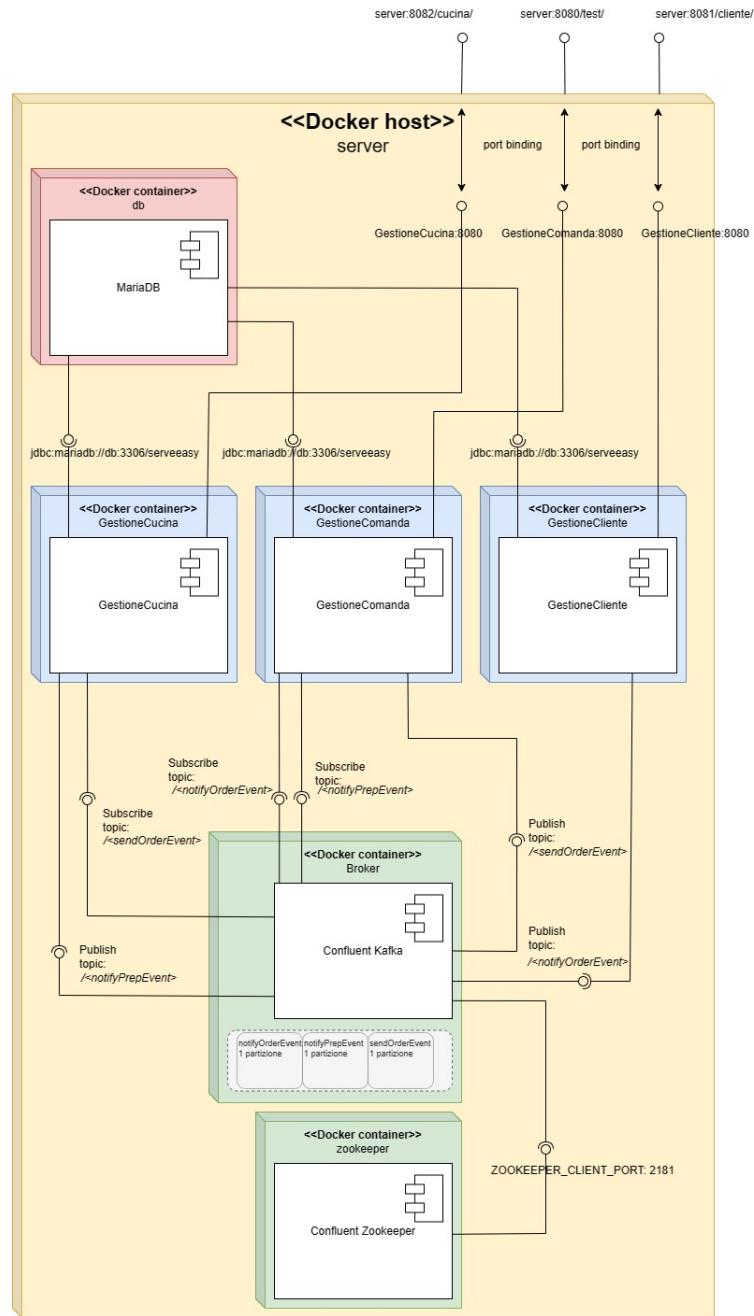


Figura 2.19: Deployment Diagram

2.6 Interface Class Diagram

In questa sezione si definiscono in alto livello le interfacce, con relativi metodi, per la comunicazione tra i componenti e sottosistemi ottenuti. In particolare, vengono definiti con lo stereotipo **signal** i canali di comunicazione ad eventi, designati nel progetto per seguire un pattern publisher-subscriber.

2.6.1 Interfacce Gestione Comanda

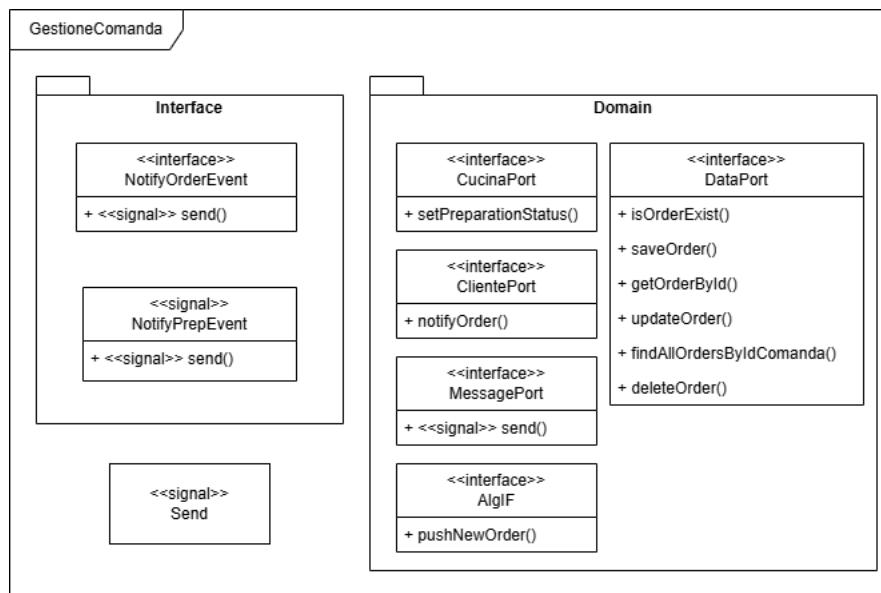


Figura 2.20: Interface class diagram - Gestione Comanda

2.6.2 Interfacce Gestione Cucina

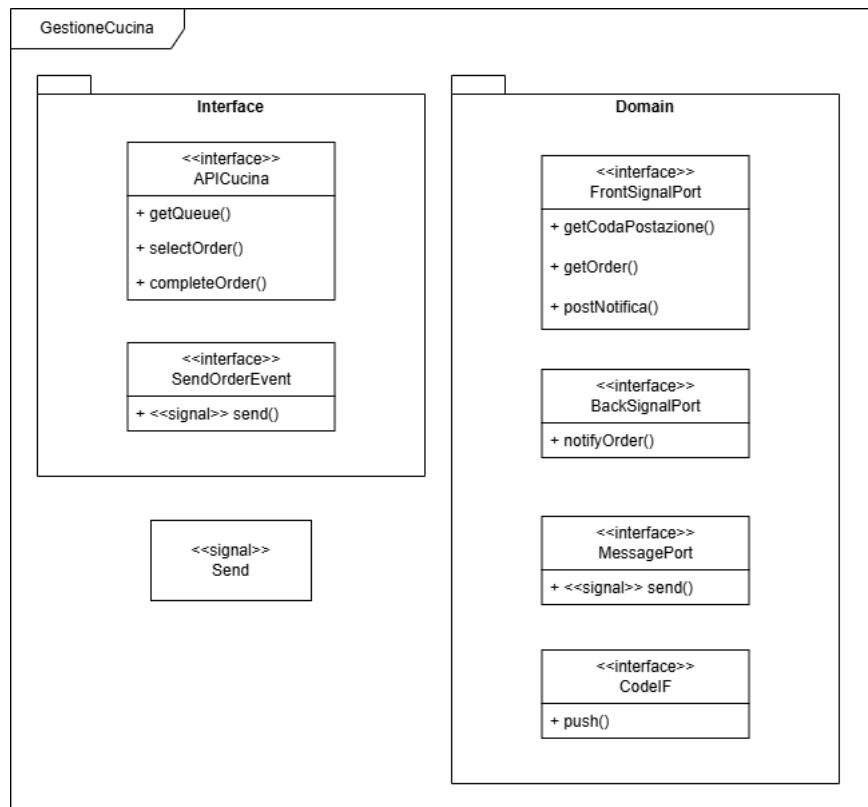


Figura 2.21: Interface class diagram - Gestione Cucina

2.6.3 Interfacce Gestione Cliente

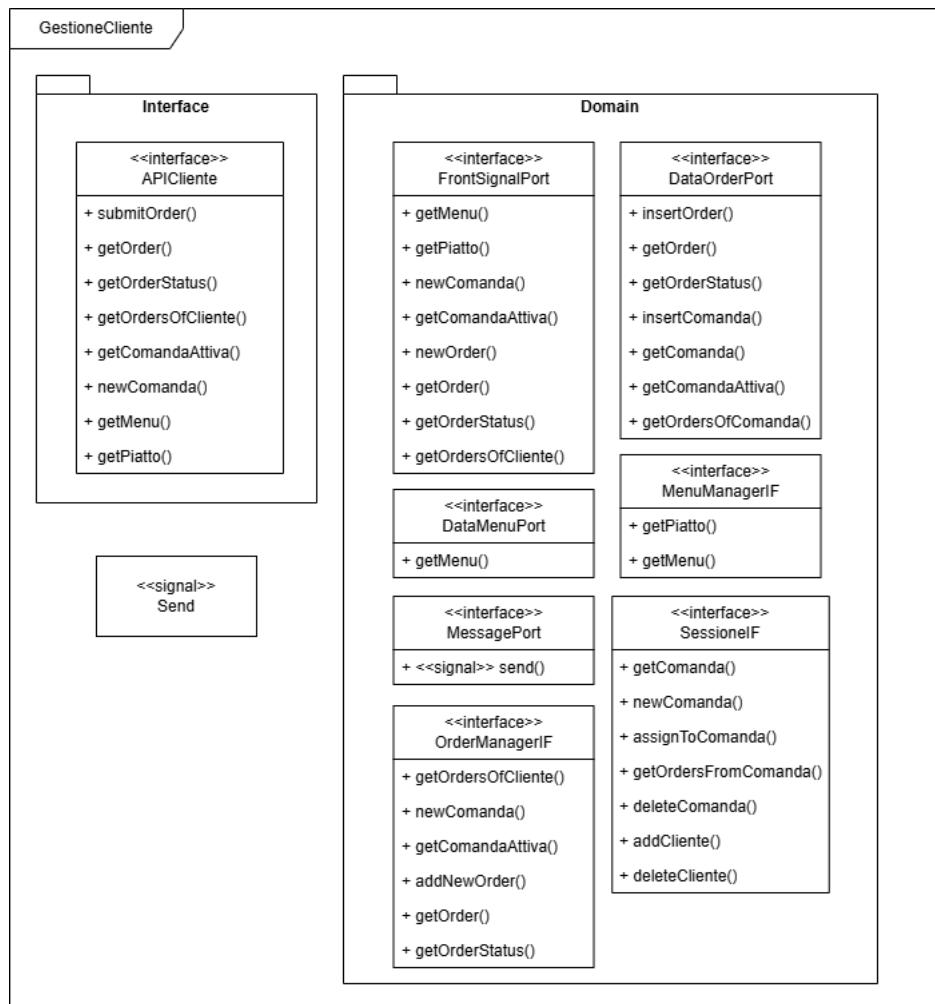


Figura 2.22: Interface class diagram - Gestione Cliente

2.7 Organizzazione del lavoro

2.7.1 Organizzazione del processo di sviluppo

Per l'implementazione dei microservizi che compongono il sistema ServeEasy, attualmente delineato al primo zoom-in, è stato applicato un approccio "polyrepo"[\[19\]](#), definendo per ogni microservizio un'area di progetto dedicata.

Scelta GitHub come piattaforma di hosting per il progetto software, un membro del team è stato incaricato del setup, controllo e gestione delle repository per i singoli microservizi.

Per raggiungere tale organizzazione, è stata prima di tutto creata un'area di lavoro per familiarizzare con le tecnologie selezionate, impiegando uno sforzo congiunto nello studio e prototipazione.

Nel processo di sviluppo sono state specificate delle regole mutualmente pattuite:

- Nessuno esegue push diretto dall'area di lavoro locale verso il main branch: ognuno lavora esclusivamente sulla propria branch;
- Nei commit e nelle pull requests va espressa una sintesi del proprio lavoro svolto;
- Cambiamenti importanti vanno discussi;
- Ogni implementazione va testata;
- Prima di effettuare una merge sul main branch, l'implementazione deve aver passato le fasi di build e di test con successo.

A supporto del processo di sviluppo, sono state introdotte automatizzazioni per garantire Continuous Integration e Continuous Delivery, servendosi di Github Workflows e Docker allo scopo di aumentare la velocità di deployment delle nuove modifiche apportate, garantendo al contempo integrità:

- Un evento di push verso il proprio branch causa l'avvio della job di CI posta a sorveglianza del proprio spazio di lavoro sulla repository, allo scopo di effettuare un controllo di compilazione;
- Un evento verso il main branch (quale, ad esempio, una merge) causa l'avvio della job di CI/CD posta a sorveglianza del main sulla repository, la quale effettua un controllo di compilazione, genera un eseguibile e procede con le fasi di build e ship dell'immagine Docker corrispondente verso il registry DockerHub [\[11\]](#).

In particolare, nel punto 2 si parla di Continuous Delivery e non Continuous Deployment, in quanto il deployment della nuova immagine va eseguita manualmente [26].

Il deployment verrà effettuato tramite Docker Compose: vengono definiti in un unico file i servizi offerti dal registry di riferimento (Dockerhub), le loro caratteristiche e dipendenze, cosicché sarà sufficiente avviare il file per poter scaricare le immagini dei microservizi e servizi d'interesse in una rete di container dedicata.

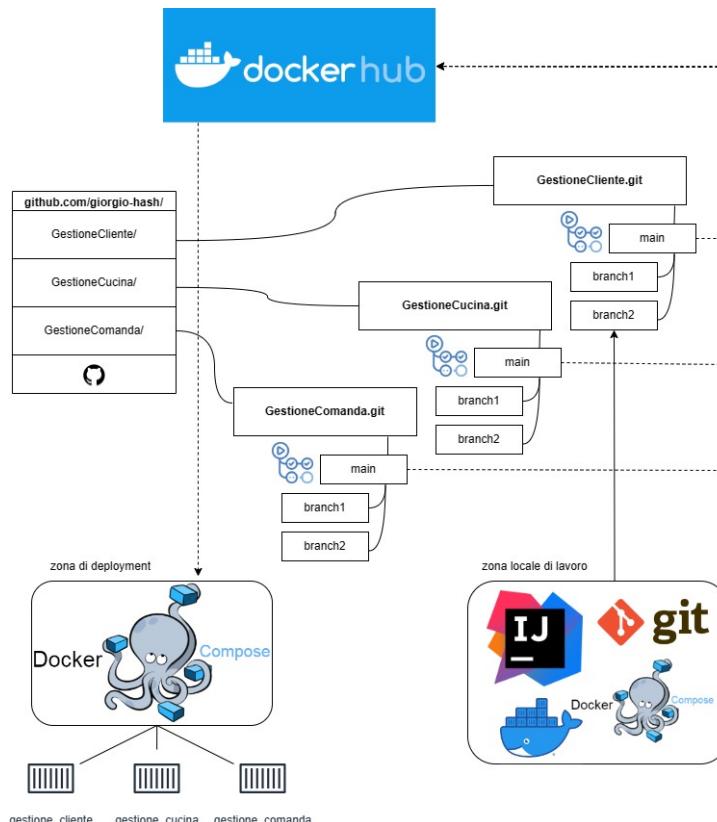


Figura 2.23: Organizzazione del lavoro cloud e locale, CI/CD e deployment

2.7.2 Organizzazione dell'area di lavoro

Oltre all'IDE di IntelliJ IDEA e Git, l'area di lavoro locale è supportata da Docker Compose per attivare i servizi a supporto dell'esecuzione del singolo microservizio: non solo dipendenze, quali Kafka, Zookeeper ed il database MariaDB, ma anche strumenti utili per la visualizzazione ed interazione ad alto livello col sistema, quali:

- Kafdrop per monitorare i messaggi passati tra pub e sub attraverso Kafka;
- PHPMyAdmin per monitorare, sviluppare ed iniettare dati nel database MariaDB.

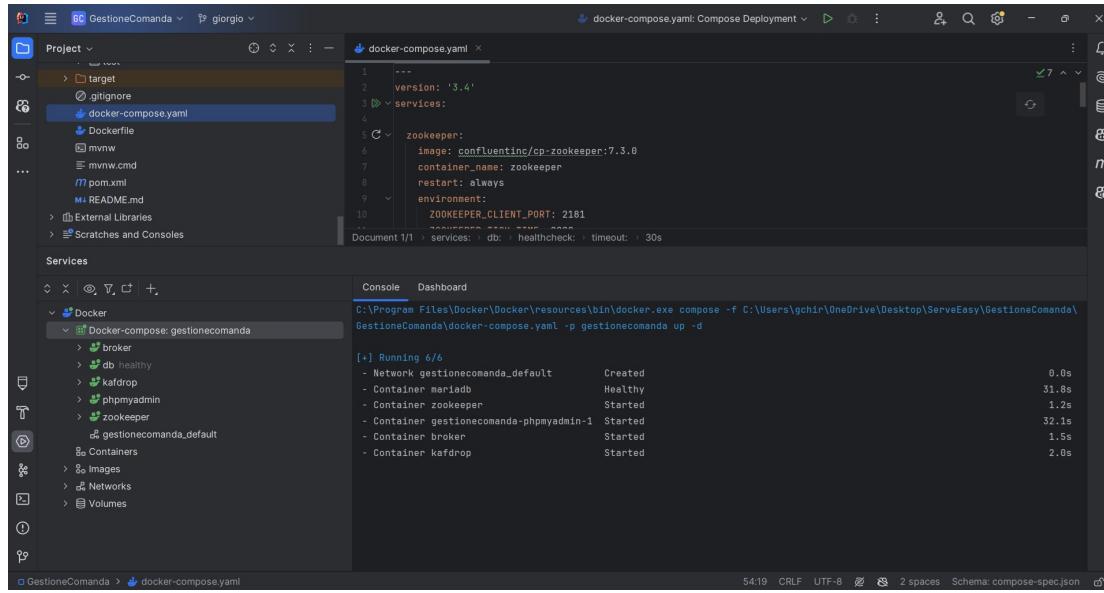


Figura 2.24: Ambiente di lavoro con IntelliJ IDEA e Docker Compose

Facendo leva sulla portabilità offerta dal framework Docker, viene garantito un ambiente altamente personalizzabile, flessibile e di facile implementazione.

Fin dal principio, l'area di lavoro è dotata della source tree che esplicita il layout dei subsystem identificati in fase di progettazione (paragrafo 2.3), comprendendo inoltre la cartella interfunzionale *config*, necessaria per contenere gli artefatti di configurazione delle funzionalità (ad es. per JPA).

Alla radice del progetto, vi è inoltre una cartella dedicata a contenere dati di persistenza (cartella */db*).

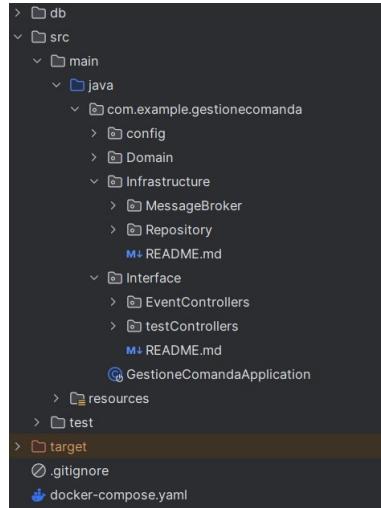


Figura 2.25: source tree del progetto GestioneComanda

L'applicativo viene costruito per essere eseguito sulla macchina locale ed esposto su *localhost:8080*. Lo sviluppo in locale è impostato per risolvere le dipendenze di

rete dell'applicazione tramite il framework Docker: i servizi offerti da database e message broker vengono quindi eseguiti in container distinti nella rete `serveeasy_default` creata da Docker (separata dalla rete `host` della macchina) ed esposte su `localhost` attraverso la tecnica del port binding.

```
1 [...]  
2     broker:  
3         image: confluentinc/cp-kafka:7.3.0  
4         container_name: broker  
5         restart: always  
6         ports:  
7             - "9092:9092"  
8 [...]
```

Codice 2.2: port-binding del servizio *broker* alla voce *ports* (porta_host:porta_container)

2.8 Costruzione dello scheletro

Vista la necessità di standardizzarsi sulle tecnologie e modalità in uso, la costruzione stessa dello scheletro ha seguito un approccio incrementale: si è iniziato da un microservizio, *GestioneComanda*, per poi proseguire con *GestioneCliente* e *GestioneCucina*.

Dunque, la realizzazione degli adattatori per ogni microservizio si baserà sempre sulla prima implementazione realizzata in *GestioneComanda* e, per tale ragione, verrà illustrato il processo di sviluppo prendendo questo componente di sistema come principale esempio.

2.8.1 Realizzazione database MariaDB

Utilizzando PHPMyAdmin, è stata semplificata la creazione ed esportazione del database. Realizzato lo schema dati, la funzione "Esporta" lo salva in un file .sql pronto per l'uso. Assieme allo schema, sono stati inoltre definiti ed esportati dei dati di prova su cui operare.

Figura 2.26: interfaccia grafica di PHPMyAdmin eseguito da container

2.8.2 Definizione Interfacce con principi SOLID

Prima di tutto, sono state scritte le interfacce necessarie per la comunicazione verso i componenti del sistema. In particolare, il design delle interfacce Port risulta cruciale per adottare correttamente il principio d'inversione delle dipendenze: il dominio deve poter ignorare la natura dei device di input, mentre deve comunicare coi dispositivi di output solo attraverso le interfacce Port.

In altre parole, si assume che l'informazione esterna arrivi al layer Interface, venga inoltrata al dominio e poi dal dominio verso Infrastructure, per poi andare ai dispositivi di

output: il principio d'inversione delle dipendenze vuole che il dominio dipenda da Interface e che Infrastructure dipenda dal dominio, mentre non è concesso il viceversa[20].

Per far ciò, si è cercato di seguire al meglio i cosiddetti principi **SOLID**[21]:

- Single responsibility: ogni interfaccia raduna funzionalità dedicate ad una specifica mansione;
- Open close: ogni elemento dev'essere estendibile ma non modificabile;
- Liskov substitution: gli oggetti di una superclasse devono essere rimpiazzabili da oggetti della sottoclasse senza avere effetti sul funzionamento del programma;
- Interface segregation: interfacce il più possibile contenute per evitarne sottouso;
- Dependency inversion: moduli di alto livello non dipendono da moduli di basso livello, entrambi dipendono da interfacce, le interfacce sono indipendenti.

```

1 public interface MessagePort<T> {
2     /**
3      * Invia l'oggetto passato come parametro sul topic del message
4      * broker
5      *
6      * @param t oggetto da inviare
7      * @throws JsonProcessingException eccezione sollevata nella
8      * serializzazione
9     */
10    void send(T t) throws JsonProcessingException;
11 }
```

Codice 2.3: Interfaccia MessagePort

```

1 public interface DataPort {
2
3     /**
4      * Controlla se l'ordine esiste nel database
5      *
6      * @param id id dell'entita' ordine da controllare la presenza
7      * @return true se esiste, false altrimenti
8      */
9     boolean isOrderExist(int id);
10
11    /**
12     * Salva l'entita' ordine all'interno del database
13     *
14     * @param ordineEntity entita' ordine da salvare
15     * @return entita' ordine salvata
16 }
```

```

16     */
17     OrdineEntity saveOrder(OrdineEntity ordineEntity);
18
19 /**
20  * Cerca nel db e restituisce l'ordine corrispondente all' id dato
21  *
22  * @param id id dell'ordine
23  * @return Optional<OrdineEntity> se esiste altrimenti Optional<
24  null>
25 /**
26 Optional<OrdineEntity> getOrderById(int id);
27
28 /**
29  * Aggiorna l'attributo stato dell'ordine
30  *
31  * @param id id dell'ordine su cui effetturare l'aggiornamento
32  * @param ordineEntity entita' con gli aggiornamenti parziali da
33  * applicare
34  * @return entita' aggiornata
35 /**
36 OrdineEntity updateOrder(int id, OrdineEntity ordineEntity);
37
38 /**
39  * Lista di tutti gli ordini per una specifica comanda
40  *
41  * @param idComanda id della comanda su cui cercare gli ordini
42  * @return lista di ordini per una data comanda
43 /**
44 List<OrdineEntity> findAllOrdersByIdComanda(int idComanda);
45
46 /**
47  * Cancella l'ordine con il dato ID dal database
48  *
49  * @param id id dell'ordine da eliminare
50 /**
51 void deleteOrder(int id);
52 }
```

Codice 2.4: Interfaccia DataPort

```

1 public interface NotifyOrderEvent {
2
3 /**
4  * Riceve un messaggio tramite Kafka dal servizio gestioneCliente
5  * in merito all'avvenuta ordinazione
```

```

5      * da parte di un cliente
6
7      *
8      * @param message il corpo del messaggio vero e proprio
9      * @param topic topic del message broker sul quale si riceve il
10     messaggio
11    *
12    * @param partition numero di partizione sul quale si riceve il
13    messaggio
14    *
15    * @param offset numero di offset che presenta il messaggio
16    ricevuto
17    */
18    @KafkaListener(id = "${spring.kafka.consumer.gestioneCliente.group-
19    id}", topics = "${spring.kafka.consumer.gestioneCliente.topic}")
20    void receive(String message, String topic, Integer partition, Long
21    offset) throws JsonProcessingException;
22
23    /**
24     * Restituisce l'ultima notifica letta dal listener
25     *
26     * @return oggetto notifica letto dal listener
27     */
28    NotificaOrdineDTO getLastMessageReceived();
29
30 }

```

Codice 2.5: Interfaccia NotifyOrderEvent

2.8.3 Adattatore Kafka

Lo sviluppo del componente adibito all’adattatore Kafka è stato un processo che ha seguito vari step:

Step 1 - Overview: Come primo passo ci si è fatti una panoramica leggendo la documentazione ufficiale di Spring Kafka [32] e si sono cercati alcuni progetti su Github di esempio.

Step 2 - Setup: Al passo numero due ci si è focalizzati sul setup dell’ambiente di lavoro, si sono definiti i seguenti container nel file "docker-compose.yaml" (Codice 2.6):

```

1 services:
2   zookeeper:
3     image: confluentinc/cp-zookeeper:7.3.0
4     container_name: zookeeper
5     restart: always
6     environment:
7       ZOOKEEPER_CLIENT_PORT: 2181

```

```

8      ZOOKEEPER_TICK_TIME: 2000
9
10     broker:
11       image: confluentinc/cp-kafka:7.3.0
12       container_name: broker
13       restart: always
14       ports:
15         - "9092:9092"
16       depends_on:
17         - zookeeper
18       environment:
19         KAFKA_BROKER_ID: 1
20         KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
21         KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,
22           PLAINTEXT_INTERNAL:PLAINTEXT
23         KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092,
24           PLAINTEXT_INTERNAL://broker:29092
25         KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
          KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
          KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1

```

Codice 2.6: Setup del docker-compose.yaml per l'adattatore Kafka

Fatto ciò si è passati ad aggiornare il file "pom.xml" (Codice 2.7) con la seguente dipendenza per Kafka:

```

1 <dependency>
2   <groupId>org.springframework.kafka</groupId>
3   <artifactId>spring-kafka</artifactId>
4 </dependency>

```

Codice 2.7: Aggiornamento dipendenze nel pom.xml per includere spring-kafka

Successivamente nel file "application.yml" (Codice 2.8) sono state aggiunte le seguenti istruzioni di configurazione, come si può notare si è partiti inizialmente solamente a considerare un'applicazione costituita da un singolo producer senza consumer:

```

1 spring:
2   kafka:
3     bootstrap-servers: localhost:9092
4     producer:
5       topic: sendOrderEvent
6       group-id: gestioneComanda

```

Codice 2.8: Aggiornamento del file 'application.yml' per il producer Kafka

Sono stati poi creati i primi bean di configurazione nella classe "KafkaConfig.java" (Codice 2.9 nella pagina successiva):

```

1  @Configuration
2  public class KafkaConfig {
3
4      @Value("${spring.kafka.bootstrap-servers}")
5      private String bootstrapServers;
6
7      @Bean
8      public ProducerFactory<String, String> producerFactory() {
9          Map<String, Object> configProps = new HashMap<>();
10         configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
11             bootstrapServers);
12         configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
13             StringSerializer.class);
14         configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
15             StringSerializer.class);
16         return new DefaultKafkaProducerFactory<>(configProps);
17     }
18
19     @Bean
20     public KafkaTemplate<String, String> kafkaTemplate() {
21         return new KafkaTemplate<>(producerFactory());
22     }

```

Codice 2.9: Classe di configurazione KafkaConfig.java

E di Object Mapper (Codice 2.10) per la serializzazione e deserializzazione in formato JSON, visto che la comunicazione tramite message broker avviene tramite messaggi in quel formato:

```

1  @Configuration
2  public class JsonConfig {
3
4      @Bean
5      public ObjectMapper objectMapper(){
6          final ObjectMapper objectMapper = new ObjectMapper();
7          // configurazione di timestamp
8          objectMapper.registerModule(new JavaTimeModule());
9          objectMapper.setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:
10             mm:ss.SSS"));
11          return objectMapper;
12      }

```

Codice 2.10: Classe di configurazione JsonConfig.java

Step 3 - Producer: Nel passo successivo si è passati a creare la prima classe adibita al compito di producer kafka, ossia CucinaPubProducer.java (Codice 2.11) che implementa la MessagePort (Codice 2.3 a pagina 48):

```

1  @Service
2  @Log
3  public class CucinaPubProducer implements MessagePort<OrdineDTO> {
4
5      private final KafkaTemplate<String, String> kafkaTemplate;
6
7      private final ObjectMapper objectMapper;
8
9      /**
10      * topic sul quale e' in ascolto la cucina
11      */
12      @Value("${spring.kafka.producer.topic}")
13      private String topic;
14
15      @Autowired
16      public CucinaPubProducer(final KafkaTemplate<String, String>
kafkaTemplate, final ObjectMapper objectMapper){
17          this.kafkaTemplate = kafkaTemplate;
18          this.objectMapper = objectMapper;
19      }
20
21      @Override
22      public void send(OrdineDTO ordineDTO) throws
JsonProcessingException {
23
24          // Serializza in un oggetto JSON
25          final String payload = objectMapper.writeValueAsString(
ordineDTO);
26
27          // invia messaggio sul topic specificato
28          CompletableFuture<SendResult<String, String>> future =
kafkaTemplate.send(topic, payload);
29          future.whenComplete((result, ex)->{
30              if(ex == null){
31                  log.info("Sent Message=[ " + payload + " ] with offset=[ "
+ result.getRecordMetadata().offset() + " ]");
32              }
33              else{
34                  log.info("Unable to send message=[ " + payload + " ] due
to : " + ex.getMessage());
35              }
36          });

```

```

37     }
38 }
```

Codice 2.11: Classe del producer kafka CucinaPubProducer.java

Si può notare che questa classe richiede un bean di ObjectMapper (Codice 2.10 a pagina 52) e KafkaTemplate (Codice 2.9 a pagina 52) definiti al passo precedente, mentre richiede dall'application.yml (Codice 2.8 a pagina 51) il nome del topic kafka in questione. Inizialmente si è usato una stringa “Hello world” al posto dell'oggetto OrdineDTO e si è verificato il funzionamento tramite log sulla console, per poter iniettare messaggi si è utilizzata l'istruzione tramite terminale:

```

1 docker exec --interactive --tty broker kafka-console-producer --
  bootstrap-server broker:9092 --topic "sendOrderEvent"
```

Codice 2.12: Operazione di post sul topic kafka

mentre per ricevere messaggi:

```

1 docker exec --interactive --tty broker kafka-console-consumer --
  bootstrap-server broker:9092 --topic "notifyOrderEvent" --from-
  beginning
```

Codice 2.13: Operazione di get sul topic kafka

Step 4 - User Interface Verificato il funzionamento tramite log si è passati ad usare Kafdrop[24] ossia una User Interface (UI) per i topic di Kafka, integrandolo nella nostra applicazione con le seguenti istruzioni nel docker-compose.yml:(Codice 2.14):

```

1 services:
2   kafdrop:
3     image: obsidiandynamics/kafdrop
4     container_name: kafdrop
5     restart: "no"
6     ports:
7       - "9001:9000"
8     environment:
9       KAFKA_BROKERCONNECT: "broker:29092"
10    depends_on:
11      - "broker"
```

Codice 2.14: Aggiornamento del docker-compose.yaml perl per Kafdrop

Di seguito (Figura 2.27 nella pagina successiva) viene mostrato un esempio della UI in funzione sulla porta localhost:9001 che mostra tutti i messaggi postati sul topic *sendOrderEvent* ordinati in base all'offset:

The screenshot shows the Kafdrop interface with the title "Topic Messages: sendOrderEvent". At the top, there are filters: "First Offset: 0", "Last Offset: 3", and "Size: 3". Below these are dropdowns for "Partition" (0), "Offset" (0), "# messages" (100), "Key format" (DEFAULT), and "Message format" (DEFAULT). A blue button labeled "View Messages" is present. The message list shows three entries:

- Offset: 0 Key: empty Timestamp: 2024-05-03 11:10:22.444 Headers: empty


```
{"id":17,"idComanda":7,"idPiatto":"SUH724","stato":1,"urgenzaCliente":0,"tordinazione":"2024-05-03 13:10:21.359"}
```
- Offset: 1 Key: empty Timestamp: 2024-05-03 11:54:02.799 Headers: empty


```
{"id":18,"idComanda":4,"idPiatto":"CAR123","stato":0,"urgenzaCliente":1,"tordinazione":"2024-05-03 13:54:02.555"}
```
- Offset: 2 Key: empty Timestamp: 2024-05-03 11:54:35.125 Headers: empty


```
{"id":19,"idComanda":3,"idPiatto":"RIS001","stato":1,"urgenzaCliente":0,"tordinazione":"2024-05-03 13:54:35.095"}
```

Figura 2.27: Esempio funzionamento kafdrop

Step 5 - Testing In questo passo è stato creato il primo test di integrazione per il producer per verificare il corretto invio di un messaggio sul topic tramite il producer appena creato e la ricezione tramite un consumer simulato utilizzando un’istanza Embedded Kafka [7]. Embedded Kafka è una libreria che fornisce istanze di Kafka e Confluent Schema Registry in memoria per eseguire i test, in modo da non dipendere da un server Kafka esterno. Viene quindi integrata nel pom.xml (Codice 2.15) la seguente dipendenza:

```

1 <dependency>
2   <groupId>org.springframework.kafka</groupId>
3   <artifactId>spring-kafka-test</artifactId>
4   <scope>test</scope>
5 </dependency>
```

Codice 2.15: Aggiornamento dipendenze nel pom.xml per includere spring-kafka-test

E aggiornato l’application.properties specifico dei test (Codice 2.16):

```

1 spring.kafka.bootstrap-servers=localhost:9092
2 spring.kafka.consumer.auto-offset-reset=earliest
3 spring.kafka.producer.topic=test.topic.comanda
```

Codice 2.16: Aggiornamento del file ‘application.properties‘ di test per il producer kafka

Fatto ciò si può procedere a scrivere la seguente classe di test (Codice 2.17):

```

1 @EnableKafka
2 @SpringBootTest()
3 @DirtiesContext
4 @TestMethodOrder(MethodOrderer.OrderAnnotation.class)
5 @EmbeddedKafka(partitions = 1,
6                 controlledShutdown = false,
```

```

7     brokerProperties = { "listeners=PLAINTEXT://localhost:9092", " 
8       port=9092" },
9     topics = "{$spring.kafka.producer.topic"})
10    class CucinaPubProducerTests {
11
12      @Autowired
13      private CucinaPubProducer producer;
14      @Autowired
15      private EmbeddedKafkaBroker embeddedKafka;
16      @Autowired
17      private ObjectMapper objectMapper;
18      @Value("${spring.kafka.producer.topic}")
19      private String topic;
20      private Logger logger;
21      private TestAppender testAppender;
22
23      @BeforeEach
24      public void setup() {
25          logger = (Logger) LoggerFactory.getLogger(CucinaPubProducer.
26 class);
27          testAppender = new TestAppender();
28          testAppender.start();
29          logger.addAppender(testAppender);
30      }
31
32      @Test
33      public void testSendingMessage() throws JsonProcessingException {
34
35          OrdineDTO ordineDTO = TestDataUtil.createOrdineDtoA();
36
37          producer.send(ordineDTO);
38          String message = objectMapper.writeValueAsString(ordineDTO);
39
40          // configurazione del consumer di Embedded Kafka
41          Map<String, Object> consumerProps = KafkaTestUtils.
42          consumerProps("testT", "false", embeddedKafka);
43          DefaultKafkaConsumerFactory<Integer, String> cf = new
44          DefaultKafkaConsumerFactory<>(consumerProps);
45          Consumer<Integer, String> consumer = cf.createConsumer();
46          embeddedKafka.consumeFromAnEmbeddedTopic(consumer, topic);
47          ConsumerRecord<Integer, String> received = KafkaTestUtils.
48          getSingleRecord(consumer, topic);
49
50          // testo il corretto invio
51          assertFalse(testAppender.events.isEmpty());

```

```

47     assertEquals("Sent Message=[" + message + "] with offset=[0]",  

48     testAppender.events.get(0).getFormattedMessage());  

49  

50     // testo la corretta ricezione  

51     assertThat(received.offset()).isEqualTo(0);  

52     assertThat(received.topic()).isEqualTo(topic);  

53     assertThat(received.partition()).isEqualTo(0);  

54     OrdineDTO ordineDTOResolved = objectMapper.readValue(received.  

55     value(), OrdineDTO.class);  

56     assertThat(ordineDTOResolved).isEqualTo(ordineDTO);  

57  

58     logger.detachAppender(testAppender);
}

```

Codice 2.17: Test di integrazione per il producer CucinaPubProducer

Che testa il corretto invio di un oggetto creato tramite la classe di supporto testDataUtil e verifica inizialmente che il log della classe CucinaPubAdapter sia quello che ci si aspetta, mentre successivamente verifica che il consumer creato utilizzando un'istanza di EmbeddedKafka riceva l'oggetto inviato, viene di conseguenza testata anche la serializzazione.

Step 6 - Consumer Creato e testato il producer si è passati alla creazione di un consumer, inizialmente si è creato un consumer generico in ascolto sullo stesso topic del producer appena creato, verificato il funzionamento si è passati a creare i due producer richiesti in maniera speculare, di seguito verrà mostrato il producer in ascolto sul topic del micro-servizio di Gestione Cliente ossia SubClienteAdapter. Come prima cosa si è aggiornato l'application.yml (Codice 2.18) con la configurazione del consumer in questione:

```

1 kafka:  

2   consumer:  

3     gestioneCliente:  

4       topic: notifyOrderEvent  

5       group-id: gestioneCliente

```

Codice 2.18: Aggiornamento del file ‘application.yml‘ per il consumer Kafka

Poi si è passati direttamente ad implementare la classe SubClienteAdapter (Codice: 2.19) che implementa l'interfaccia NotifyOrderEvent (Codice: 2.5 a pagina 49)

```

1 @Component  

2 @Log  

3 public class SubClienteAdapter implements NotifyOrderEvent {  

4

```

```
5     private ObjectMapper objectMapper;
6     private ClientePort clientePort;
7
8     /**
9      * variabile thread safe che serve per fini di test per verificare
10     * che il listener abbia ricevuto un messaggio
11     */
12
13    private CountDownLatch latch = new CountDownLatch(1);
14    private final Logger logger = LoggerFactory.getLogger(
15        SubCucinaAdapter.class);
16    private NotificaOrdineDTO lastMessageReceived;
17
18    @Autowired
19    public SubClienteAdapter(final ClientePort clientePort, final
20    ObjectMapper objectMapper) {
21        this.clientePort = clientePort;
22        this.objectMapper = objectMapper;
23    }
24
25    @Override
26    public void receive(@Payload String message,
27                        @Header(KafkaHeaders.RECEIVED_TOPIC) String
28    topic,
29                        @Header(KafkaHeaders.RECEIVED_PARTITION)
30    Integer partition,
31                        @Header(KafkaHeaders.OFFSET) Long offset)
32    throws JsonProcessingException {
33        logger.info("Received a message {}, from {} topic, " +
34                    "{} partition, and {} offset", message.toString(),
35        topic, partition, offset);
36        NotificaOrdineDTO notificaOrdineDTO = objectMapper.readValue(
37        message, NotificaOrdineDTO.class);
38        clientePort.notifyOrder(notificaOrdineDTO);
39        lastMessageReceived = notificaOrdineDTO;
40        latch.countDown();
41    }
42
43    /**
44     * resetta il valore del latch
45     */
46
47    public void resetLatch() {
48        latch = new CountDownLatch(1);
49    }
50
51    /**
52     *
```

```

42     * restituisce il latch
43     *
44     * @return latch: variabile thread safe che serve per fini di test
45     * per verificare
46     * che il listener abbia ricevuto un messaggio
47     */
48     public CountDownLatch getLatch() {
49         return latch;
50     }
51
52     /**
53     * Restituisce l'ultimo messaggio letto dal listener
54     *
55     * @return l'ultimo messaggio letto dal listener
56     */
57     @Override
58     public NotificaOrdineDTO getLastMessageReceived() {
59         return lastMessageReceived;
60     }

```

Codice 2.19: Classe del consumer kafka SubClienteAdapter.java

Nella quale il metodo receive è annotato con `@KafkaListener`, questo permette di stare costantemente in ascolto sul topic specificato e svolgere il contenuto del metodo quando viene rilevato un messaggio, nello specifico si produce un log di corretta ricezione e si salva l'oggetto ricevuto. Inizialmente si è lavorato solamente tramite log con un messaggio “Hello world”, successivamente verificato il funzionamento si è passati a lavorare con oggetti e quindi viene effettuata la deserializzazione tramite `objectMapper` e il salvataggio nel campo `lastMessageReceived`. Per quanto riguarda il `latch` è una variabile thread safe che serve a fini di test per verificare che il listener abbia ricevuto un messaggio utilizzando un comportamento sincrono, non viene coinvolta nel normale svolgimento dell’applicazione. Si passa quindi alla fase di test per verificare che il consumer `SubClienteAdapter` stia in ascolto e riceva messaggi correttamente dal topic del message broker, inizialmente si configura l’application.properties (Codice 2.20) di test con il topic di test del consumer:

- 1 `spring.kafka.consumer.gestioneCliente.topic=test.topic.cliente`

Codice 2.20: Aggiornamento del file ‘application.properties‘ di test per il consumer Kafka

Poi in modo analogo alla classe del producer viene creato il seguente test di integrazione (Codice 2.21) sempre sfruttando Embedded Kafka:

- 1 `@EnableKafka`
- 2 `@SpringBootTest`

```
3 @DirtiesContext
4 @Log
5 @EmbeddedKafka(partitions = 1,
6     controlledShutdown = false,
7     brokerProperties = { "listeners=PLAINTEXT://localhost:9092", "port=9092" },
8     topics = {"${spring.kafka.consumer.gestioneCliente.topic}")})
9 public class SubClienteAdapterTests {
10     @Autowired
11     private EmbeddedKafkaBroker embeddedKafka;
12     @Autowired
13     private SubClienteAdapter subClienteAdapter; // kafka consumer
14     @Value("${spring.kafka.consumer.gestioneCliente.topic}")
15     private String topic;
16     private Logger logger;
17     private TestAppender testAppender;
18
19     @BeforeEach
20     public void setup() {
21         subClienteAdapter.resetLatch();
22         logger = (Logger) LoggerFactory.getLogger(SubCucinaAdapter.class);
23     };
24         testAppender = new TestAppender();
25         testAppender.start();
26         logger.addAppender(testAppender);
27     }
28
29     @Test
30     public void testOutput1() throws Exception {
31         NotificaOrdineDTO notificaOrdineDTO = TestDataUtil.
32             createNotificaOrdineDTOA();
33         String notifica = TestUtil.serialize(notificaOrdineDTO);
34
35         CompletableFuture<SendResult<Integer, String>> future = TestUtil
36             .sendMessageToTopic(topic, notifica, embeddedKafka);
37         log.info("Sent Message=[ " + notifica + " ] with offset=[0]");
38
39         boolean messageConsumed = subClienteAdapter.getLatch().await(10,
40             TimeUnit.SECONDS);
41
42         //testo il corretto invio
43         future.whenComplete((result, ex)->{
44             assertThat(ex).isNull();
45             // verifica che non sia stata sollevata alcuna eccezione
46         });
47 }
```

```

43
44     // testo la corretta ricezione
45     assertTrue(messageConsumed);
46     assertFalse(testAppender.events.isEmpty());
47     assertEquals("Received a message " + notifica + ", from " +
48     topic + " topic, 0 partition, and 0 offset", testAppender.events.
49     get(0).getFormattedMessage());
50
51     NotificaOrdineDTO notificaOrdineDTOReceived = subClienteAdapter.
52     getLastMessageReceived();
53     assertEquals(notificaOrdineDTO, notificaOrdineDTOReceived);
54     logger.detachAppender(testAppender);
55 }
56 }
```

Codice 2.21: Test di integrazione per il consumer SubClienteAdapter

In questa classe di test si utilizza un’istanza di Embedded Kafka per simulare il producer, mentre come consumer si utilizza la classe sotto test SubClienteAdapter. Si testa il corretto invio di un oggetto creato tramite la classe di supporto testDataUtil e si verifica che il producer invii effettivamente un messaggio sul topic, poi si verifica la ricezione verificando inizialmente che il log della classe SubClienteAdapter sia quello che ci si aspetta, mentre successivamente si verifica che riceva l’oggetto inviato, viene di conseguenza testata anche la deserializzazione.

2.8.4 Adattatore JPA

Lo sviluppo del componente adibito all’adattatore JPA è stato un processo che ha seguito vari step:

Step 1 - Overview: Come primo passo ci si è fatti una panoramica leggendo la documentazione ufficiale di Spring Data JPA [31] e si sono cercati alcuni progetti su Github di [esempio](#).

Step 2 - Setup: Al passo numero due ci si è focalizzati sul setup dell’ambiente di lavoro, andando ad aggiornare il file "pom.xml" (Codice 2.22) con la seguente dipendenza per Spring Data JPA:

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-jpa</artifactId>
4 </dependency>
```

Codice 2.22: Aggiornamento dipendenze nel pom.xml per includere spring-data-jpa

Successivamente nel file "application.yml" (Codice 2.23) sono state aggiunte le seguenti istruzioni di configurazione:

```

1  spring:
2      jpa:
3          properties:
4              hibernate:
5                  dialect: org.hibernate.dialect.MariaDBDialect
6                  temp:
7                      use_jdbc_metadata_defaults: false
8          hibernate:
9              ddl-auto: none #update
10         show-sql: true

```

Codice 2.23: Aggiornamento del file ‘application.yml’ per Spring Data JPA

Step 3 - Entità: Nel passo successivo si è passati a creare la prima entità, le entità in Spring Boot JPA sono fondamentalmente POJOs (Plain Old Java Objects) che rappresentano dati che possono essere resi persistenti nel database. Ogni istanza di un’entità rappresenta una riga in una tabella del database. Si è definita quindi l’entità Ordine (Codice 2.24) in questo modo:

```

1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  @Builder
5  @Entity
6  @Table(name = "Ordine", schema = "serveeasy", catalog = "")
7  public class OrdineEntity {
8      /**
9      * Identificatore dell'ordine
10     */
11     @GeneratedValue(strategy = GenerationType.IDENTITY)
12     @Id
13     @Column(name = "ID", nullable = false, insertable = false,
14             updatable = false)
15     private int id;
16
17     /**
18     * Identificatore della comanda di cui l'ordine fa parte
19     */
20     @Column(name = "ID_COMANDA", nullable = false, updatable = false)
21     private int idComanda;
22
23     /**

```

```

23     * Identificatore del piatto ordinato dal cliente
24     */
25     @Basic
26     @Column(name = "ID_piatto", nullable = false, length = 20)
27     private String idPiatto;
28
29 /**
30     * Stato dell'ordine
31     * 0: Ordine preso in carico
32     * 1: Ordine in coda di preparazione
33     * 2: Ordine in preparazione
34     * 3: Ordine preparato
35     */
36     @Basic
37     @Column(name = "stato", updatable = true)
38     private Integer stato;
39
40 /**
41     * Istante temporale in cui viene effettuata l'ordinazione
42     * pattern : "yyyy-MM-dd HH:mm:ss.SSS"
43     */
44     @Basic
45     @CreationTimestamp
46     @Column(name = "t_ordinazione", updatable = false)
47     private Timestamp t0ordinazione;
48
49 /**
50     * Attributo urgenza del cliente
51     * 0 : espresso non urgenza
52     * 1 : espresso urgenza
53     * 2 : default
54     */
55     @Basic
56     @Column(name = "urgenza_cliente", updatable = true)
57     private Integer urgenzaCliente;
58 }

```

Codice 2.24: Classe entità OrdineEntity.java

Grazie alla notazione `@Entity` segnaliamo a JPA che questa classe è un'entità, con `@Table` specifichiamo il nome della tabella nel database e con `@Id` definiamo la chiave primaria della entità.

Step 4 - Repository: Nello step 4 andiamo a definire una repository per la classe ordine, ossia un meccanismo per l'incapsulamento dello storage, recupero e comportamento di

ricerca che emula una collezione di oggetti:

```

1 @Repository
2 public interface OrdineRepository extends CrudRepository<OrdineEntity,
3   Integer> {
4
5   /**
6    * Permette di ottenere una lista di Entita' Ordine con lo stesso
7    * id di comanda specificato
8    *
9    * @param idComanda codice identificativo della comanda
10   * @return oggetto Iterable che punta a una lista contenente entita'
11   * ordine con lo stesso id di comanda specificato
12   */
13  Iterable<OrdineEntity> findOrdineEntitiesByIdComanda(int idComanda)
14  ;
15 }
```

Codice 2.25: Classe repository OrdineRepository.java

La classe OrdineRepository (Codice 2.25) gestisce le operazioni CRUD (Create, Read, Update, Delete) per l'entità OrdineEntity, inoltre questa interfaccia definisce un metodo personalizzato: *findOrdineEntitiesByIdComanda(int idComanda)* che restituisce un Iterable di OrdineEntity che hanno lo stesso *idComanda* specificato. Spring Data JPA implementerà automaticamente questo metodo, non è quindi necessario fornire un'implementazione personalizzata a meno che non si desideri un comportamento specifico che non è coperto dalle convenzioni di denominazione di Spring Data JPA.

Step 5 - JPA Adapter: A questo punto siamo pronti per definire la classe adattatore di JPA JPADBAdapter.java (Codice 2.26) che implementa la DataPort (Codice 2.4 a pagina 48):

```

1 @Repository
2 public class JPADBAdapter implements DataPort {
3
4   private final OrdineRepository ordineRepository;
5
6   @Autowired
7   public JPADBAdapter(OrdineRepository ordineRepository) {
8     this.ordineRepository = ordineRepository;
9   }
10
11  @Override
12  public boolean isOrderExist(int id) {
13    return ordineRepository.existsById(id);
14 }
```

```

14     }
15
16     @Override
17     public OrdineEntity saveOrder(OrdineEntity ordineEntity) {
18         return ordineRepository.save(ordineEntity);
19     }
20
21     @Override
22     public Optional<OrdineEntity> getOrderById(int id) {
23         return ordineRepository.findById(id);
24     }
25
26     @Override
27     public OrdineEntity updateOrder(int id, OrdineEntity ordineEntity) {
28         ordineEntity.setId(id);
29
30         return ordineRepository.findById(ordineEntity.getId()).map(
31             existingOrder -> {
32                 Optional.ofNullable(ordineEntity.getStato()).ifPresent(
33                     existingOrder::setStato);
34                 Optional.ofNullable(ordineEntity.getUrgenzaCliente()).ifPresent(
35                     existingOrder::setUrgenzaCliente);
36                 return ordineRepository.save(existingOrder);
37             }).orElseThrow(() -> new RuntimeException("Order does not exist
38             "));
39         }
40
41     @Override
42     public List<OrdineEntity> findAllOrdersByIdComanda(int idComanda) {
43         return StreamSupport.stream(ordineRepository.
44             findOrdineEntitiesByIdComanda(idComanda).spliterator(), false).
45             collect(Collectors.toList());
46     }

```

Codice 2.26: Classe adattatore JPA JPADBAdapter.java

Questa classe funge da adattatore tra il database e l'applicazione, fornendo un'implementazione dei metodi definiti nell'interfaccia DataPort, consentendo operazioni di accesso ai dati per l'entità ordine. È possibile notare come viene utilizzato l'oggetto istanza della

classe OrdineRepository definita poc’anzi (Codice 2.25 a pagina 64) per eseguire operazioni di persistenza dei dati relativi agli ordini senza aver fornito un’implementazione, ma lasciando questo compito a Spring Data JPA.

Step 6 - Testing: In questo passo si sono svolti i test sulla DataPort per verificare il corretto funzionamento dell’adattatore JPA correlato. Per eseguire i test si è utilizzato un database H2[16] in memoria poichè offre velocità, isolamento, semplicità di configurazione e indipendenza dall’infrastruttura esterna, rendendo i test più efficienti e affidabili. Quindi come prima cosa si integra la dipendenza di H2 nel file "pom.xml" (Codice 2.27) in questo modo:

```

1 <dependency>
2   <groupId>com.h2database</groupId>
3   <artifactId>h2</artifactId>
4   <scope>runtime</scope>
5 </dependency>
```

Codice 2.27: Aggiornamento del file pom.xml per la dipendenza di H2

Poi si aggiorna il file application.properties (Codice 2.28) specifico dei test come segue:

```

1 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.
  MariaDBDialect
2 spring.jpa.hibernate.ddl-auto=create-drop
3 spring.jpa.show-sql=true
4 # H2 Database in memory to simulate MariaDB
5 spring.datasource.url=jdbc:h2:mem:testdb;MODE=MariaDB;DATABASE_TO_LOWER
  =TRUE
6 spring.datasource.username=user
7 spring.datasource.password=user
8 spring.datasource.driver-class-name=org.h2.Driver
```

Codice 2.28: Aggiornamento del file application.properties per i test con H2

A questo punto si può passare a creare il test di integrazione della DataPort (Codice 2.29):

```

1 @SpringBootTest
2 @ExtendWith(SpringExtension.class)
3 @DirtiesContext(classMode = DirtiesContext.ClassMode.
  AFTER_EACH_TEST_METHOD)
4 public class DataPortTests {
5
6   private DataPort dataPort;
7
8   @Autowired
9   public DataPortTests(DataPort dataPort) {
10     this.dataPort = dataPort;
```

```

11     }
12
13     @Test
14     public void testSaveOrderAndFindOrderById() {
15         OrdineEntity ordineEntity = TestDataUtil.createOrdineEntityA();
16         OrdineEntity savedOrdineEntity = dataPort.saveOrder(
17             ordineEntity);
18         Optional<OrdineEntity> result = dataPort.getOrderById(
19             savedOrdineEntity.getId());
20         assertThat(result).isPresent();
21         assertThat(result.get()).isEqualTo(savedOrdineEntity);
22         assertThat(result.get().getId()).isEqualTo(1);
23         assertThat(result.get().getIdComanda()).isEqualTo(ordineEntity.
24             getIdComanda());
25         assertThat(result.get().getStato()).isEqualTo(ordineEntity.
26             getStato());
27         assertThat(result.get().getIdPiatto()).isEqualTo(ordineEntity.
28             getIdPiatto());
29         assertThat(result.get().getTordinazione()).isEqualTo(
30             ordineEntity.getTordinazione());
31     }
32
33     @Test
34     public void testMultipleSaveOrderAndFindByIdComanda() {
35         OrdineEntity ordineEntityA = TestDataUtil.createOrdineEntityA()
36         ;
37         OrdineEntity savedOrdineEntityA = dataPort.saveOrder(
38             ordineEntityA);
39         OrdineEntity ordineEntityB = TestDataUtil.createOrdineEntityB()
40         ;
41         OrdineEntity savedOrdineEntityB = dataPort.saveOrder(
42             ordineEntityB);
43         OrdineEntity ordineEntityC = TestDataUtil.createOrdineEntityC()
44         ;
45         OrdineEntity savedOrdineEntityC = dataPort.saveOrder(
46             ordineEntityC);
47
48         Iterable<OrdineEntity> result = dataPort.
49         findAllOrdersByIdComanda(ordineEntityA.getIdComanda());
50         assertThat(result).hasSize(2).containsExactly(
51             savedOrdineEntityA, savedOrdineEntityB);
52     }
53
54     @Test
55     public void testOrderPartialUpdate() {

```

```

42     OrdineEntity ordineEntity = TestDataUtil.createOrdineEntityA();
43     OrdineEntity savedOrdineEntity = dataPort.saveOrder(
44         ordineEntity);
45     savedOrdineEntity.setStato(0);
46     savedOrdineEntity.setUrgenzaCliente(1);
47     dataPort.saveOrder(savedOrdineEntity);
48     Optional<OrdineEntity> result = dataPort.getOrderById(
49         savedOrdineEntity.getId());
50     assertThat(result).isPresent();
51     assertThat(result.get()).isEqualTo(savedOrdineEntity);
52 }
53
54 @Test
55 public void testDeleteOrder() {
56     OrdineEntity ordineEntityA = TestDataUtil.createOrdineEntityA()
57     ;
58     OrdineEntity savedOrdineEntityA = dataPort.saveOrder(
59         ordineEntityA);
60     dataPort.deleteOrder(savedOrdineEntityA.getId());
61     Optional<OrdineEntity> result = dataPort.getOrderById(
62         savedOrdineEntityA.getId());
63     assertThat(result).isEmpty();
64 }
```

Codice 2.29: Classe adattatore JPA JPADBAdapter.java

In questa classe di test vengono testate tutte e 4 le operazioni CRUD, ossia Create (Creazione), Read (Lettura), Update (Aggiornamento) e Delete (Eliminazione) per mezzo della DataPort e quindi della sua implementazione JPADBAdapter (2.26 a pagina 64) verso il database, le entità vengono create tramite una classe di supporto TestDataUtil e la DataPort viene iniettata tramite il costruttore grazie all’annotazione @Autowired.

2.8.5 Continuous Integration

Una volta creati tutti gli adattatori principali l’attenzione si è spostata ad introdurre tecniche di Continuous Integration (CI), la CI è una pratica di sviluppo software che prevede che i membri di un team integrino frequentemente il loro lavoro in un repository condiviso, al fine di rilevare e risolvere rapidamente eventuali problemi o conflitti nel codice.

Step 1 - Overview: Come primo passo ci si è fatti una panoramica leggendo la documentazione ufficiale di GitHub Actions [9].

Step 2 - Setup: Al passo numero due ci si è focalizzati sul setup dell’ambiente di lavoro, andando a creare la cartella ".github/workflows" nella repository principale di Gestione Comanda e creando il file "maven.yml" (Codice 2.30) come segue:

```

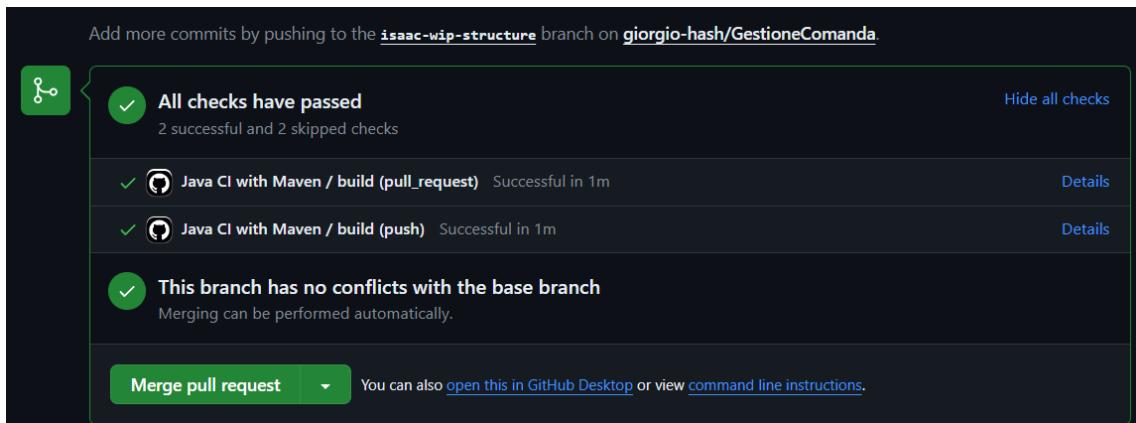
1 name: Java CI with Maven
2
3 on:
4   push:
5     branches: [ "*" ]
6   pull_request:
7     branches: [ "main" ]
8
9 jobs:
10   build:
11     runs-on: ubuntu-latest
12
13
14   steps:
15     - uses: actions/checkout@v4
16     - name: Set up JDK 17
17       uses: actions/setup-java@v4
18       with:
19         java-version: '17'
20         distribution: 'temurin'
21         cache: maven
22     - name: Run the Maven verify phase
23       run: mvn --batch-mode clean verify

```

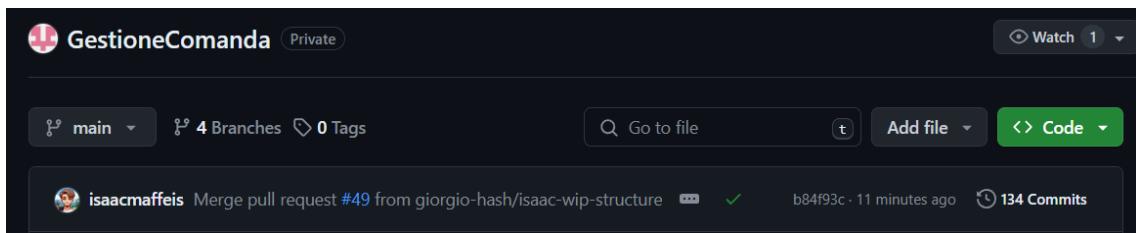
Codice 2.30: Creazione del file maven.yml

Questo flusso di lavoro configura l’ambiente di esecuzione, scarica il codice sorgente del progetto, imposta il JDK 17 ed infine esegue i test del progetto utilizzando Maven. È un’applicazione di CI che garantisce che il codice sia testato automaticamente ogni volta che vengono apportate modifiche su ogni branch e ogni volta che si effettua una pull-request verso il main.

Step 3 - Testing: Per testare il funzionamento basta fare il push di un commit verso la cartella remota di GitHub o eseguire una pull-request verso il main, prendiamo in considerazione questo esempio in cui si esegue la seconda delle due opzioni appena citate:

**Figura 2.28:** CI merge a pull-request

Dalla Figura 2.28 è possibile notare come all'interno della pull-request sia presente un campo dedicato ai controlli che GitHub Actions ha effettuato e siccome tutti i controlli hanno avuto esito positivo, è possibile procedere al merge della richiesta.

**Figura 2.29:** CI main check

Una volta completato il merge con successo è possibile notare una spunta verde nella repository principale del progetto (Figura 2.29) che ci informa che allo stato corrente il codice sul main ha passato tutti i test proposti.

2.8.6 Continuous Delivery

Applicando modifiche al maven.yml precedentemente definito, è stata aggiunta una job che, a seguito della fase CI di cui sopra, crei l'immagine Docker contenente l'eseguibile .jar, destinata alla registry Dockerhub. Al fine di non eseguire molteplici jobs, sono state introdotte delle condizioni affinchè Github Actions capisca quali job eseguire e quali saltare, specificando così l'esecuzione di una job di CI/CD per gli eventi che interessano il branch main, mentre gli altri branch eseguono solo una job di CI.

La nuova job introdotta sfrutta il Dockerfile definito all'interno del progetto per creare l'immagine da spedire verso la repository definita dalle credenziali salvate dall'amministratore di repository. Tali segreti sono conservati nella repository al percorso **Settings > Security > Secrets and Variables > Actions > Environment Secrets**.

```

1 [...] 
2
3 jobs:
4   build:
5     runs-on: ubuntu-latest
6
7     if: github.ref != 'refs/heads/main'
8     steps:
9       - uses: actions/checkout@v4
10      [...]
11
12   build-and-ship:
13     runs-on: ubuntu-latest
14
15     if: github.ref == 'refs/heads/main'
16     steps:
17       - uses: actions/checkout@v4
18       - name: Set up JDK 17
19         uses: actions/setup-java@v4
20         with:
21           java-version: '17'
22           distribution: 'temurin'
23           cache: maven
24       - name: Run the Maven verify phase
25         run: mvn --batch-mode clean verify
26       - name: Build & push docker image
27         uses: mr-smithers-excellent/docker-build-push@v5
28         with:
29           image: gchiricoli/gestione_comanda
30           tags: latest
31           registry: docker.io
32           dockerfile: Dockerfile
33           username: ${{ secrets.DOCKERHUB_USERNAME }}
34           password: ${{ secrets.DOCKERHUB_TOKEN }}
```

Codice 2.31: modifica file maven.yml per CI/CD

```

1 FROM openjdk:latest
2 COPY target/GestioneComanda-0.0.1-SNAPSHOT.jar GestioneComanda-0.0.1-
3   SNAPSHOT.jar
4 EXPOSE 8080
5 ENTRYPOINT ["java", "-jar", "GestioneComanda-0.0.1-SNAPSHOT.jar"]
```

Codice 2.32: Dockerfile di GestioneComanda

2.8.7 DTO

Gli oggetti di trasferimento dati (Data Transfer Object) sono un design pattern utilizzato per trasferire dati tra sottosistemi di un'applicazione software, servono a semplificare e a rendere più efficiente la comunicazione tra i vari livelli di un'applicazione, nel nostro caso tra Interface, Domain e Infrastructure.

Step 1 - Overview: Come primo passo ci si è fatti una panoramica leggendo la documentazione ufficiale di Model Mapper [22], ossia la libreria di mapping che useremo per convertire le entità in DTO e viceversa.

Step 2 - Setup: Al passo numero due ci si è focalizzati sul setup dell'ambiente di lavoro, andando ad aggiornare inizialmente il file "pom.xml" (Codice 2.33) con la seguente dipendenza per Model Mapper:

```

1 <dependency>
2   <groupId>org.modelmapper</groupId>
3   <artifactId>modelmapper</artifactId>
4   <version>3.0.0</version>
5 </dependency>
```

Codice 2.33: Aggiornamento dipendenze nel pom.xml per includere model-mapper

E successivamente è stato creato il bean di configurazione nella classe "MapperConfig.java" (Codice 2.34):

```

1 @Configuration
2 public class MapperConfig {
3
4     @Bean
5     ModelMapper modelMapper(){
6         ModelMapper modelMapper = new ModelMapper();
7         modelMapper.getConfiguration().setMatchingStrategy(
8             MatchingStrategies.LOOSE);
9         return modelMapper;
10    }
```

Codice 2.34: Classe di configurazione MapperConfig.java

La strategia di corrispondenza LOOSE in ModelMapper ignora le differenze di maiuscole/minuscole e gli underscore nei nomi dei campi, considera i campi come corrispondenti anche se i nomi dei campi nell'oggetto sorgente e nell'oggetto destinazione non sono esattamente gli stessi, ma contengono le stesse parole.

Step 3 - DTO: Al terzo passo è stata creata la classe DTO per l'entità Ordine che abbiamo definito in precedenza (Codice 2.24 a pagina 62), semplicemente creando una nuova classe Java (Codice 2.35) con gli stessi campi della classe entità, ma senza la logica di business o le annotazioni specifiche dell'entità:

```

1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  @Builder
5  public class OrdineDTO {
6
7      /**
8      * Identificatore dell'ordine
9      */
10     private int id;
11
12    /**
13    * Identificatore della comanda di cui l'ordine fa parte
14    */
15    private int idComanda;
16
17    /**
18    * Identificatore del piatto ordinato dal cliente
19    */
20    private String idPiatto;
21
22    /**
23    * Stato dell'ordine
24    * 0: Ordine preso in carico
25    * 1: Ordine in coda di preparazione
26    * 2: Ordine in preparazione
27    * 3: Ordine preparato
28    */
29    private Integer stato;
30
31    /**
32    * Istante temporale in cui viene effettuata l'ordinazione
33    * pattern : "yyyy-MM-dd HH:mm:ss.SSS"
34    */
35    private Timestamp tOrdinazione;
36
37    /**
38    * Attributo urgenza del cliente
39    * 0 : espresso non urgenza
40    * 1 : espresso urgenza

```

```

41     * 2 : default
42     */
43     private Integer urgenzaCliente;
44
45 }
```

Codice 2.35: Classe DTO per l'entità ordine OrderDTO.java

Step 4 - Mapper: Le classi mapper sono utilizzate per convertire oggetti tra classi entità e DTO (e viceversa), viene inizialmente creata la seguente interfaccia (Codice 2.36):

```

1 public interface Mapper<A,B> {
2
3     /**
4      * Mappa l'oggetto A (Entita') nell'oggetto B (DTO)
5      *
6      * @param a A Entita'
7      * @return B DTO
8      */
9     B mapTo(A a);
10
11    /**
12     * Mappa l'oggetto B (DTO) nell'oggetto A (Entita')
13     *
14     * @param b B DTO
15     * @return A Entita'
16     */
17    A mapFrom(B b);
```

Codice 2.36: Interfaccia Mapper.java

Nella quale con *mapTo* possiamo passare da un'entità ad un DTO, mentre con *mapFrom* possiamo convertire un DTO in un'entità. Viene di conseguenza creata la classe implementazione di questa interfaccia (Codice 2.37):

```

1 @Component
2 public class OrdineMapper implements Mapper<OrdineEntity, OrdineDTO> {
3
4     private ModelMapper modelMapper;
5
6     public OrdineMapper(ModelMapper modelMapper) {
7         this.modelMapper = modelMapper;
8     }
9
10    @Override
11    public OrdineDTO mapTo(OrdineEntity ordineEntity) {
```

```

12     return modelMapper.map(ordineEntity, OrdineDTO.class);
13 }
14
15 @Override
16 public OrdineEntity mapFrom(OrdineDTO ordineDTO) {
17     return modelMapper.map(ordineDTO, OrdineEntity.class);
18 }
19 }
```

Codice 2.37: Classe OrdineMapper.java implementazione di ModelMapper.java

Si può facilmente notare come l'utilizzo di ModelMapper abbia semplificato notevolmente il processo di mapping, infatti ci basta richiamare il metodo *map* offerto dall'istanza modelMapper per svolgere la conversione tra due oggetti.

Step 5 - Testing: In questo passo sono stati svolti i test di integrazione (Codice 2.38) per verificare che la conversione tra entità e DTO sia stata implementata in modo corretto.

```

1 @SpringBootTest
2 public class OrdineMapperTests {
3
4     @Autowired
5     private OrdineMapper ordineMapper;
6
7     @Test
8     public void testMapTo(){
9         OrdineEntity ordineEntity = TestDataUtil.createOrdineEntityA();
10        OrdineDTO ordineDTO = ordineMapper.mapTo(ordineEntity);
11        assertThat(ordineDTO.getId()).isEqualTo(ordineEntity.getId());
12        assertThat(ordineDTO.getIdComanda()).isEqualTo(ordineEntity.
13            getIdComanda());
14        assertThat(ordineDTO.getStato()).isEqualTo(ordineEntity.
15            getStato());
16        assertThat(ordineDTO.getIdPiatto()).isEqualTo(ordineEntity.
17            getIdPiatto());
18        assertThat(ordineDTO.getUrgenzaCliente()).isEqualTo(
19            ordineEntity.getUrgenzaCliente());
20        assertThat(ordineDTO.getT0ordinazione()).isEqualTo(ordineEntity.
21            getT0ordinazione());
22    }
23
24    @Test
25    public void testMapFrom(){
26        OrdineDTO ordineDTO = TestDataUtil.createOrdineDtoB();
27        OrdineEntity ordineEntity = ordineMapper.mapFrom(ordineDTO);
28        assertThat(ordineEntity.getId()).isEqualTo(ordineDTO.getId());
```

```

24     assertThat(ordineEntity.getIdComanda()).isEqualTo(ordineDTO.
25         getIdComanda());
26     assertThat(ordineEntity.getStato()).isEqualTo(ordineDTO.
27         getStato());
28     assertThat(ordineEntity.getIdPiatto()).isEqualTo(ordineDTO.
29         getIdPiatto());
30     assertThat(ordineEntity.getUrgenzaCliente()).isEqualTo(
31         ordineDTO.getUrgenzaCliente());
32     assertThat(ordineEntity.getTordinazione()).isEqualTo(ordineDTO.
33         getTordinazione());
34   }
35 }
```

Codice 2.38: Test di integrazione OrdineMapperTests.java

Il metodo *testMapTo()* verifica che l'operazione di mappatura da *OrdineEntity* a *OrdineDTO* produca risultati attesi, mentre il metodo *testMapFrom()* verifica che l'operazione inversa di mappatura da *OrdineDTO* a *OrdineEntity* produca risultati attesi, in entrambi i casi si considerano dei test di equivalenza su tutti i campi degli oggetti.

2.8.8 Interfaccia di Test

Step 1 - Overview: Come primo passo ci si è fatti una panoramica leggendo la documentazione ufficiale di Spring per costruire un controller REST [29].

Step 2 - Setup: Al passo numero due ci si è focalizzati sul setup dell'ambiente di lavoro, siccome il microservizio GestioneComanda è sprovvisto di un componente HTTP Controller nella sua Interfaccia (che contiene solo EventController), viene creato un controller di TEST per interagire direttamente con i componenti del servizio per i soli fini di test, per questo motivo viene aggiornato il diagramma UML di Interface di GestioneComanda come segue (Figura 2.30 nella pagina successiva):

zoom-in interface di gestione comanda

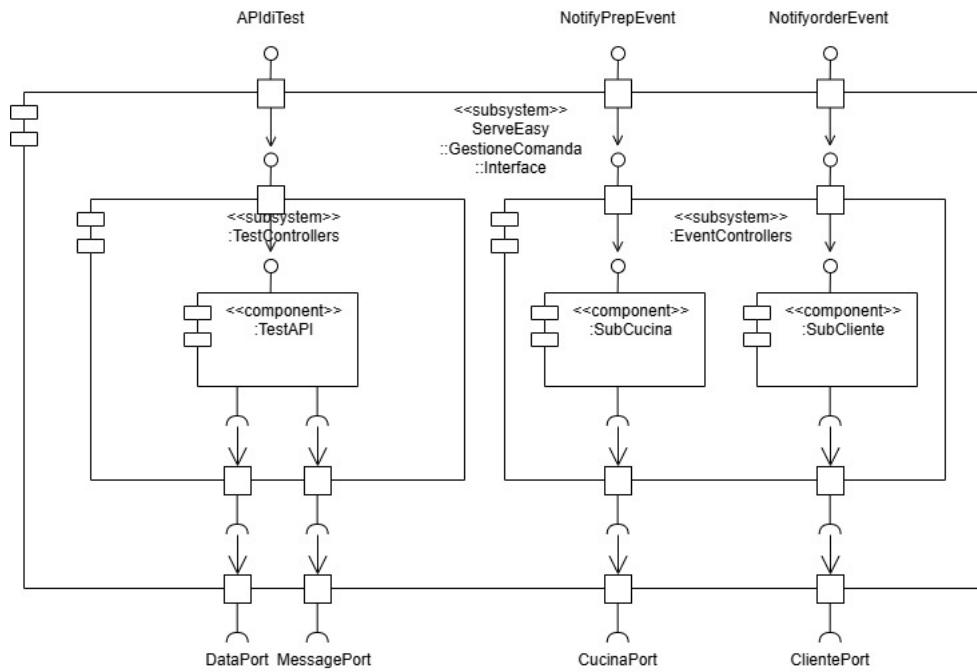


Figura 2.30: Component diagram - Gestione Comanda - Interface con Test

- EventControllers: SubCucina e SubCliente, permettono la ricezione di messaggi tramite message broker dagli altri microservizi.
- TestControllers: TestAPI per poter testare le API di Test utilizzando direttamente la DataPort e la MessagePort

Risulta importante specificare come il componente *TestAPI* utilizzi direttamente la *DataPort* e la *MessagePort* invece che passare per le interfacce *CucinaPort* e *ClientePort* ovverte dal *Domain*, questo viene fatto per non sconvolgere l'intera struttura dell'applicazione per i soli fini di test, infatti questo componente non verrà utilizzato in produzione.

Successivamente viene aggiornato il file "pom.xml"(Codice 2.39) con le seguenti dipendenze:

```

1 <!-- Spring web -->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5 </dependency>
6
7 <!-- Jackson -->
8 <dependency>
9   <groupId>com.fasterxml.jackson.core</groupId>

```

```

10     <artifactId>jackson-databind</artifactId>
11 </dependency>
12
13 <dependency>
14     <groupId>com.fasterxml.jackson.datatype</groupId>
15     <artifactId>jackson-datatype-jsr310</artifactId>
16 </dependency>

```

Codice 2.39: Aggiornamento dipendenze nel pom.xml per includere spring-web

La prima (Spring web) è utilizzata per configurare un'applicazione Spring Boot che presenta un controller REST, include infatti tutte le dipendenze necessarie per avviare un'applicazione web, inclusi Tomcat, Spring MVC e altre dipendenze di supporto. Mentre le seconde (Jackson) sono dipendenze Jackson utilizzate per la serializzazione e deserializzazione degli oggetti Java in formato JSON e viceversa fondamentali quando si sviluppano servizi RESTful con Spring Boot.

Step 3 - API: In questo passo vengono definite le API di Test che dovrà esporre il TestController, viene quindi creata l'interfaccia (Codice 2.40) annotandola con `@RequestMapping("/test")`, il che significa che tutti gli endpoint definiti all'interno di questa interfaccia sono raggiungibili tramite l'URI base "/test":

```

1 @RequestMapping("/test")
2 public interface TestAPI {
3
4     /**
5      * Salva nel database l'oggetto ordine dato un ordineDTO
6      *
7      * @param ordineDTO DTO dell'entità ordine da salvare
8      * @return entità risposta che contiene l'oggetto creato e una
9      *         risposta HTTP associata
10     */
11    @PostMapping(path="/order")
12    ResponseEntity<OrdineDTO> addOrdine(@RequestBody OrdineDTO
13    ordineDTO);
14
15    /**
16     * Restituisci l'ordine corrispondente all'id dato in input
17     *
18     * @param id id dell'ordine richiesto
19     * @return entità risposta che contiene l'oggetto richiesto e una
20     *         risposta HTTP associata
21     */
22    @GetMapping(path="order/{id}")
23    ResponseEntity<OrdineDTO> getOrdine(@PathVariable int id);

```

```

21
22     /**
23      * Restituisce una lista con tutti gli ordini relativi a una data
24      * comanda
25      *
26      * @param idComanda id della comanda di riferimento
27      * @return entita' risposta che contiene la lista richiesta e una
28      * risposta HTTP associata
29      */
30
31     @GetMapping(path = "orders/{idComanda}")
32     ResponseEntity<List<OrdineDTO>> getAllOrdersByIdComanda(
33         @PathVariable int idComanda);
34
35
36
37
38     /**
39      * Aggiornamento parziale dell'entita' ordine, e' possibile fornire
40      * solamente gli oggetti da aggiornare
41      *
42      * @param id id dell'ordine da aggiornare
43      * @param ordineDTO oggetto DTO con le modifiche da effettuare
44      * @return entita' risposta che contiene l'oggetto aggiornato e una
45      * risposta HTTP associata
46      */
47
48     @PatchMapping(path="order/{id}")
49     ResponseEntity<OrdineDTO> partialUpdateOrdine(@PathVariable int id,
50         @RequestBody OrdineDTO ordineDTO);
51
52
53
54
55
56

```

```
    serializzazione
57     */
58     @PostMapping(path = "/sendorderevent")
59     ResponseEntity<OrdineDTO> sendOrderEvent(@RequestBody OrdineDTO
60     ordineDTO) throws JsonProcessingException;
61
62     /**
63      * Espone una API di GET con la quale e' possibile ottenere l'
64      * ultimo messaggio letto sul topic SendOrderEvent
65      * Si testa il topic notifyPrepEvent da gestione cucina verso
66      * gestione comanda
67      * @return entita' risposta che contiene l'oggetto richiesto e una
68      * risposta HTTP associata
69      */
70
71     @GetMapping(path = "/sendorderevent")
72     ResponseEntity<String> getMessageFromTopicSendOrderEvent();
73
74     /**
75      * Espone una API di POST con la quale e' possibile iniettare all'
76      * interno del broker oggetti al fine di test
77      * Si testa il topic notifyOrderEvent da gestione cliente verso
78      * gestione comanda
79      * @param notificaOrdineDTO contenuto dell'oggetto da iniettare
80      * @return entita' risposta che contiene l'oggetto creato e una
81      * risposta HTTP associata
82      * @throws JsonProcessingException eccezione sollevata dalla
83      * serializzazione
84      */
85
86     @PostMapping(path = "/notifyorderevent")
87     ResponseEntity<NotificaOrdineDTO> sendNotifyOrderEvent(@RequestBody
88     NotificaOrdineDTO notificaOrdineDTO) throws
89     JsonProcessingException;
90
91     /**
92      * Espone una API di GET con la quale e' possibile ottenere l'
93      * ultimo messaggio letto sul topic NotifyOrderEvent
94      * Si testa il topic notifyPrepEvent da gestione cucina verso
95      * gestione comanda
96      * @return entita' risposta che contiene l'oggetto richiesto e una
97      * risposta HTTP associata
98      */
99
100    @GetMapping(path = "/notifyorderevent")
101    ResponseEntity<NotificaOrdineDTO>
102    getMessageFromTopicNotifyOrderEvent();
```

```

87 /**
88  * Espone una API di POST con la quale e' possibile iniettare all'
89  * interno del broker oggetti al fine di test
90  * Si testa il topic notifyPreEvent da gestione cucina verso
91  * gestione comanda
92  * @param notificaPreEventDTO contenuto dell'oggetto da iniettare
93  * @return entita' risposta che contiene l'oggetto creato e una
94  * risposta HTTP associata
95  * @throws JsonProcessingException eccezione sollevata dalla
96  * serializzazione
97  */
98 @PostMapping(path = "/notifyPreEvent")
99 ResponseEntity<NotificaPreEventDTO> sendNotifyPreEvent(
100 @RequestBody NotificaPreEventDTO notificaPreEventDTO) throws
101 JsonProcessingException;
102
103 /**
104  * Espone una API di GET con la quale e' possibile ottenere l'
105  * ultimo messaggio letto sul topic NotifyPreEvent
106  * Si testa il topic notifyPreEvent da gestione cucina verso
107  * gestione comanda
108  * @return entita' risposta che contiene l'oggetto richiesto e una
109  * risposta HTTP associata
110 */
111 @GetMapping(path = "/notifyPreEvent")
112 ResponseEntity<NotificaPreEventDTO>
113 getMessageFromTopicNotifyPreEvent();
114 }
```

Codice 2.40: Interfaccia TestAPI.java

In questo modo definiamo una serie di endpoint per API REST. Ogni metodo dell’interfaccia rappresenta un endpoint dell’API e specifica il tipo di richiesta HTTP supportata, il percorso dell’endpoint e gli eventuali parametri richiesti.

Step 4 - RestController: A questo punto possiamo costruire l’implementazione dell’interfaccia TestAPI appena definita (Codice 2.40 a pagina 78) andando a creare un controller REST di test (Codice 2.41):

```

1 @RestController
2 public class TestController implements TestAPI {
3     private TestService testService;
4     private Mapper<OrdineEntity, OrdineDTO> ordineMapper;
5     private DataPort dataPort;
6     @Value("${spring.kafka.consumer.gestioneCliente.topic}")
7     private String topic_notifyOrderEvent;
```

```
8  @Value("${spring.kafka.consumer.gestioneCucina.topic}")
9  private String topic_notifyPrepEvent;
10
11 @Autowired
12 public TestController(TestService testService, Mapper<OrdineEntity,
13 OrdineDTO> ordineMapper, DataPort dataPort) {
14     this.testService = testService;
15     this.ordineMapper = ordineMapper;
16     this.dataPort = dataPort;
17 }
18
19 @Override
20 public ResponseEntity<OrdineDTO> addOrdine(OrdineDTO ordineDTO) {
21     OrdineEntity ordineEntity = ordineMapper.mapFrom(ordineDTO);
22     OrdineEntity savedOrdineEntity = dataPort.saveOrder(
23         ordineEntity);
24     OrdineDTO savedOrdineDTO = ordineMapper.mapTo(savedOrdineEntity
25 );
26     return new ResponseEntity<>(savedOrdineDTO, HttpStatus.CREATED)
27 ;
28 }
29
30 @Override
31 public ResponseEntity<OrdineDTO> getOrdine(int id) {
32     Optional<OrdineEntity> ordineEntity = dataPort.getOrderById(id)
33 ;
34     if(ordineEntity.isPresent()){
35         OrdineDTO ordineDTO = ordineMapper.mapTo(ordineEntity.get()
36 );
37         return new ResponseEntity<>(ordineDTO, HttpStatus.OK);
38     }
39     return new ResponseEntity<>(HttpStatus.NOT_FOUND);
40 }
41
42 @Override
43 public ResponseEntity<OrdineDTO> partialUpdateOrdine(@PathVariable
44 int id, @RequestBody OrdineDTO ordineDTO) {
45     if(!dataPort.isOrderExist(id))
46         return new ResponseEntity<>(HttpStatus.NOT_FOUND);
47
48     OrdineEntity ordineEntity = ordineMapper.mapFrom(ordineDTO);
49     OrdineEntity updatedEntity = dataPort.updateOrder(id,
50         ordineEntity);
51
52     return new ResponseEntity<>(ordineMapper.mapTo(updatedEntity),
```

```

    HttpStatus.OK);
45
46
47     @Override
48     public ResponseEntity<List<OrdineDTO>> getAllOrdersByIdComanda(int
49     idComanda) {
50         List<OrdineEntity> ordini = dataPort.findAllOrdersByIdComanda(
51         idComanda);
52         if(!ordini.isEmpty())
53             return new ResponseEntity<>(ordini.stream()
54                 .map(ordinemapper::mapTo)
55                 .collect(Collectors.toList()),HttpStatus.OK);
56         else
57             return new ResponseEntity<>(HttpStatus.NOT_FOUND);
58     }
59
60
61     @Override
62     public ResponseEntity deleteOrder(@PathVariable("id") int id) {
63         dataPort.deleteOrder(id);
64         return new ResponseEntity(HttpStatus.NO_CONTENT);
65     }
66
67     @Override
68     public ResponseEntity<OrdineDTO> sendOrderEvent(@RequestBody
69     OrdineDTO ordineDTO) throws JsonProcessingException {
70         OrdineEntity ordineEntity = ordinemapper.mapFrom(ordineDTO);
71         OrdineEntity savedOrdineEntity = dataPort.saveOrder(
72         ordineEntity);
73         OrdineDTO savedOrdineDTO = ordinemapper.mapTo(savedOrdineEntity
74     );
75         testService.sendMessageToTopicSendOrderEvent(savedOrdineDTO);
76         return new ResponseEntity<>(savedOrdineDTO, HttpStatus.CREATED)
77     ;
78     }
79
80     @Override
81     public ResponseEntity<String> getMessageFromTopicSendOrderEvent() {
82         Optional<String> message = testService.peekFromSendOrderEvent()
83     ;
84         if(message.isPresent())
85             return new ResponseEntity<>(message.get(), HttpStatus.OK);
86         else
87             return new ResponseEntity<>(HttpStatus.NOT_FOUND);
88     }
89
90
91

```

```
82     @Override
83     public ResponseEntity<NotificaOrdineDTO> sendNotifyOrderEvent(
84         @RequestBody NotificaOrdineDTO notificaOrdineDTO) throws
85         JsonProcessingException {
86         String message = testService.serializeObject(notificaOrdineDTO)
87         ;
88         testService.sendMessageToTopic(message, topic_notifyOrderEvent)
89         ;
90         return new ResponseEntity<>(notificaOrdineDTO, HttpStatus.
91             CREATED);
92     }
93
94
95     @Override
96     public ResponseEntity<NotificaOrdineDTO>
97     getMessageFromTopicNotifyOrderEvent() {
98         Optional<NotificaOrdineDTO> message = testService.
99         peekFromNotifyOrderEvent();
100        if(message.isPresent())
101            return new ResponseEntity<>(message.get(), HttpStatus.OK);
102        else
103            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
104    }
105
106    @Override
107    public ResponseEntity<NotificaPrepOrdineDTO> sendNotifyPrepEvent(
108        @RequestBody NotificaPrepOrdineDTO notificaPrepOrdineDTO) throws
109        JsonProcessingException {
110        String message = testService.serializeObject(
111            notificaPrepOrdineDTO);
112        testService.sendMessageToTopic(message, topic_notifyPrepEvent);
113        return new ResponseEntity<>(notificaPrepOrdineDTO, HttpStatus.
114             CREATED);
115    }
116
117    @Override
118    public ResponseEntity<NotificaPrepOrdineDTO>
119    getMessageFromTopicNotifyPrepEvent() {
120        Optional<NotificaPrepOrdineDTO> message = testService.
121        peekFromNotifyPrepEvent();
122        if(message.isPresent())
123            return new ResponseEntity<>(message.get(), HttpStatus.OK);
124        else
125            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
126    }
127 }
```

Codice 2.41: Classe REST Controller di test TestController.java

Questa classe è un controller Spring MVC che gestisce le richieste HTTP relative alle operazioni CRUD (Create, Read, Update, Delete) sull’entità ordine del database passando per i corrispettivi DTO e gestisce le richiesti di operazioni sui topic del message broker, utilizzando il pattern architetturale API RESTful. Per una documentazione più dettagliata su ogni singola API si rimanda al capitolo 2.10 a pagina 91.

Step 5 - Testing: In questo passo ci siamo occupati di testare il REST Controller appena creato, lo abbiamo fatto utilizzando MockMVC [27], ovvero una classe fornita da Spring MVC Test Framework che consente di simulare le richieste HTTP e testare il comportamento dei controller Spring MVC senza dover effettivamente avviare un server HTTP.

```

1 @SpringBootTest
2 @EnableKafka
3 @DirtiesContext(classMode = DirtiesContext.ClassMode.
4     AFTER_EACH_TEST_METHOD)
5 @EmbeddedKafka(partitions = 1,
6     controlledShutdown = false,
7     brokerProperties = { "listeners=PLAINTEXT://localhost:9092", "port=9092" },
8     topics = "{$spring.kafka.producer.topic}",
9         "{$spring.kafka.consumer.gestioneCliente.topic}" ,
10        "{$spring.kafka.consumer.gestioneCucina.topic}" })
11 @AutoConfigureMockMvc
12 public class TestControllerTests {
13     @Autowired
14     private MockMvc mockMvc;
15     @Autowired
16     private ObjectMapper objectMapper;
17     @Autowired
18     private DataPort dataPort;
19     @Autowired
20     private EmbeddedKafkaBroker embeddedKafka;
21     @Value("${spring.kafka.consumer.gestioneCliente.topic}")
22     private String topic_notifyOrderEvent;
23     @Value("${spring.kafka.consumer.gestioneCucina.topic}")
24     private String topic_notifyPrepEvent;
25     @Value("${spring.kafka.producer.topic}")
26     private String topic_sendOrderEvent;
27     @Test

```

```

28     public void testThatGetOrderReturnsHttpStatus200WhenOrderExist()
29         throws Exception {
30
31         OrdineEntity ordineEntity = TestDataUtil.createOrdineEntityB();
32         dataPort.saveOrder(ordineEntity);
33         mockMvc.perform(MockMvcRequestBuilders.get("/test/order/1").
34             contentType(MediaType.APPLICATION_JSON)).andExpect(
35             MockMvcResultMatchers.status().isOk());
36     }
37
38
39     @Test
40     public void testThatGetOrderReturnsHttpStatus404WhenNoOrderExists()
41         throws Exception {
42         mockMvc.perform(MockMvcRequestBuilders.get("/test/order/99").
43             contentType(MediaType.APPLICATION_JSON)).andExpect(
44             MockMvcResultMatchers.status().isNotFound());
45     }
46
47     @Test
48     public void testThatGetOrderReturnsOrderWhenOrderExist() throws
49     Exception {
50
51         OrdineEntity ordineEntity = TestDataUtil.createOrdineEntityB();
52         dataPort.saveOrder(ordineEntity);
53
54         mockMvc.perform(
55             MockMvcRequestBuilders.get("/test/order/1")
56                 .contentType(MediaType.APPLICATION_JSON)
57         ).andExpect(
58             MockMvcResultMatchers.jsonPath("$.id").value(1)
59         ).andExpect(
60             MockMvcResultMatchers.jsonPath("$.idComanda").value(
61             ordineEntity.getIdComanda())
62         ).andExpect(
63             MockMvcResultMatchers.jsonPath("$.idPiatto").value(
64             ordineEntity.getIdPiatto())
65         ).andExpect(
66             MockMvcResultMatchers.jsonPath("$.stato").value(
67             ordineEntity.getStato())
68         ).andExpect(
69             MockMvcResultMatchers.jsonPath("$.urgenzaCliente").
70             value(ordineEntity.getUrgenzaCliente())));
71     }
72     ...
73 }
```

Codice 2.42: Classe Test per il REST Controller di test TestControllerTests.java

Vengono mostrati per semplicità solamente i casi di test che riguardano la API di GET (`getOrdine(int id)`) che richiede l'ordine dato un id), si testa lo stato della risposta e l'oggetto ricevuto, i restanti casi di test sono analoghi. Si nota come viene utilizzato mockMVC: si utilizza `perform()` per simulare richieste HTTP, `andExpect()` per verificare lo stato della risposta e `andDO()` per ispezionare la risposta.

Step 6 - Postman: A questo punto vengono testate le API tramite Postman, si rimanda nuovamente alla sezione 2.10 a pagina 91 per una documentazione dettagliata di ogni API. È stato creato un workspace condiviso tra i membri del team, nel quale si sono salvate tutte le API da testare, viene mostrato un esempio applicativo (in Figura 2.31):

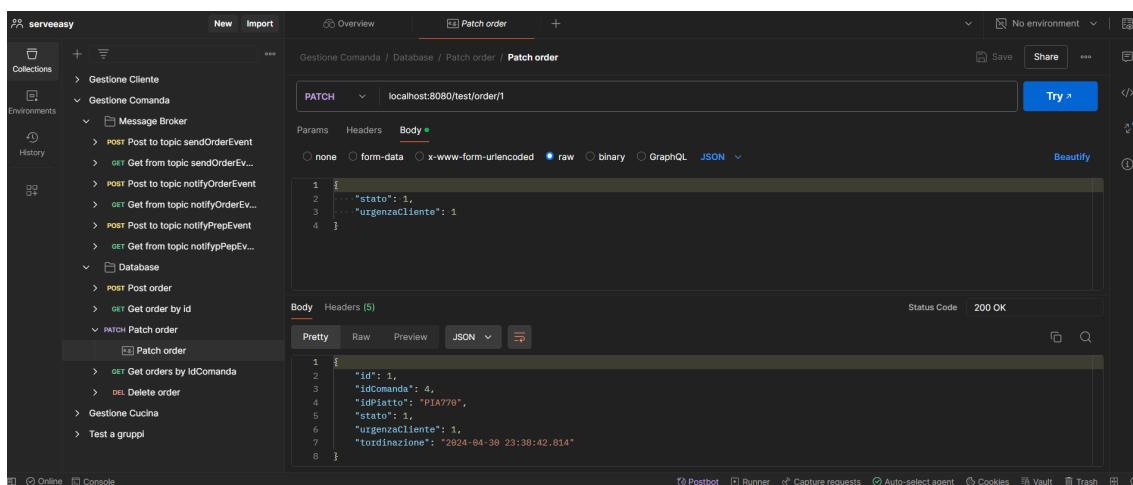


Figura 2.31: Esempio applicativo Postman

In questo esempio viene effettuata una richiesta di PATCH di aggiornamento parziale di un ordine, viene specificata la variabile di percorso `id=1` e il corpo della richiesta, ossia il JSON con il quale sono descritti i campi di `stato` e `urgenzaCliente` da aggiornare. Si può notare il codice di risposta 200 OK che viene restituito quando una richiesta HTTP è stata completata con successo e il corpo della risposta, ovvero un JSON contenente l'oggetto `OrdineDTO` serializzato.

2.8.9 Definizione Domain

Eseguiti i test sulle funzionalità in rete dei layer Interface e Infrastructure, si è passati all'attuale implementazione di una elementare business logic. Attualmente, l'obiettivo principale è la verifica del passaggio dati da un componente ad un altro: i componenti di dominio, per mezzo delle port, devono poter ricevere e trasmettere in maniera "cieca", ignorando qualsiasi attuale implementazione per la ricetrasmissione.

Soffermandoci sul punto **L** dei principi **SOLID**, l'annotazione `@Autowired` ha permesso una maggiore enfatizzazione del concetto: le classi della business logic dichiarano

l’interfaccia che utilizzano per interagire col livello Infrastructure, ma *senza sapere quale implementazione*, poichè quest’ultima è iniettata automaticamente da Spring Boot.

Analogalmente, sempre grazie all’annotazione `@Autowired`, anche il livello Interfaccia comunicherà verso il dominio attraverso una porta, senza sapere *chi sta offrendo* le funzionalità.

Attualmente, per motivi di debug, sono state inserite variabili nel dominio come `topic_notifyPrepEvent`, la quale esplicita il topic kafka, ma non lede i principi SOLID almeno dal punto di vista strutturale: il valore non è stato iniettato dal altri livelli ma, bensì, è stato estratto dal *application.properties*.

```

1  @Service
2  @Log
3  public class GestionePrioritaOrdini implements ClientePort {
4      private final DataPort dataPort;
5      private final AlgIF algIF;
6      @Value("${spring.kafka.consumer.gestioneCliente.topic}")
7      private String topic_notifyPrepEvent;
8
9      @Autowired
10     public GestionePrioritaOrdini(DataPort dataPort, AlgIF algIF) {
11         this.dataPort = dataPort;
12         this.algIF = algIF;
13     }
14
15     @Override
16     public void notifyOrder(NotificaOrdineDTO notificaOrdineDTO) {
17         [...]
18         extractOrderAndSendToHeap(notificaOrdineDTO);
19     }
20
21     private void extractOrderAndSendToHeap(NotificaOrdineDTO
22     notificaOrdineDTO){
23
24         Optional<OrdineEntity> no = dataPort.getOrderById(
25             notificaOrdineDTO.getId());
26         if(no.isPresent()){
27             log.info( "pushing " + no.get() + "to heap!");
28             algIF.pushNewOrder(no.get());
29         }else
30             log.info( "oggetto Ordine non trovato!!");
31     }
32 }
```

Codice 2.43: classe GestionePrioritàOrdini.java nel dominio di GestioneComanda

2.9 Deployment dei microservizi con Docker

2.9.1 Healthcheck

L'healthcheck è una funzionalità di Docker che permette di definire dei controlli di salute sul container: specificando un comando da mandare in azione periodicamente, Docker controlla se il container è "vivo e in salute". Per capire se l'applicazione è sia "viva" (quindi online) che "in salute" (e quindi operativa), risulta cruciale la scelta del comando da mandare periodicamente in azione.

Alcuni applicativi, come il DBMS MariaDB adottato, contengono degli script dedicati al controllo della salute del sistema (nel caso di MariaDB, essa contiene nel suo filesystem *healthcheck.sh*, dedicato proprio a questa operazione), mentre altri utilizzano delle operazioni note.

I microservizi in Spring Boot, in particolare, hanno la possibilità di installare la dipendenza *Actuator*^[23] che consente l'healthcheck. Tale dipendenza espone una API apposita per restituire lo stato in salute dell'applicativo. È stato possibile quindi utilizzare l'API `/actuator/health` del microservizio per effettuare un controllo periodico del suo stato in salute.

L'healthcheck può inoltre servire per valutare la salute complessiva della rete di microservizi: è possibile introdurre un vincolo di dipendenza tra i componenti di sistema affinchè un certo container vada online solo se i container da cui è dipendente sono "vivi e in salute".

2.9.2 Setup della rete di microservizi

Viene impostato il *docker-compose.yaml* eslicitando i microservizi che compongono l'architettura. Spring Boot dà la possibilità di impostare le variabili ambientali sui container Docker Compose, in modo da sovrascrivere le proprietà definite nel *application.yaml* e *application.properties*. In questo modo, è possibile riadattare l'esecuzione dei microservizi per assumere un'impostazione su misura per la runtime in rete.

1 **GestioneCucina:**
 2 **image:** gchirico1/gestione_cucina:latest
 3 **container_name:** GestioneCucina
 4 **restart:** always
 5 **environment:**
 6 **SERVER_PORT:** 8080
 7 **SERVER_ADDRESS:** 0.0.0.0
 8 **SPRING_DATASOURCE_URL:** jdbc:mariadb://db:3306/serveeasy
 9 **SPRING_KAFKA_BOOTSTRAP-SERVERS:** broker:29092 #plaintext_internal
 10 **depends_on:**

```

11   broker:
12     condition: service_healthy
13
14   db:
15     condition: service_healthy
16
17   healthcheck:
18     test: "curl --fail --silent localhost:8080/actuator/health |"
19       grep UP || exit 1"
20     interval: 20s
21     timeout: 5s
22     start_period: 40s
23
24   expose:
25     - "8080"
26
27   ports:
28     - "8082:8080"

```

Codice 2.44: GestioneCucina con healthcheck, dipendenze e variabili sovrascritte

2.9.3 Attivazione della rete di microservizi

Oltre al comando `docker compose up`, è possibile attivare e gestire la rete da IntelliJ IDEA. Quest'ultima modalità è risultata molto più comoda per il controllo della salute e delle attività della rete: il cruscotto offerto dall'IDE consente, infatti, una visuale completa e dettagliata di ogni log in tempo reale.

Attivato il `docker-compose.yml`, vengono scaricati i microservizi da Dockerhub e attivati i container seguendo la sequenza specificata attraverso le dipendenze.

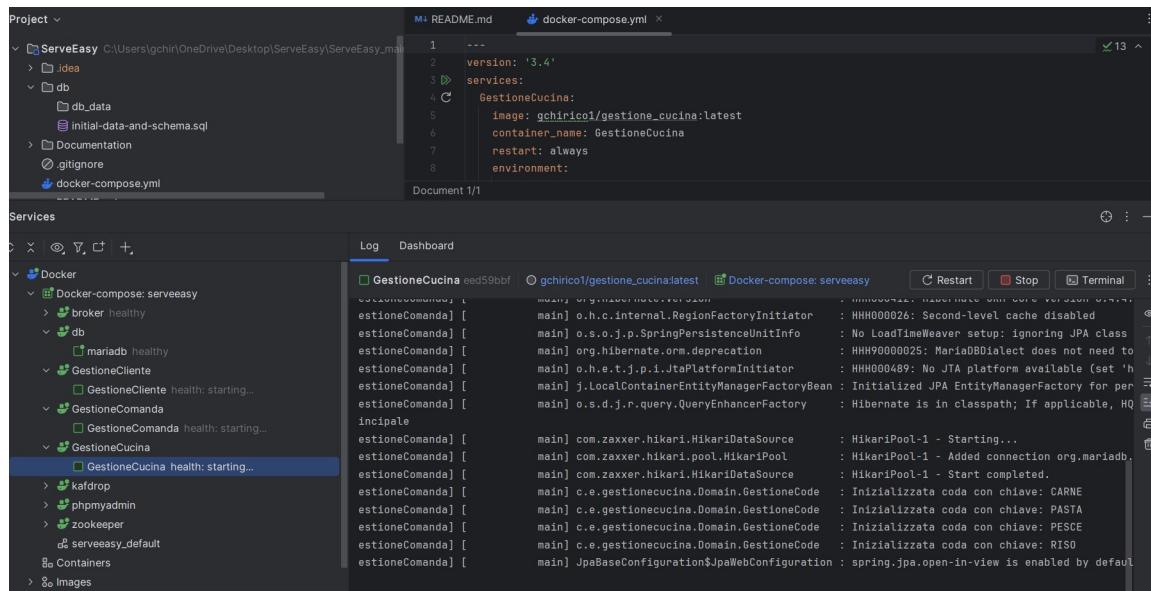


Figura 2.32: Startup della rete, inizializzazione code in GestioneCucina

2.10 Documentazione delle API

Le API esposte dai microservizi sono state testate tramite [postman](#) utilizzato in locale tramite [postman agent](#) creando un workspace condiviso tra il team.

Come discusso nella sezione 2.8.8 a pagina 76, il microservizio GestioneComanda è sprovvisto di un componente HTTP Controller nella sua Interfaccia (che contiene solo EventController), è quindi stato creato un controller di TEST per interagire direttamente con i componenti del servizio ai soli fini di test.

Viene di seguito allegata la documentazione delle API redatta utilizzando lo strumento [documenter.getpostman](#), link ai documenti ufficiali con anche esempi:

- Gestione Comanda: <https://documenter.getpostman.com/view/32004409/2sA3JDhkaG>
- Gestione Cliente: <https://documenter.getpostman.com/view/32004409/2sA3JFBQDv>
- Gestione Cucina: <https://documenter.getpostman.com/view/32004409/2sA3JF9iav>

Gestione Comanda

Il microservizio Gestione Comanda si occupa principalmente di gestire gli ordini dei clienti (microservizio GestioneCliente) e fornire alla cucina (microservizio GestioneCucina) gli ordini da preparare.

La comunicazione con gli altri microservizi avviene tramite Message Broker come segue:

- Il microservizio GestioneCliente comunica verso GestioneComanda tramite il topic Kafka NotifyOrderEvent.
- Il microservizio GestioneComanda comunica verso GestioneCucina tramite il topic Kafka SendOrderEvent.
- Il microservizio GestioneCucina comunica verso GestioneComanda tramite il topic Kafka NotifyPrepEvent.

Il microservizio GestioneComanda è sprovvisto di un componente HTTP Controller nella sua Interfaccia (che contiene solo EventController), viene quindi creato un controller di TEST per interagire direttamente con i componenti del servizio ai soli fini di test.

Le API di test riguardano principalmente il Message Broker e il DataBase.

Message Broker

E' possibile interagire direttamente con i tre topic (NotifyOrderEvent, SendOrderEvent, NotifyPrepEvent) tramite operazioni di GET e di POST:

- GET: le chiamate GET all'indirizzo `.../test/{topic}` restituiscono l'ultimo messaggio passato sul topic specificato;
 - POST: le chiamate POST all'indirizzo `.../test/{topic}` permettono di iniettare dall'esterno dei messaggi sul topic specificato, necessitano di un corpo in formato JSON che equivale alla serializzazione dell'oggetto che si aspetta quel topic.
-

POST Post to topic sendOrderEvent

localhost:8080/test/sendorderevent

API di POST con la quale è possibile iniettare all'interno del broker oggetti al fine di test.

Si testa il topic sendOrderEvent da gestione comanda verso gestione cucina.

Parametri della richiesta (body JSON):

- `id` (numero intero, opzionale) : Identificatore dell'ordine (autogenerato in ogni caso dal sistema)
- `idComanda` (numero intero, obbligatorio) : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` (stringa, obbligatorio) : Identificatore del piatto ordinato dal cliente
- `stato` (numero intero tra 0 e 3, obbligatorio) : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` (stringa timestamp di pattern "yyyy-MM-dd HH:mm:ss.SSS", opzionale): Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` (numero intero tra 0 e 2, obbligatorio) : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` : Identificatore del piatto ordinato dal cliente
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Body raw (json)

json

```
{
  "idComanda":7,
  "idPiatto":"SUH724",
  "stato":1,
  "urgenzaCliente":0
}
```

GET Get from topic sendOrderEvent

localhost:8080/test/sendorderevent

API di GET con la quale è possibile ottenere l'ultimo messaggio letto sul topic SendOrderEvent.

Si testa il topic notifyPrepEvent da gestione cucina verso gestione comanda

Risposta:

json

```
{  
    "id": 7,  
    "idComanda": 7,  
    "idPiatto": "SUH724",  
    "stato": 1,  
    "urgenzaCliente": 0,  
    "tordinazione": "2024-04-30 22:03:17.199"  
}
```

POST Post to topic notifyOrderEvent

localhost:8080/test/notifyorderevent

API di POST con la quale è possibile iniettare all'interno del broker oggetti al fine di test.

Si testa il topic notifyOrderEvent da gestione cliente verso gestione comanda.

Parametri della richiesta (body JSON):

- `id` (numero intero, obbligatorio) : Identificatore dell'ordine
- `idComanda` (numero intero, obbligatorio) : Identificatore della comanda di cui l'ordine fa parte

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte

Body raw (json)

json

```
{
  "id": 1,
  "idComanda": 4
}
```

GET Get from topic notifyOrderEvent

localhost:8080/test/notifyorderevent

Espone una API di GET con la quale è possibile ottenere l'ultimo messaggio letto sul topic NotifyOrderEvent.

Si testa il topic notifyPrepEvent da gestione cucina verso gestione comanda.

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte

POST Post to topic notifyPrepEvent

localhost:8080/test/notifyprepevent

API di POST con la quale è possibile iniettare all'interno del broker oggetti al fine di test.

Si testa il topic notifyPrepEvent da gestione cucina verso gestione comanda.

Parametri della richiesta (body JSON):

- `id` (numero intero, obbligatorio) : Identificatore dell'ordine
- `idComanda` (numero intero, obbligatorio) : Identificatore della comanda di cui l'ordine fa parte
- `stato` (numero intero tra 0 e 3, obbligatorio) : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in

preparazione, 3: Ordine preparato)

Body raw (json)

json

```
{  
    "id": 1,  
    "idComanda": 4,  
    "stato": 2  
}
```

GET Get from topic notifyPepEvent

localhost:8080/test/notifyprepevent

API di GET con la quale è possibile ottenere l'ultimo messaggio letto sul topic NotifyPreEvent.

Si testa il topic notifyPreEvent da gestione cucina verso gestione comanda.

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)

Database

Le API di test verso il database riguardano unicamente l'entità Ordine, vengono testati:

- l'inserimento nel database;
- la ricerca di un elemento;
- la modifica parziale di un elemento;
- la ricerca di tutti gli ordini per una data comanda;
- la rimozione di un ordine

POST Post order

```
localhost:8080/test/order
```

Salva nel database l'oggetto ordine dato un ordineDTO

Parametri della richiesta (body JSON):

- `id` (numero intero, opzionale) : Identificatore dell'ordine (autogenerato in ogni caso dal sistema)
- `idComanda` (numero intero, obbligatorio) : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` (stringa, obbligatorio) : Identificatore del piatto ordinato dal cliente
- `stato` (numero intero tra 0 e 3, obbligatorio) : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` (stringa timestamp di pattern "yyyy-MM-dd HH:mm:ss.SSS", opzionale): Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` (numero intero tra 0 e 2, obbligatorio) : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` : Identificatore del piatto ordinato dal cliente
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Body raw (json)

```
json
```

```
{
  "idComanda":4,
  "idPiatto":"PIA770",
  "stato":0,
  "urgenzaCliente":1
}
```

3

GET Get order by id

```
localhost:8080/test/order/1
```

Restituisci l'ordine corrispondente all'id dato in input

Parametri della query string:

- `id` (obbligatorio): Identificatore dell'ordine da recuperare

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` : Identificatore del piatto ordinato dal cliente
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

PATCH Patch order

```
localhost:8080/test/order/1
```

Aggiornamento parziale dell'entità ordine, è possibile fornire solamente gli oggetti da aggiornare.

Parametri della query string:

- `id` (obbligatorio): Identificatore dell'ordine da recuperare

Parametri della richiesta (body JSON):

- `stato` (numero intero tra 0 e 3, opzionale) : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `urgenzaCliente` (numero intero tra 0 e 2, opzionale) : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` : Identificatore del piatto ordinato dal cliente
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Body raw (json)

json

```
{  
  "stato": 1,  
  "urgenzaCliente": 1  
}
```

GET Get orders by IdComanda

localhost:8080/test/orders/4

Restituisce una lista con tutti gli ordini relativi a una data comanda

Parametri della query string:

- `idComanda` (obbligatorio): Identificatore della comanda di cui gli ordini fanno parte

Risposta:

Lista di oggetti con i seguenti parametri:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` : Identificatore del piatto ordinato dal cliente
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

DELETE Delete order

localhost:8080/test/order/8

Cancella l'ordine con il dato ID dal database

Parametri della query string:

- `id` (obbligatorio): Identificatore dell'ordine da recuperare

Risposta:

Plain Text

204 No Content

Gestione Cliente

Il microservizio Gestione Cliente si occupa principalmente di gestire l'interazione di un cliente col sistema e di gestire la sua relativa comanda di ordini.

La comunicazione con gli altri microservizi avviene tramite Message Broker come segue:

- Il microservizio GestioneCliente comunica verso GestioneComanda tramite il topic Kafka NotifyOrderEvent.

Le API riguardano l'interazione del cliente con il servizio

GET Get Menù

localhost:8080/cliente/menu



API di Get con la quale si richiede il menù (lista di piatti disponibili)

Risposta:

Lista di oggetti piatto:

- `id` : identificativo del piatto
 - `idIngPrinc` : identificativo dell'ingrediente principale
 - `descrizione` : descrizione del piatto
 - `prezzo` : prezzo del piatto
 - `tPreparazione` : tempo medio di preparazione del piatto
-

GET Get piatto from menù

localhost:8080/cliente/menu/car123

API di Get con la quale si richiede uno specifico piatto dal menù

Parametri della query string:

- `idpiatto` (obbligatorio): Identificatore del piatto da recuperare

Risposta:

- `id` : identificativo del piatto
 - `idIngPrinc` : identificativo dell'ingrediente principale
 - `descrizione` : descrizione del piatto
 - `prezzo` : prezzo del piatto
 - `tPreparazione` : tempo medio di preparazione del piatto
-

GET Get new Comanda from Cliente

localhost:8080/cliente/tavolo1/comanda/new

API di Get con la quale si richiede una nuova comanda per un determinato cliente

Parametri della query:

- `idCliente` (obbligatorio, stringa) : identificativo del cliente

Risposta:

- `id` : identificativo della comanda
 - `idCliente` : identificativo del cliente
 - `codicePagamento` : codice di pagamento
 - `totaleScontrino` : costo totale dei piatti ordinati a scontrino
-

GET Get Comanda attiva from Cliente

localhost:8080/cliente/tavolo1/comanda/attiva

API di Get con la quale si richiede la comanda attiva per un dato cliente

Parametri della query:

- `idCliente` (obbligatorio, stringa) : identificativo del cliente

Risposta:

- `id` : identificativo della comanda
- `idCliente` : identificativo del cliente
- `codicePagamento` : codice di pagamento
- `totaleScontrino` : costo totale dei piatti ordinati a scontrino

POST Post new Order by Cliente

localhost:8080/cliente/order/add

API di Post con la quale si salva un ordine per un dato cliente all'interno del sistema

Parametri del body:

- `idcliente` (stringa, obbligatorio) : identificativo del cliente
- `idpiatto` (stringa, obbligatorio) : identificativo del piatto
- `urgenzacliente` (numero, obbligatorio): Attributo urgenza del cliente(0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` : Identificatore del piatto ordinato dal cliente
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Body urlencoded

<code>idcliente</code>	tavolo1
<code>idpiatto</code>	CAR123
<code>urgenzacliente</code>	0

GET Get order

```
localhost:8080/cliente/order?id=2
```

API di Get con la quale si richiede un ordine specifico

Parametri della query:

- `id` (numero, obbligatorio) : identificativo dell'ordine richiesto

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` : Identificatore del piatto ordinato dal cliente
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

PARAMS

<code>id</code>	2
-----------------	---

GET Get Order status

```
localhost:8080/cliente/order/status?id=2
```

API di Get con la quale si richiede lo stato di un ordine specifico

Parametri della query:

- `id` (numero, obbligatorio) : identificativo dell'ordine richiesto

Risposta:

- `ordine` : Identificatore dell'ordine
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in

preparazione, 3: Ordine preparato)

PARAMS

id	2
-----------	---

GET Get orders from cliente

localhost:8080/cliente/tavolo1/orders

API di Get con la quale si richiede una lista di tutti gli ordini per un dato cliente

Parametri della query string:

- `idCliente` (obbligatorio, stringa): Identificatore del cliente in questione

Risposta:

Lista di oggetti Ordine:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `idPiatto` : Identificatore del piatto ordinato dal cliente
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
- `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
- `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)

Gestione Cucina

Il microservizio Gestione Cucina si occupa principalmente di ricevere gli ordini da preparare da Gestione Comanda, disporli nella corretta coda di postazione e gestire l'interazione di queste postazioni.

La comunicazione con gli altri microservizi avviene tramite Message Broker come segue:

- Il microservizio GestioneComanda comunica verso GestioneCucina tramite il topic Kafka SendOrderEvent.
- Il microservizio GestioneCucina comunica verso GestioneComanda tramite il topic Kafka NotifyPrepEvent.

Le API riguardano l'interazione delle postazioni di lavoro con il servizio

GET Get CodaPostazione from Cucina

localhost:8080/cucina/codapostazione/riso

API di Get con la quale è possibile ottenere la coda della postazione corrispondente all'identificativo di ingrediente principale specificato

Parametri della query string:

- `ingredientePrincipale` (obbligatorio): identificativo della coda di postazione

Risposta:

- `ingredientePrincipale` : identificativo della coda di postazione
- `numeroOrdiniPresenti` : numero ordini presenti in coda
- `gradoRiempimento` : grado di riempimento attuale della coda
- `capacita` : capacita' massima della coda
- `queue` : coda di ordinazioni (lista di oggetti ordine)

GET Get nextOrder from CodaPostazione

localhost:8080/cucina/codapostazione/riso/nextorder

API di Get con la quale è possibile ricevere l'ordine che deve essere preparato per una determinata postazione della cucina

Parametri della query string:

- `ingredientePrincipale` (obbligatorio): identificativo della coda di postazione

Risposta:

- `id` : Identificatore dell'ordine
 - `idComanda` : Identificatore della comanda di cui l'ordine fa parte
 - `idPiatto` : Identificatore del piatto ordinato dal cliente
 - `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)
 - `tOrdinazione` : Istante temporale in cui viene effettuata l'ordinazione
 - `urgenzaCliente` : Attributo urgenza del cliente (0 : espresso non urgenza, 1 : espresso urgenza, 2 : default)
-

POST Post NotificaOrdine from cucina

localhost:8080/cucina/codapostazione/riso

API di Post con la quale è possibile notificare l'avvenuta preparazione di un ordine da parte di una determinata postazione della cucina

Parametri della richiesta (body JSON):

- `id` (numero intero, obbligatorio) : Identificatore dell'ordine
- `idComanda` (numero intero, obbligatorio) : Identificatore della comanda di cui l'ordine fa parte
- `stato` (numero intero tra 0 e 3, obbligatorio) : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)

Risposta:

- `id` : Identificatore dell'ordine
- `idComanda` : Identificatore della comanda di cui l'ordine fa parte
- `stato` : Stato dell'ordine (0: Ordine preso in carico, 1: Ordine in coda di preparazione, 2: Ordine in preparazione, 3: Ordine preparato)

Body raw (json)

json

```
{  
  "id" : 1,  
  "idComanda" : 7,  
  "stato" : 3  
}
```

2.11 Analisi statica CheckStyle

2.11.1 Introduzione

Garantire la qualità del codice sorgente è fondamentale per assicurare la stabilità, l'affidabilità e la manutenibilità delle applicazioni. Tra gli strumenti utilizzati per questo scopo, l'analisi statica del codice riveste un ruolo cruciale. In questo Progetto è quindi stato utilizzato **Checkstyle**, <https://checkstyle.sourceforge.io/>, che permette una valutazione automatica della conformità del codice e linee guida da seguire.

Questo documento si propone di fornire una panoramica dettagliata sull'utilizzo di Checkstyle per condurre analisi statiche del codice sorgente. Esploreremo le sue funzionalità, le principali regole di analisi implementate e i benefici derivanti dall'integrazione di questa pratica nella fase di sviluppo del software. Vengono ora mostrati alcuni report generati dai vari microservizi.

È inoltre possibile visualizzare per intero i report generati nei seguenti documenti:

- Gestione Comanda:

<https://giorgio-hash.github.io/ServeEasy/Report/GestioneComanda/site/checkstyle.html>

- Gestione Cliente:

<https://giorgio-hash.github.io/ServeEasy/Report/GestioneCliente/site/checkstyle.html>

- Gestione Cucina:

<https://giorgio-hash.github.io/ServeEasy/Report/GestioneCucina/site/checkstyle.html>

Per poter visualizzare i file di report online si è utilizzato il servizio offerto da GitHub: GitHub Pages[10], esso permette di ospitare siti web statici generati da repository GitHub pubbliche.

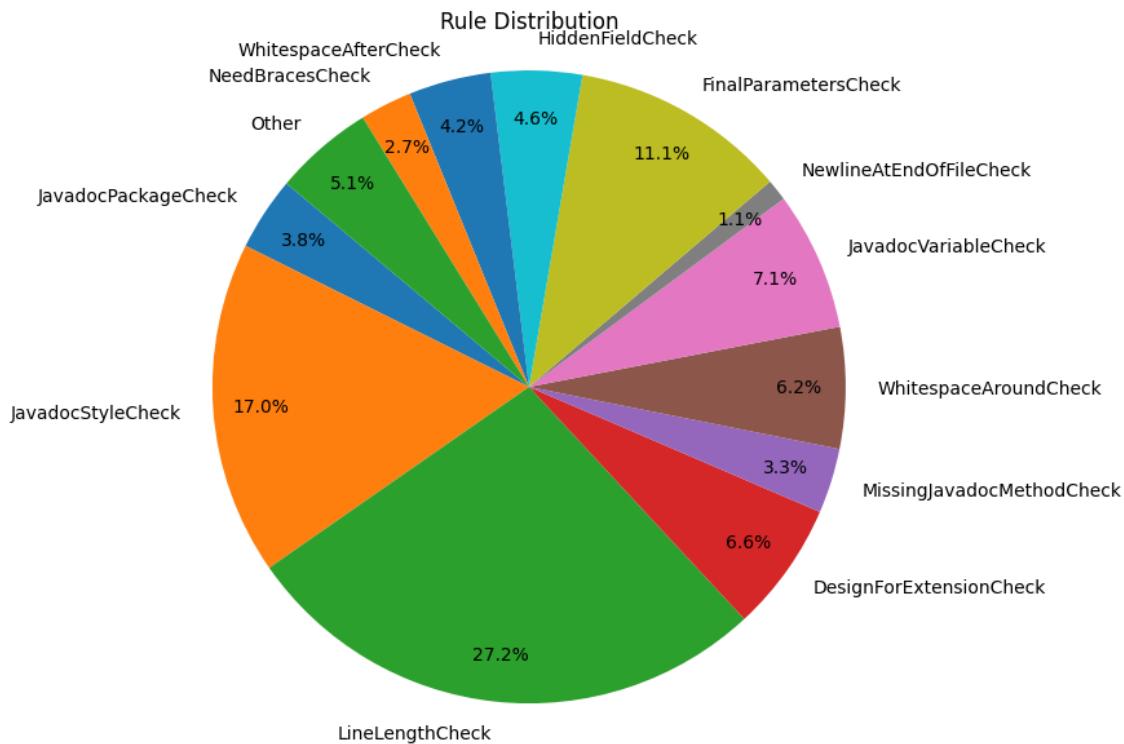
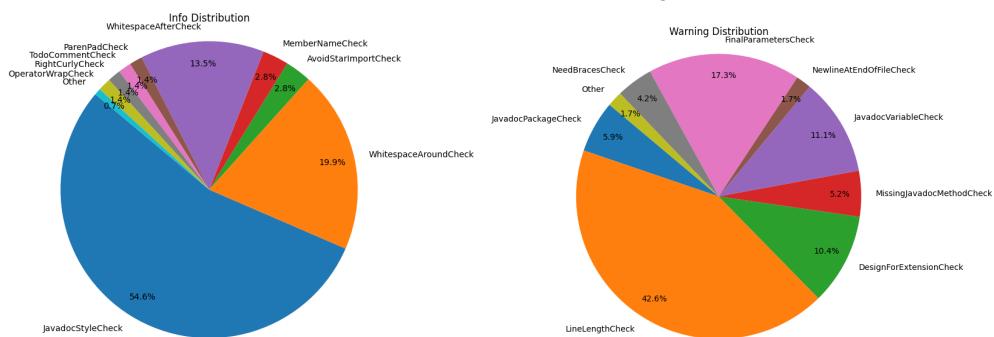
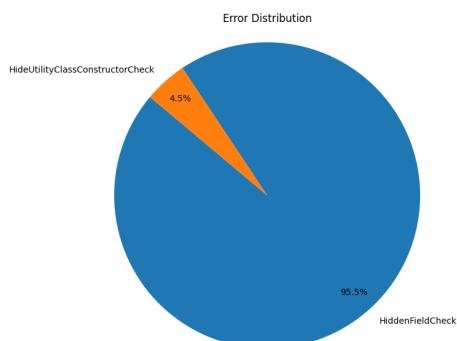
2.11.2 Report Gestione Comanda

Summary			
Files	Info	Warnings	Errors
29	141	289	22

Figura 2.33: Sommario di gestione comanda

Rules			
Category	Rule	Violations	Severity
blocks	NeedBraces	12	Warning
	RightCurly	2	Info
coding	HiddenField	21	Error
	MagicNumber	1	Warning
design	DesignForExtension	30	Warning
	HideUtilityClassConstructor	1	Error
imports	AvoidStarImport	4	Info
	UnusedImports	2	Warning
• processJavadoc: "false"			
javadoc	JavadocMethod	2	Warning
	JavadocPackage	17	Warning
	JavadocStyle	77	Info
	JavadocVariable	32	Warning
misc	MissingJavadocMethod	15	Warning
	FinalParameters	50	Warning
	NewlineAtEndOfFile	5	Warning
naming	TodoComment	2	Info
	LocalVariableName	1	Info
	MemberName	4	Info
sizes	LineLength	123	Warning
	• fileExtensions: "java"		
whitespace	OperatorWrap	2	Info
	ParenPad	2	Info
	WhitespaceAfter	19	Info
	WhitespaceAround	28	Info

Figura 2.34: Rules generate da gestione comanda

**Figura 2.35:** Grafico di tutte le regole**Figura 2.36:** Grafico delle info**Figura 2.37:** Grafico degli warnings**Figura 2.38:** Grafico degli errori

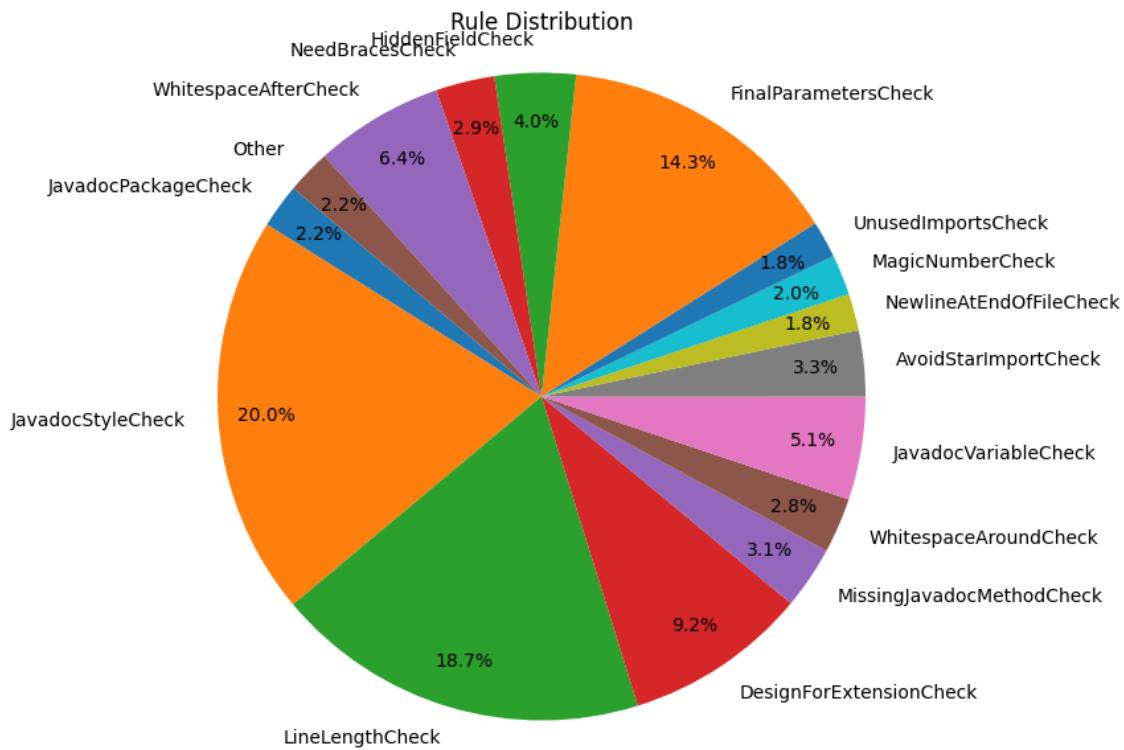
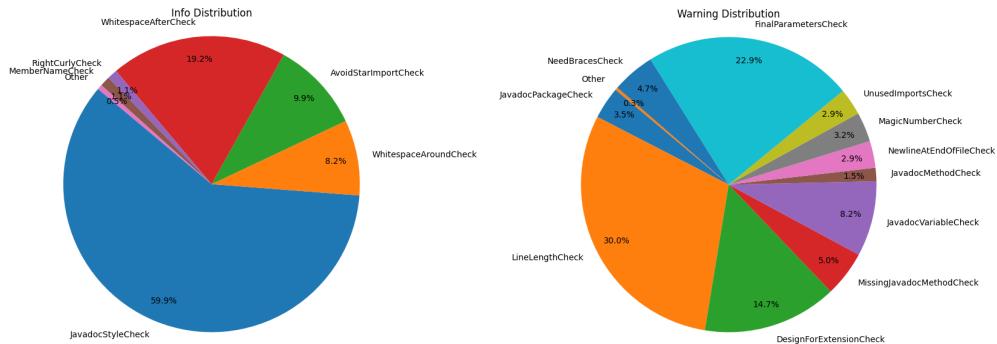
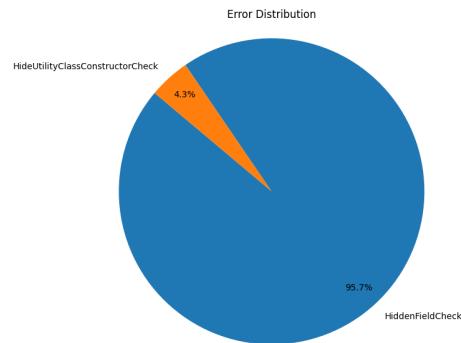
2.11.3 Report Gestione Cliente

Summary			
Files	Info	Warnings	Errors
45	182	340	23

Figura 2.39: Sommario di gestione cliente

Rules				
Category	Rule	Violations	Severity	
blocks	NeedBraces 	16	 Warning	
	RightCurly 	2	 Info	
coding	HiddenField 	22	 Error	
	MagicNumber 	11	 Warning	
design	DesignForExtension 	50	 Warning	
	HideUtilityClassConstructor 	1	 Error	
imports	AvoidStarImport 	18	 Info	
	UnusedImports 	10	 Warning	
• processJavadoc: "false"				
javadoc	JavadocMethod 	5	 Warning	
	JavadocPackage 	12	 Warning	
	JavadocStyle 	109	 Info	
	JavadocVariable 	28	 Warning	
• MissingJavadocMethod 				
misc	FinalParameters 	78	 Warning	
	NewlineAtEndOfFile 	10	 Warning	
modifier	RedundantModifier 	1	 Warning	
naming	LocalVariableName 	1	 Info	
	MemberName 	2	 Info	
sizes	LineLength 	102	 Warning	
	• fileExtensions: "java"			
whitespace	WhitespaceAfter 	35	 Info	
	WhitespaceAround 	15	 Info	

Figura 2.40: Rules generate da gestione cliente

**Figura 2.41:** Grafico di tutte le regole**Figura 2.42:** Grafico delle info**Figura 2.43:** Grafico degli warnings**Figura 2.44:** Grafico degli errori

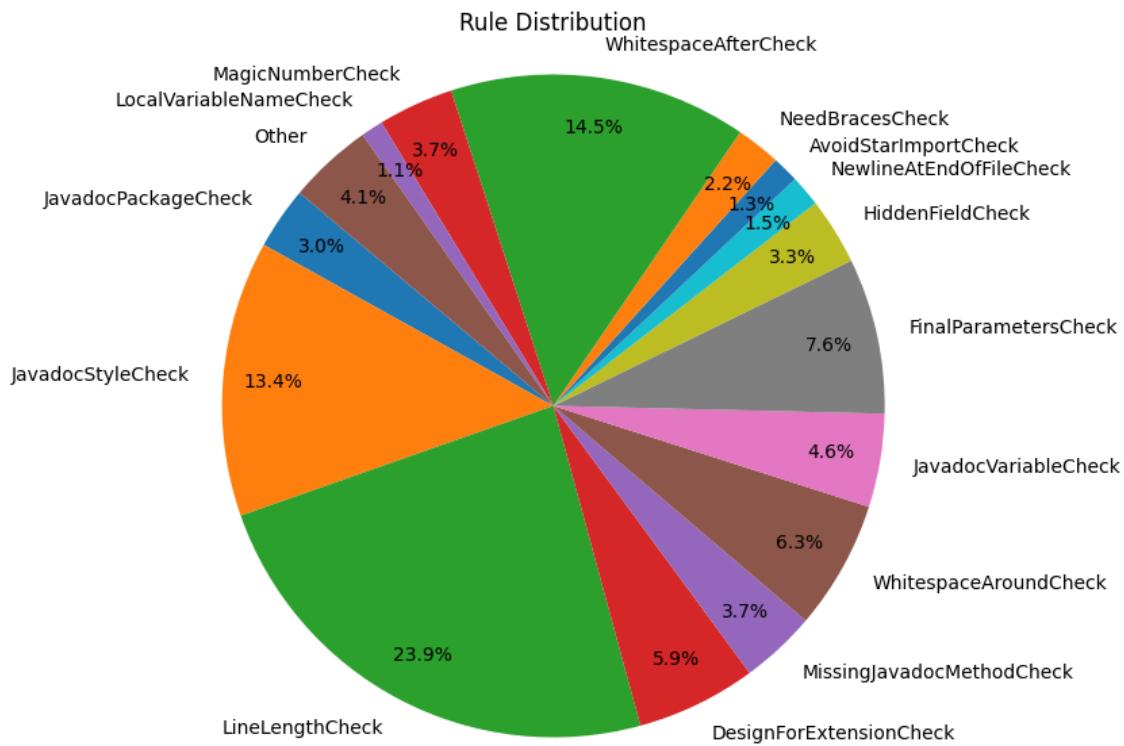
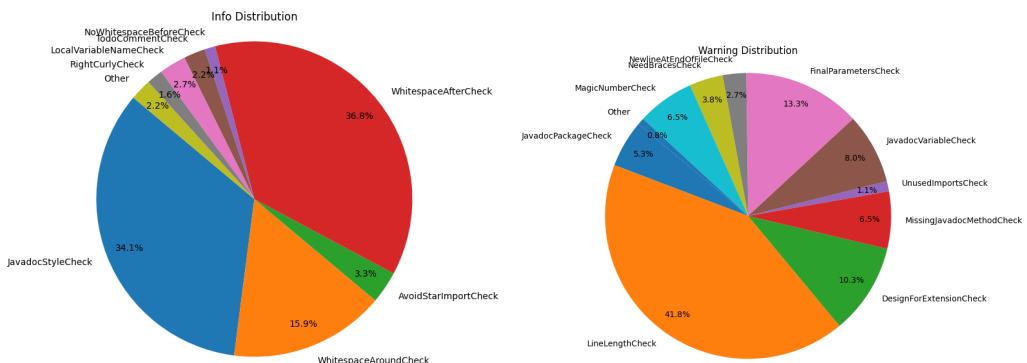
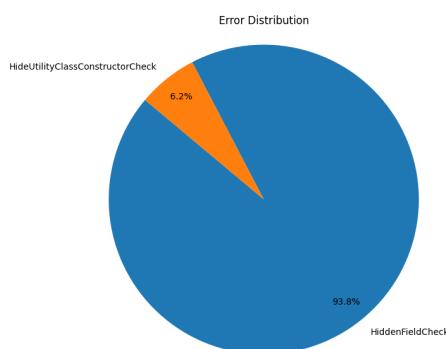
2.11.4 Report Gestione Cucina

Summary			
Files	Info	Warnings	Errors
29	182	263	16

Figura 2.45: Sommario di gestione cucina

Rules				
Category	Rule	Violations	Severity	
blocks	LeftCurly ↗	1	Info	
	NeedBraces ↗	10	Warning	
	RightCurly ↗	3	Info	
coding	HiddenField ↗	15	Error	
	MagicNumber ↗	17	Warning	
design	DesignForExtension ↗	27	Warning	
	HideUtilityClassConstructor ↗	1	Error	
imports	AvoidStarImport ↗	6	Info	
	UnusedImports ↗	3	Warning	
	• processJavadoc: "false"			
javadoc	JavadocMethod ↗	2	Warning	
	JavadocPackage ↗	14	Warning	
	JavadocStyle ↗	62	Info	
	JavadocVariable ↗	21	Warning	
	MissingJavadocMethod ↗	17	Warning	
misc	FinalParameters ↗	35	Warning	
	NewlineAtEndOfFile ↗	7	Warning	
	TodoComment ↗	4	Info	
naming	LocalVariableName ↗	5	Info	
sizes	LineLength ↗	110	Warning	
	• fileExtensions: "java"			
whitespace	MethodParamPad ↗	1	Info	
	NoWhitespaceBefore ↗	2	Info	
	OperatorWrap ↗	1	Info	
	ParenPad ↗	1	Info	
	WhitespaceAfter ↗	67	Info	
	WhitespaceAround ↗	29	Info	

Figura 2.46: Rules generate da gestione cucina

**Figura 2.47:** Grafico di tutte le regole**Figura 2.48:** Grafico delle info**Figura 2.49:** Grafico degli warnings**Figura 2.50:** Grafico degli errori

2.11.5 Specifica

Di seguito vengono ora date le Specifiche segnalate dai vari report.

Errors:

- La regola **HiddenField** controlla se una variabile locale o un parametro ha lo stesso nome di un campo (field) definito nella stessa classe.
- La regola **hideUtilityClassConstructor** si assicura che le classi di utilità non abbiano un costruttore pubblico. Le classi di utilità sono spesso utilizzate per raggruppare funzionalità comuni e non dovrebbero essere istanziate direttamente. Pertanto, i costruttori di queste classi dovrebbero essere privati o, se si desidera consentire l'ereditarietà, protetti.

Warning:

- La regola **NeedBraces** controlla se i blocchi di codice (if, else, for, while, ecc.) sono racchiusi tra parentesi graffe.
- La regola **MagicNumber** controlla se ci sono numeri letterali nel codice che non sono definiti come costanti. In altre parole, i “magic numbers” sono valori numerici che compaiono direttamente nel codice senza essere assegnati a una variabile o costante.
- La regola **designForExtension** controlla se le classi sono progettate per essere estese tramite sottoclassi. È particolarmente utile nei progetti di librerie (non nei progetti di applicazioni) che si preoccupano di seguire un design OOP ideale per garantire che le classi funzionino correttamente in tutti i casi, anche in caso di uso improprio.
- La regola **UnusedImports** verifica gli import nonutilizzati all'interno del codice.
- La regola **JavadocMethod** controlla la documentazione Javadoc di un metodo o di un costruttore.
- La regola **JavadocPackage** verifica che ogni pacchetto Java abbia un file Javadoc utilizzato per i commenti. Di default, consente solo un file package-info.java, ma può essere configurata per consentire un file “package.html”. Verrà segnalata una violazione se entrambi i file esistono, poiché ciò non è consentito dallo strumento Javadoc.

- La regola **JavadocVariable** controlla se una variabile ha un commento Javadoc. Viene segnalata una violazione se manca il commento Javadoc per qualsiasi membro di visibilità.
- La regola **MissingJavadocMethod** verifica la presenza di commenti Javadoc mancanti per metodi o costruttori.
- La regola **FinalParameters** verifica che i parametri per metodi, costruttori, blocchi catch e blocchi for-each siano dichiarati come final. Tuttavia, i metodi di interfaccia, astratti e nativi non vengono controllati: la parola chiave final non ha senso per i parametri dei metodi di interfaccia, astratti e nativi poiché non esiste alcun codice che potrebbe modificare il parametro.
- La regola **NewlineAtEndOfFile** verifica se i file terminano con un separatore di riga.
- La regola **LineLength** verifica se le righe sono troppo lunghe. Questo è importante perché le righe lunghe possono essere difficili da leggere, specialmente quando si stampa il codice o quando gli sviluppatori hanno uno spazio limitato sullo schermo (ad esempio, se l'IDE mostra altre informazioni come l'albero del progetto o la gerarchia delle classi).

Info:

- La regola **LeftCurly** controlla la posizione delle parentesi graffe aperte all'interno dei blocchi di codice.
- La regola **RightCurly** controlla se le parentesi graffe chiuse sono posizionate correttamente all'interno dei blocchi di codice.
- La regola **AvoidStarImport** controlla se ci sono dichiarazioni di importazione che utilizzano l'asterisco *. Importare tutte le classi da un pacchetto o membri statici da una classe porta a un accoppiamento stretto tra pacchetti o classi e potrebbe causare problemi quando una nuova versione di una libreria introduce conflitti di nomi.
- La regola **JavadocStyle** verifica che i commenti Javadoc siano ben formati. (punteggiatura alla fine della prima frase, verifica della presenza di descrizione, verifica dei tag html incompleti, verifica documentazione del pacchetto, tag html consentiti).
- La regola **TodoComment** verifica la presenza di commenti con la parola chiave “TODO:”. In realtà, è un matcher generico di pattern per i commenti Java. Per verificare altri pattern nei commenti Java, è possibile impostare la proprietà format.

L'utilizzo dei commenti “TODO:” è un ottimo modo per tenere traccia dei compiti da svolgere.

- La regola **LocalVariableName** verifica che i nomi delle variabili locali (variabili dichiarate all'interno di un metodo o di un blocco)siano conformi a un pattern specificato.
- La regola **MethodParamPad** controlla la spaziatura tra l'identificatore di una definizione di metodo, una definizione di costruttore,una chiamata di metodo o una chiamata di costruttore e la parentesi sinistra della lista dei parametri.
- La regola **NoWhitespaceBefore** verifica che non ci sia spazio bianco prima di un token specifico. In particolare, controlla che il token non sia preceduto da spazio bianco
- La regola **OperatorWrap** verifica come gli operatori sono posizionati rispetto alle righe di codice. In particolare, controlla se gli operatori dovrebbero essere posizionati sulla stessa riga o su una nuova riga. Questo è importante per mantenere la leggibilità del codice e per seguire le convenzioni di stile.
- La regola **ParenPad** verifica come le parentesi sono posizionate rispetto alle righe di codice. In particolare, controlla se le parentesi dovrebbero essere seguite da spazio o se devono essere adiacenti senza spazi.
- La regola **WhitespaceAfter** controlla come gli spazi bianchi sono posizionati rispetto ai token nel codice sorgente. In particolare,verifica se gli operatori dovrebbero essere seguiti da spazio o se devono essere adiacenti senza spazi.
- La regola **WhitespaceAround** verifica che un token sia circondato da spazio bianco. Questa regola si applica a vari contesti, come loop vuoti e lambda vuote.
- La regola **redundantModifier** è progettata per individuare e segnalare i modificatori ridondanti all'interno del codice sorgente Java. Questa regola controlla i modificatori di accesso, come public, protected,private, e anche i modificatori come final, abstract, static e strictfp.

2.11.6 Checkstyle

Installazione: Per poter utilizzare le funzionalità di checkstyle all'interno dell'IDEA itelliJ è stato installato un plugin disponibile all'interno del marketplace di IntelliJ, ovvero CheckStyle-IDEA[18]. Di seguito viene mostrata la schermata che permette l'installazione di quest'ultima.

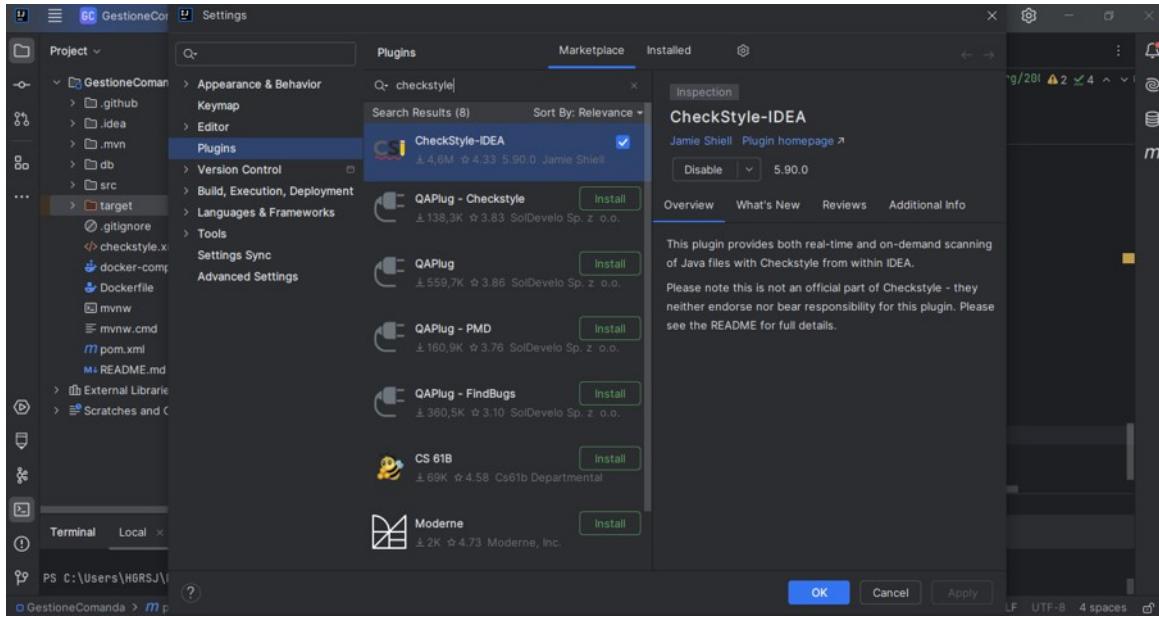


Figura 2.51: Schermata installazione plugin

Configurazione e personalizzazione: Per la configurazione del plugin di checkstyle in maven è stato introdotto all'interno del file **pom.xml** la seguente:

```

1 <plugin>
2     <groupId>org.apache.maven.plugins</groupId>
3     <artifactId>maven-checkstyle-plugin</artifactId>
4     <version>3.3.1</version>
5     <configuration>
6         <configLocation>checkstyle.xml</configLocation>
7     </configuration>
8     <executions>
9         <execution>
10            <goals>
11                <goal>checkstyle</goal>
12            </goals>
13        </execution>
14    </executions>
15 </plugin>
```

Codice 2.45: plugin Maven con Maven Checkstyle Plugin

Abbiamo quindi creato un file **Checkstyle.xml** personalizzando le regole di Sun Checkstyle, che definiscono un insieme di regole di programmazione per la scrittura del codice. Questo ci aiuta a adattare le regole alle nostre specifiche esigenze, garantendo uno stile di codifica coerente e di alta qualità. Le categorie di violazioni controllate sono: Blocks, coding, design, javadoc, imports, misc, naming, sizes, whitespace e modifier.

```
1 <?xml version="1.0"?>
```

```
2  <!DOCTYPE module PUBLIC
3      "-//Checkstyle//DTD Checkstyle Configuration 1.3//EN"
4      "https://checkstyle.org/dtds/configuration_1_3.dtd">
5
6  <!--
7      Checkstyle configuration that checks the sun coding
8  -->
9
10 <module name="Checker">
11
12     <property name="severity" value="error"/>
13
14     <property name="fileExtensions" value="java, properties, xml"/>
15
16     <!-- Excludes all 'module-info.java' files -->
17     <module name="BeforeExecutionExclusionFileFilter">
18         <property name="fileNamePattern" value="module\-\info\.java$"/>
19     </module>
20
21     <module name="SuppressionFilter">
22         <property name="file" value="${org.checkstyle.sun.
23             suppressionfilter.config}"
24             default="checkstyle-suppressions.xml" />
25         <property name="optional" value="true"/>
26     </module>
27
28     <!-- Checks that a package-info.java file exists for each package.
29         -->
30     <module name="JavadocPackage">
31         <property name="severity" value="warning"/>
32     </module>
33
34     <!-- Checks whether files end with a new line.
35         -->
36     <module name="NewlineAtEndOfFile">
37         <property name="severity" value="warning"/>
38     </module>
39
40     <!-- Checks that property files contain the same keys. -->
41     <module name="Translation"/>
42
43     <!-- Checks for Size Violations. -->
44     <module name="FileLength">
45         <property name="severity" value="warning"/>
46     </module>
```

```

44     <module name="LineLength">
45         <property name="fileExtensions" value="java"/>
46         <property name="severity" value="warning"/>
47     </module>
48
49     <!-- Checks for whitespace -->
50     <module name="FileTabCharacter">
51         <property name="severity" value="warning"/>
52     </module>
53
54     <module name="TreeWalker">
55
56         <!-- Checks for Javadoc comments. -->
57         <module name="InvalidJavadocPosition">
58             <property name="severity" value="warning"/>
59         </module>
60         <module name="JavadocMethod">
61             <property name="severity" value="warning"/>
62         </module>
63
64         <!-- Checks for Naming Conventions. -->
65         <module name="ConstantName">
66             <property name="severity" value="info"/>
67         </module>
68         <module name="LocalFinalVariableName">
69             <property name="severity" value="info"/>
70
71         <!-- Checks for imports -->
72         <module name="AvoidStarImport">
73             <property name="severity" value="info"/>
74         </module>
75         <module name="IllegalImport">
76
77         <!-- Checks for Size Violations. -->
78         <module name="MethodLength">
79             <property name="severity" value="info"/>
80         </module>
81         <module name="ParameterNumber">
82             <property name="severity" value="info"/>
83         </module>
84
85         <!-- Checks for whitespace -->
86         <module name="EmptyForIteratorPad">
87             <property name="severity" value="info"/>
88         </module>

```

```

89      <module name="GenericWhitespace">
90          <property name="severity" value="info"/>
91      </module>

92
93      <!-- Checks for blocks. You know, those {}'s -->
94      <module name="AvoidNestedBlocks">
95          <property name="severity" value="warning"/>
96      </module>
97      <module name="EmptyBlock">
98          <property name="severity" value="error"/>
99      </module>

100
101     <!-- Checks for class design -->
102     <module name="DesignForExtension">
103         <property name="severity" value="warning"/>
104     </module>
105     <module name="FinalClass">
106         <property name="severity" value="warning"/>
107     </module>

108
109
110     <module name="SuppressionXpathFilter">
111         <property name="file" value="${org.checkstyle.sun.
112             suppressionxpathfilter.config}"
113                 default="checkstyle-xpath-suppressions.xml" />
114         <property name="optional" value="true"/>
115     </module>

116     </module>
117
118 </module>

```

Codice 2.46: Personalizzazioni regole di Sun Checkstyle

Esecuzione: Avvenuta la fase di installazione e configurazione è ora possibile identificare e correggere le violazioni di stile definite nel file di configurazione.

Per eseguire il controllo di stile è sufficiente eseguire il comando:

```
1 mvn site
```

Codice 2.47: Avvio controllo checkstyle

Esso permetterà di generare all'interno della directory **target > site** contenente i file html e xml con i report generati.

2.11.7 Generazione Grafi

Al fine di poter visualizzare con più semplicità i file di report sono stati generati dei grafici a torta che mostrano il tipo di errore rilevato e la percentuale delle volte in cui è stato commesso. Per creare questi grafici abbiamo creato un semplice script python da allegare al progetto di ogni microservizio.

Script Python

Questo script si basa sul file "checkstyle-result.xml" generato dal report di checkstyle e posizionato nella cartella target, quindi come prima cosa viene caricato questo file:

```
1 script_dir = os.path.dirname(os.path.abspath(__file__))
2 input_xml_file =
3     os.path.join(script_dir, "target", "checkstyle-result.xml")
```

Codice 2.48: Script Python - aggiunta checkstyle-result.xml

Successivamente si passa a fare il parsing di questo file, andando a scandire i vari elementi <error> per ogni singolo <file>, in particolare si classificano gli errori in base all'attributo severity che può essere info, warning ed error. In questo modo si creano 3 dizionari con questi tipi di valore di severità, oltre a un altro globale che contiene la somma di tutti e 3 gli errori chiamato rule, ed ad ogni occorrenza di un errore data dall'attributo source di error si aumenta il conteggio del dizionario alla chiave corrispondente a quel attributo. Come mostrato da questo pezzo di codice:

```
1 tree = Et.parse(xml_file)
2 root = tree.getroot()
3 error_counts = defaultdict(int)
4 for file in root.findall('file'):
5     for error in file.findall('error'):
6         if error.get('severity') == severity or skip is True:
7             source = error.get('source')
8             error_counts[error_type] += 1
```

Codice 2.49: Script Python - parsing checkstyle-result.xml

Successivamente si passa a convertire ogni dizionario in un file csv, il quale sarà costruito in modo tale da avere come prima colonna il tipo di errore e come seconda il numero di volte che è stato commesso quell'errore. Nel passo successivo questi file csv vengono convertiti in un grafico a torta utilizzando la libreria matplotlib.pyplot e salvati in un file png.

```
1 errors = list(error_counts.keys())
2 counts = list(error_counts.values())
```

```

3 [...] 
4 plt.figure(figsize=(10, 7))
5 patches, texts, _ = plt.pie(counts, labels=errors, autopct='%.1f%%',
6     startangle=140, pctdistance=0.85)
7 plt.axis('equal')
8 plt.title(title + " Distribution")
9 plt.savefig(output_file, bbox_inches='tight')

```

Codice 2.50: Script Python - Plot pie chart

I file csv e le immagini png vengono salvati nella cartella di percorso target/output/csv per i primi, mentre target/output/images per le seconde.

Come avviarlo: per poter avviare lo script python è necessario avere Python installato sul proprio PC[25] ed eseguire il seguente comando per installare le librerie necessarie:

```
1 pip install -r python/requirements.txt
```

Codice 2.51: Python - installare i pacchetti necessari

Successivamente eseguire il seguente per poter avviare lo script:

```
1 python main.py
```

Codice 2.52: Avvio script python

È possibile trovare i file generati in target/output dalla root del progetto.

Output: esempio di file csv in output:

```

1 Warning ,Count
2 JavadocPackageCheck ,17
3 LineLengthCheck ,123
4 DesignForExtensionCheck ,30
5 MissingJavadocMethodCheck ,15
6 UnusedImportsCheck ,2
7 JavadocVariableCheck ,32
8 JavadocMethodCheck ,2
9 NewlineAtEndOfFileCheck ,5
10 MagicNumberCheck ,1
11 FinalParametersCheck ,50
12 NeedBracesCheck ,12

```

Codice 2.53: file checkstyle_warning_severity_counts.csv warning di gestione comanda

Grafico a torta associato:

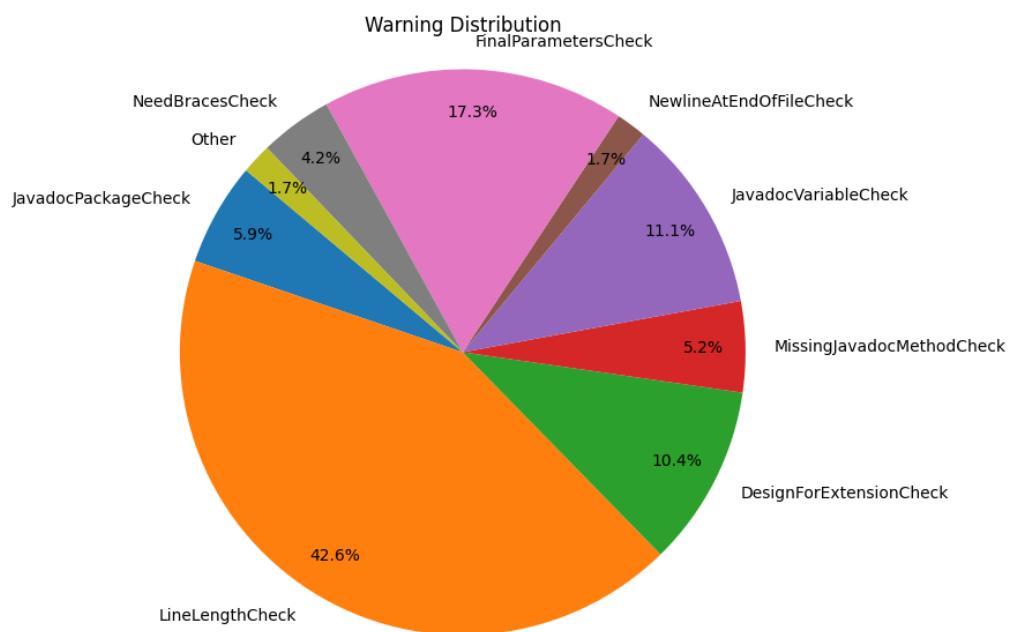


Figura 2.52: Grafico warnings di gestione comanda

2.12 Analisi statica SonarQube

A causa della mancanza di ulteriori analisi all'interno del plugIn di Checkstyle è stata inserito un ulteriore strumento, ovvero SonaQube [1].

Esso è uno strumento che consente ad un team di scrivere codice più pulito e più sicuro. Esso garantisce un'ispezione continua del codice e mette a disposizione migliaia di regole automatizzate finalizzate all'analisi statica del codice. Queste regole forniscono protezione al progetto esaminato e guidano il team di sviluppo.

La home page dello strumento fornisce una fotografia della situazione, in termini di qualità, del progetto sottoposto all'analisi.

Tramite la pagina Issues è possibile valutare in dettaglio ogni problematica trovata, quali sono le issue principali, dove si trovano nel codice e quando sono stati aggiunti. Per ciascun dominio SonarQube fornisce un diagramma a bolle che mette in correlazione diverse metriche.

2.12.1 SonarQube

Creazione Container SonarQube: Per poter utilizzare le funzionalità di sonarqube all'interno del nostro progetto è stato scelto l'implementazione attraverso offerta dal framework Docker, dove viene garantito un ambiente altamente personalizzabile, flessibile e di facile implementazione all'interno di ogni container. Una volta istanziati i volumi di Docker e modificato il *docker-compose.yaml*, la localhost utilizzata dal componente di sonarqube sarà la porta **9000**.

Installazione: Creati i volumi sono stati inseriti all'interno del *pom.xml* di maven i seguenti plugin.

```
1 <!-- JaCoCo test coverage -->
2 <plugin>
3   <groupId>org.jacoco</groupId>
4   <artifactId>jacoco-maven-plugin</artifactId>
5   <version>0.8.11</version>
6   <executions>
7     <execution>
8       <id>prepare-agent</id>
9       <goals>
10        <goal>prepare-agent</goal>
11      </goals>
12    </execution>
13    <execution>
14      <id>report</id>
```

```

15         <goals>
16             <goal>report</goal>
17         </goals>
18     </execution>
19   </executions>
20 </plugin>
21 </plugins>
22 <pluginManagement>
23   <plugins>
24     <plugin>
25       <!-- SonarQube -->
26       <groupId>org.sonarsource.scanner.maven</groupId>
27       <artifactId>sonar-maven-plugin</artifactId>
28       <version>3.4.0.905</version>
29     </plugin>
30   </plugins>
31 </pluginManagement>
32 </build>

```

Codice 2.54: Implementazioni pom.xml

Come si può notare è stato necessario inserire anche **Jacoco**[12] come plugin maven, ovvero una copertura di codice che misura la qualità di un programma che è stato effettivamente eseguito durante un test, così da determinare se il codice è stato testato in modo completo ed efficace.

Come usare SonarQube: Una volta fatto partire il container contenente SonarQube sarà necessario:

- accedere all’interfaccia localhost:9000;
- inserire user e password iniziali (username: admin / password: admin);
- modificare la propria password con una propria locale;
- creare un nuovo progetto locale impostando nomi e branch desiderati;
- generare un **token** locale e specificare maven come proprio framework;
- lanciare su cmd il proprio comando:

```

1 ./mvnw clean verify sonar:sonar -Dsonar.projectKey=
    GestioneCliente -Dsonar.projectName='GestioneCliente' -
    Dsonar.host.url=http://localhost:9000 -Dsonar.token=
    INSERIRE_PROPRI_TOKEN SONARQUBE

```

Codice 2.55: Avvio sonarqube

- attendere il fine compilazione e ricaricare la pagina localhost:9000.

2.12.2 Analisi report SonarQube

Mostriamo ora significative parti del report generato su alcuni dei nostri microservizi.

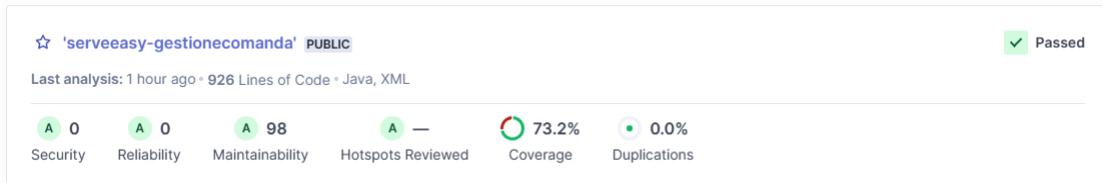


Figura 2.53: Analisys list gestione comanda

Nella specifica questi controlli servono per:

- **Security (Sicurezza):**

- La sicurezza del software riguarda la protezione contro le minacce esterne e interne, come hacking, malware e accessi non autorizzati.
- Include pratiche come la gestione delle vulnerabilità, l’implementazione di misure di protezione e l’esecuzione di test di sicurezza (penetration testing, security scans).

- **Reliability (Affidabilità):**

- L’affidabilità del software si riferisce alla capacità del software di funzionare correttamente e senza errori per un periodo di tempo specificato.
- Include la gestione degli errori, la ridondanza e la capacità del sistema di recuperare da fallimenti (fault tolerance).

- **Maintainability (Manutenibilità):**

- La manutenibilità si riferisce alla facilità con cui il software può essere modificato per correggere errori, migliorare le prestazioni o adattarsi a nuovi requisiti.
- Comprende aspetti come la leggibilità del codice, la modularità, la documentazione e l’aderenza a standard di codifica.

- **Hotspots Reviewed (Revisioni dei Punti Caldi):**

- I "punti caldi" sono parti del codice che sono frequentemente modificate o che presentano problemi ricorrenti.

- La revisione dei punti caldi implica l'analisi e l'ottimizzazione di queste aree critiche per migliorare la qualità del codice e ridurre il rischio di problemi futuri.

- **Coverage (Copertura):**

- La copertura del codice si riferisce alla percentuale di codice che viene eseguito durante i test.
- Include metriche come la copertura delle linee, delle funzioni e delle condizioni, aiutando a identificare le parti del codice che non sono state testate.

- **Duplications (Duplicazioni):**

- Le duplicazioni si riferiscono alla presenza di codice duplicato all'interno del progetto software.
- Il codice duplicato può aumentare il rischio di errori e rendere il mantenimento più difficile. Rimuovere le duplicazioni aiuta a mantenere il codice più pulito e manutenibile.

Alcune immagini create per mostrare i test di sonarqube che permettono di individuare le parti di maggior interesse. Esse Vengono rappresentate anche attraverso grafici di bole facilmente leggibili e interattivi.

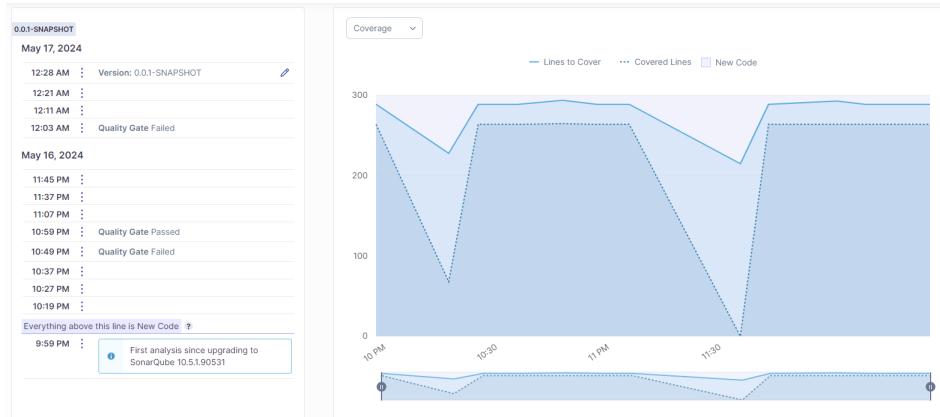


Figura 2.54: Coverage variata nel corso delle modifiche testate

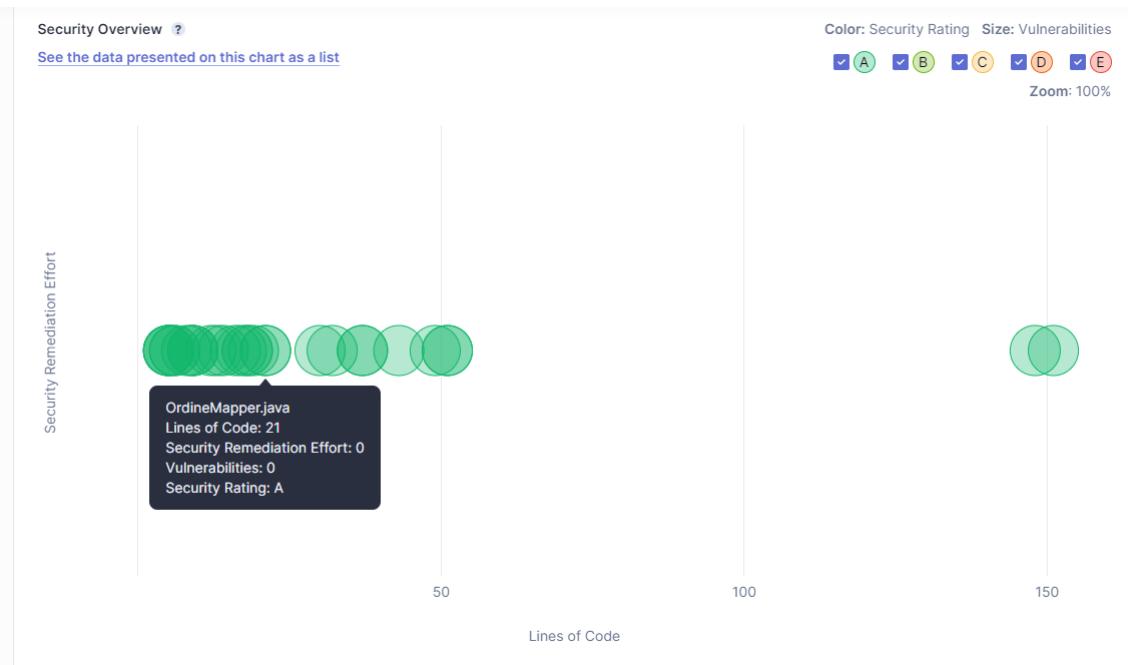
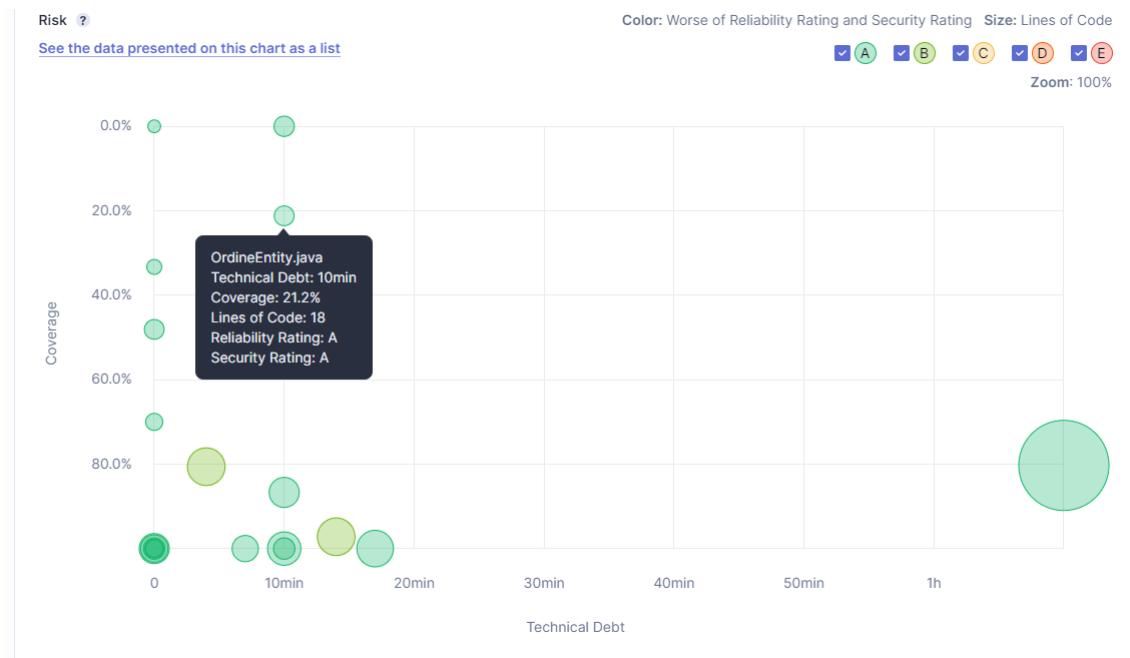
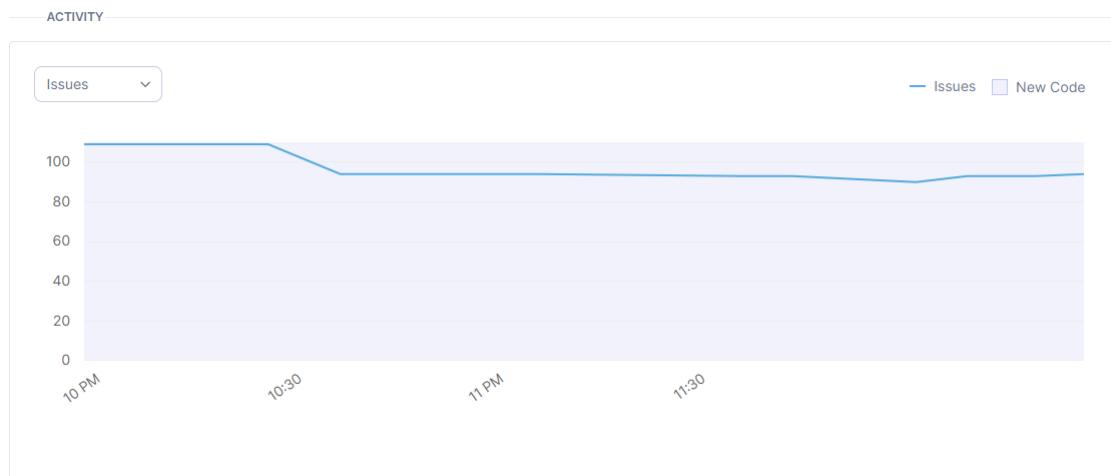


Figura 2.55: diagramma di bole sulla secutity

Una breve Panoramica delle rules di java esistenti su SonarQube, ognuna di esse può essere totalmente modificata.

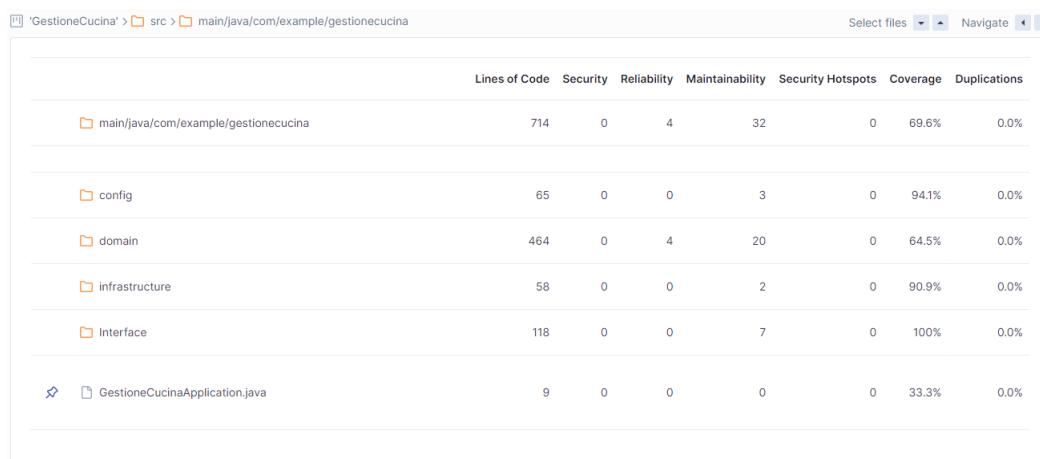
**Figura 2.56:** diagramma di bolle sui risk**Figura 2.57:** overview issues**Figura 2.58:** issues specifico

The screenshot shows a SonarQube interface for a Java file named `GestioneCucinaApplication.java`. The code editor displays several lines of Java code, with line 67 highlighted by a red squiggly underline. A tooltip box appears over the code, containing the message: "Provide the parametrized type for this generic." The code itself includes annotations like `@DeleteMapping` and `ResponseEntity`.

```

62 isaacm_ *
63     * @param id id dell'ordine da eliminare
64     * @return entità risposta che contiene la risposta HTTP associata
65     */
66     @DeleteMapping(path = "/order/{id}")
67     ResponseEntity deleteOrder(@PathVariable("id") int id);

```

Figura 2.59: view singolo issue**Figura 2.60:** gestione cucina overview

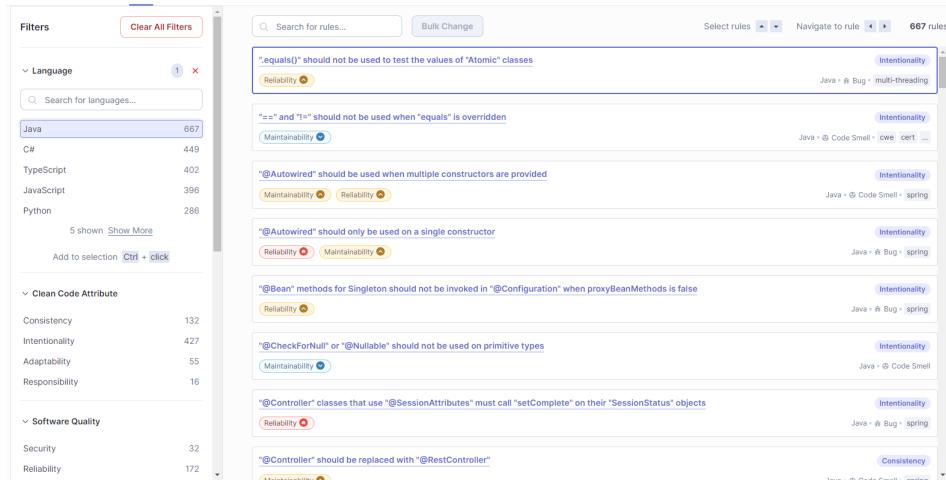
The screenshot shows a Java code editor with a specific line of code highlighted by a red squiggly underline. A tooltip indicates that the code is "Partially covered code". The code is part of a method that handles order requests.

```

@Override
public Optional<OrdineDTO> getOrder(String ingredientePrincipale) throws JsonProcessingException {
    Optional<CodaPostazioneEntity> codaPostazioneEntity = Optional.ofNullable(postazioni.get(ingredientePrincipale
.toUpperCase()));
    if(codaPostazioneEntity.isPresent()) {
        Optional<OrdineEntity> ordineEntity = codaPostazioneEntity.get().element();
        // TODO: settare stato di ordine su: in preparazione
        if (ordineEntity.isPresent()) {
            OrdineDTO ordineDTO = ordineMapper.mapTo(ordineEntity.get());
            NotificaPrepOrdineDTO notifica = NotificaPrepOrdineDTO.builder()
                .id(ordineDTO.getId())
                .idComanda(ordineDTO.getIdComanda())
                .stato(2)
                .build();
            messagePort.send(notifica);
            return Optional.ofNullable(ordineDTO);
        }
    }
    return Optional.empty();
}

```

Figura 2.61: esempio caso uncoverage

**Figura 2.62:** java rules overview sonarqube

Iterazione 2

3.1 Introduzione

Nella Iterazione 2 ci si è focalizzati a studiare l'algoritmo più adatto per la nostra applicazione, partendo dall'organizzare le entità interessate, passando poi per scrivere lo pseudo-codice ed infine ad analizzare le strutture dati necessarie.

Briefing Nell'ambito di questa applicazione si considera che ogni piatto sia composto da un ingrediente principale e da più ingredienti secondari. Ogni piatto ordinato viene chiamato ordine, quindi un ordine comprende un singolo piatto, mentre la comanda contiene tutti gli ordini di un singolo cliente. Nel corso di un brainstorming, si è maturata l'idea di organizzare la cucina in postazioni, ognuna focalizzata su un ingrediente principale: ogni postazione si occuperà quindi di preparare e completare piatti accomunati dallo stesso ingrediente principale.

3.2 Organizzazione

Di seguito viene illustrata l'organizzazione delle entità coinvolte nella gestione dell'algoritmo:

3.2.1 Ordine

Ogni ordine contiene un singolo piatto del menù, viene classificato per ingrediente principale univoco (es. riso, pasta, pesce, . . .), ogni ordine presenta poi più parametri, questi contribuiscono a calcolare la priorità ad esso associata. Parametri ordine:

- Ingrediente principale;
- Tempo di preparazione;
- Numero ordine effettuato (primo, secondo, . . .);
- Urgenza del cliente;
- Tempo in attesa.

3.2.2 Cucina

La cucina viene organizzata in postazioni di lavoro, ossia delle aree dedicate organizzate per svolgere specifiche attività culinarie adibite alla preparazione di piatti che hanno in comune il medesimo ingrediente principale, nello specifico:

- ogni postazione di lavoro è adibita al massimo a 1 ingrediente principale;

3.2.3 Postazione

Con postazione si intende uno spazio di lavoro attrezzato con gli strumenti necessari per lavorare con un particolare tipo di ingrediente principale. Una singola postazione presenta una struttura dati per gestire gli ordini in coda di preparazione.

- 1 ingrediente principale;
- 1 struttura dati (coda);
- capacità massima;
- grado di riempimento.

3.3 Funzione di priorità

La funzione di priorità è una funzione matematica che assegna un valore numerico decimale di priorità nell'intervallo tra 0 e 1 basandosi sui parametri specifici di ogni ordine. Il primo passo consiste nel processo di normalizzazione dei parametri, il quale permette di standardizzare i valori in modo che siano compresi tra 0 e 1, in maniera tale da mettere i diversi parametri su una scala comune e uniforme

3.3.1 Parametri

x1 ingrediente principale: Indica il valore di priorità che presenta l'ingrediente predominante dell'ordine, questo valore è influenzato direttamente dallo stato della postazione di lavoro associata in cucina attraverso il grado di riempimento della coda di postazione.

- se la cucina è satura ridurre il valore di x1 (min 0)
- se la cucina è scarica aumentare il valore di x1(max 1)

x2 tempo di preparazione: Rappresenta la durata stimata necessaria per preparare un determinato ordine.

- normalizzazione:

$$tp_{norm} = \frac{tp - tp_{min}}{tp_{max} - tp_{min}}$$

- **tp:** tempo di preparazione
- **tp_{max}:** tempo di preparazione massimo;
- **tp_{min}:** tempo di preparazione minimo;
- Considerare $x2 = tp_{norm}$ per prioritizzare ordini più lunghi, oppure $x2 = 1 - tp_{norm}$ per prioritizzare ordini più brevi.

x3 urgenza del cliente: Consente ai clienti di specificare la tempestività con cui desiderano ricevere il proprio ordine, in particolare i clienti possono chiedere espressamente di avere urgenza, al contrario possono non dire nulla e tenere un valore di urgenza di default.

- 1 se il cliente ha espresso urgenza;
- 0 se il cliente non ha espresso alcuna urgenza - *parametro di default*.

x4 numero ordine effettuato: Specifica il numero dell'ordine del cliente in ordine temporale, in particolare indica la posizione relativa di un ordine all'interno della sequenza di ordini effettuati da egli stesso.

- il primo ordine effettuato ha priorità maggiore, mentre i successivi hanno priorità decrescente data dalla normalizzazione, fino ad un numero di ordine soglia oltre il quale la priorità sarà minima;

- funzione:

$$f(\text{noe}) = \begin{cases} 1 - \frac{\text{noe}-1}{\text{sogliaMax}-1} & \text{se } \text{noe} \leq \text{sogliaMax} \\ 0 & \text{se } \text{noe} > \text{sogliaMax} \end{cases}$$

- **noe** : numero ordine effettuato;
- **sogliaMax** : massimo numero ordini effettuabili;
- **ESEMPIO:** sia dato $\text{sogliaMax} = 5$, il 1° avrà priorità 1 (massima), dal 2° al 5° avremo priorità normata, dopo 5 ordine il parametro x4 avrà priorità 0 (minima).

x5 tempo in attesa: Rappresenta il periodo di tempo trascorso da quando un ordine è stato effettuato fino al momento in cui viene elaborato.

- funzione:

$$f(\text{tempo di attesa}) = \begin{cases} \frac{\text{tempo di attesa}}{\text{tempo max in attesa}} & \text{se tempo di attesa} \leq \text{tempo max di attesa} \\ 1 & \text{se tempo di attesa} > \text{tempo max di attesa} \end{cases}$$

- **tempo di attesa:** è dato dalla differenza fra istante di tempo attuale e istante di tempo della richiesta ordine;
- **tempo max di attesa:** massimo tempo di attesa ordini oltre il quale la priorità è massima;
- considerare un valore massimo di tempo in attesa consentito, in prossimità del quale si ha la priorità più elevata.

3.3.2 Pesi

I pesi sono utilizzati per attribuire un grado di importanza relativo a ciascun parametro all'interno della funzione di priorità. Questi pesi indicano quanto ciascun parametro dovrebbe influenzare il calcolo complessivo della priorità di un determinato elemento. Si elencano di seguito i pesi per ciascun parametro definito poc' anzi:

- p1: peso ingrediente principale;
- p2: peso tempo di preparazione;
- p3: peso urgenza cliente;
- p4: peso numero ordine;
- p5: peso tempo in attesa.

Viene quindi fatto un ragionamento sull'importanza da attribuire a ogni parametro tramite l'incidenza assegnata al singolo peso. Si dividono quindi i pesi in tre categorie.

Maggiore incidenza I pesi che devono essere più incidenti sono:

- p1 peso ingrediente principale: per evitare di sovraccaricare una postazione rispetto alle altre o per non avere postazioni vuote;
- p5 peso tempo in attesa: un ordine non può restare in attesa troppo a lungo.

Incidenza media Il peso con incidenza media è:

- p3 peso urgenza del cliente: è meno importante dei vincoli di sovraccarico e attesa, ma deve essere comunque una scelta significativa.

Bassa incidenza I pesi con bassa incidenza sulla priorità sono:

- p2 peso tempo di preparazione: in confronto ad altri parametri con pesi più elevati, questo è considerato meno critico;
- p4 peso numero ordine effettuato: ha un impatto di poco conto sulla priorità dell'ordine.

Valore dei pesi

I valori dei pesi vengono quindi definiti inizialmente:

- $p1 = 0.25$;
- $p2 = 0.15$;
- $p3 = 0.20$;
- $p4 = 0.15$;
- $p5 = 0.25$.

Prospettive Future Questi valori possono essere regolati col tempo per aumentare l'efficienza dell'algoritmo, diventa così importante raccogliere dati storici per poterli analizzare e comprendere come i vari parametri influenzano le prestazioni del sistema, oltre a raccogliere feedback dei clienti, sulla base di ciò sarà richiesto un tuning dei pesi più accurato, ad esempio tramite un modello di machine learning. Per questo motivo è richiesta una certa flessibilità in modo da consentire l'aggiornamento dei pesi dei parametri in modo dinamico.

3.3.3 Funzione matematica

L'equazione proposta rappresenta una somma pesata dei parametri, dove ciascun parametro (x_1, x_2, x_3, x_4, x_5) viene moltiplicato per il suo relativo peso (p_1, p_2, p_3, p_4, p_5). I pesi indicano l'importanza relativa dei parametri nel determinare la priorità complessiva di un elemento. La somma pesata dei parametri produce un valore (y) compreso tra 0 e 1, dove 0 indica un valore meno urgente e 1 indica un valore più urgente.

$$y = p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4 + p_5 * x_5$$

3.4 Pseudocodice

Entità: I dati utilizzati da questa parte di algoritmo sono elencati nella seguente per avere una miglior lettura degli pseudocodici.

- OrdinePQ (idOrdine, idComanda, timestamp, stato, urgenzaCliente, valorePriorita, ingredientePrincipale, tpDiPreparazione, numOrdineEffettuato);
- CodaPostazione (ingredientePrincipale, lunghezza, gradoDiRiempimento, tpDiPreparazione).

Di seguito vengono ora riportati gli pseudocodici suddivisi in funzioni per una migliore lettura.

3.4.1 Assegna valore di priorità:

Questo pseudocodice rappresenta la parte principale, esso definisce pesi e assegna parametri portando al calcolo dell'effettivo **ValorePriorita** assegnato ad ogni ordine.

Algorithm 1 Algoritmo che assegna un valore di priorità ad un ordine dato in input in base a pesi fissi e al valore dei parametri calcolato per mezzo di sottofunzioni e assegna o aggiorna questo ordine in una coda a priorità

procedure ASSEGNApriorita(ordinEPQ, priorityQueue) ▷ Calcola il ValorePriorità dell'ordine.

```

 $p1 \leftarrow 0.25$ 
 $p2 \leftarrow 0.15$ 
 $p3 \leftarrow 0.20$ 
 $p4 \leftarrow 0.15$ 
 $p5 \leftarrow 0.25$ 

 $x1 \leftarrow \text{CalcolaIngredientePrincipale}(\text{ordinEPQ.ingredientePrincipale})$ 
 $x2 \leftarrow \text{CalcolaTempoDiPreparazione}(\text{ordinEPQ.tpDiPreparazione})$ 
 $x3 \leftarrow \text{CalcolaUrgenzaDelCliente}(\text{ordinEPQ.urgenzaCliente})$ 
 $x4 \leftarrow \text{CalcolaNumeroOrdineEffettuato}(\text{ordinEPQ.numOrdineEffettuato})$ 
 $x5 \leftarrow \text{CalcolaTempoDiAttesa}(\text{ordinEPQ.timestamp})$ 

 $\text{valorePriorita} \leftarrow (p1 \cdot x1) + (p2 \cdot x2) + (p3 \cdot x3) + (p4 \cdot x4) + (p5 \cdot x5)$ 
 $\text{ordinEPQ.valorePriorita} \leftarrow \text{valorePriorita}$ 

if ordinEPQ is in priorityQueue then
    priorityQueue.update(valorePriorita, ordinEPQ)
else
    priorityQueue.add(valorePriorita, ordinEPQ)
end if
end procedure
```

Costo temporale: Il costo temporale di un algoritmo è una misura del tempo di esecuzione in funzione della dimensione dell'input. In questo algoritmo, il costo temporale dipende principalmente dalle funzioni **CalcolaIngredientePrincipale**, **CalcolaTempoDiPreparazione**, **CalcolaUrgenzaDelCliente**, **CalcolaNumeroOrdineEffettuato**, **CalcolaTempoDiAttesa** e dall'operazione **add** oppure **update** sulla **PriorityQueue**.

Costo spaziale: Il costo spaziale di un algoritmo è una misura dello spazio di memoria utilizzato in funzione della dimensione dell'input. Il costo spaziale di questo algoritmo dipende principalmente dalla dimensione della **PriorityQueue** e dalle variabili utilizzate.

3.4.2 Funzione di aggiornamento:

Essendo il sistema del ristorante basato su parametri costantemente aggiornati (nuovi ordini, intervalli temporali, cambio esigenze) avremo bisogno di verificare periodicamente il valore del **ValorePriorità** assegnato ad ogni ordine.

Anche in questo caso avremo una complessità temporale e spaziale che si basa sul numero di ordini da andare a gestire, all'interno della **PriorityQueue**.

Algorithm 2 Funzione di aggiornamento che permette di richiamare la funzione assegna priorità

```

procedure FUNZIONEAGGIORNAMENTO(priorityQueue)
    for all ordine in priorityQueue do
        assegnaPriorità(ordine, priorityQueue)
    end for
end procedure
```

3.4.3 x1 ingrediente principale:

Il parametro x1 permette di verificare quanto la coda di preparazione dell'ingrediente principale richiesto dal cliente sia piena.

Algorithm 3 Funzione che calcola il parametro x1 riferito all'ingrediente principale

```

function CALCOLAINGREDIENTEPRINCIPALE(ingredientePrincipale)
    x1 ← parametro ingrediente principale
    codaPostazione ← findCodaPostazione(ingredientePrincipale)
    x1 ← codaPostazione.gradoDiRiempimento
    return 1 – x1                                ▷ 0: se coda satura, 1: se coda scarica
end function
```

Costo temporale: si riferisce al tempo di esecuzione dell'algoritmo. in questo algoritmo non abbiamo particolari costi da verificare, l'unica operazione rilevante sarà la ricerca delle coda postazione che però possiamo supporre ad accesso diretto e quindi avrà costo unitario **O(1)**.

Costo spaziale: lo spazio di memoria utilizzato dall'algoritmo valuta le variabili che stiamo memorizzando sono **x1**, **ingredientePrincipale** e **codaPostazione** , quindi il costo spaziale è costante, ovvero **O(1)**.

3.4.4 x2 tempo di preparazione:

Il parametro x2 permette di dare maggior priorità agli ordini che hanno un tempo maggiore minore di preparazione.

Algorithm 4 Funzione che calcola il parametro x2 riferito al tempo di preparazione

```

function CALCOLATEMPODIPIREPARAZIONE(tpDiPreparazione)
    x2 ← Parametro tempo di preparazione
    tpMax ← massimo tempo di preparazione fra gli ordini effettuati
    tpMin ← minimo tempo di preparazione fra gli ordini effettuati
    if tpDiPreparazione ≤ tpMax then
        x2 ←  $\frac{\text{tpDiPreparazione}-\text{tpMin}}{\text{tpMax}-\text{tpMin}}$                                 ▷ Normalizzazione
    else
        x2 ← 1
    end if
    return x2                                ▷ Possibile variazione: return 1 -x2
end function
```

Costo temporale: L'istruzione condizionale dipende solo dall'input **tpDiPreparazione** e dai valori di **tpMax** e **tpMin**, che si assume siano già stati calcolati. Quindi, la complessità temporale è **O(1)**, il che significa che l'algoritmo richiede un tempo costante per essere eseguito.

Costo spaziale: L'algoritmo non utilizza strutture dati che crescono con la dimensione dell'input, come ad esempio array o liste. Quindi, la complessità spaziale è **O(1)**, il che significa che l'algoritmo utilizza una quantità costante di memoria.

3.4.5 x3 urgenza del cliente:

Il parametro x3 permette, attraverso la scelta del cliente, il grado di urgenza col quale egli vuol essere servito.

Algorithm 5 Funzione che calcola il parametro x3 riferito all'urgenza del cliente

```

function CALCULARGENZADELCLIENTE(urgenzaCliente)
    x3 ← Parametro urgenza cliente
    if urgenzaCliente = false then
        x3 ← 0
    else if urgenzaCliente = true then
        x3 ← 1
    end if
    return x3
end function

```

Costo temporale: La complessità temporale di questo pseudocodice è **O(1)**, poiché esegue un numero costante di operazioni.

Costo spaziale: La complessità spaziale di questo pseudocodice è **O(1)**, poiché utilizza un numero costante di variabili. Inoltre, lo pseudocodice non utilizza strutture dati che crescono con la dimensione dell'input, quindi la complessità spaziale è costante.

3.4.6 x4 numero ordine effettuato:

Il parametro x4 permette di dare maggior priorità a chi ha più ordini in attesa di preparazione. Il valore sogliaMax rappresenta il valore massimo di ordini oltre il quale la priorità minima.

Algorithm 6 Funzione che calcola il parametro x4 riferito al numero ordine effettuat

```

function CALCULANUMEROORDINEEFFETTUATO(numOrdineEffettuato)
    x4 ← parametro numero ordine effettuato (0 minima, 1 massima)
    sogliaMax ← soglia massima ordini
    if numOrdineEffettuato < sogliaMax then
        x4 ←  $\frac{\text{numOrdineEffettuato}-1}{\text{sogliaMax} - 1}$ 
    else
        x4 ← 1
    end if
    return 1 - x4
end function

```

Costo temporale: La complessità temporale di questo pseudocodice è **O(1)**, poiché esegue un numero costante di operazioni.

Costo spaziale: La complessità spaziale di questo pseudocodice è **O(1)**, poiché utilizza un numero costante di variabili. Inoltre, lo pseudocodice non utilizza strutture dati che crescono con la dimensione dell'input, quindi la complessità spaziale è costante. In generale, lo pseudocodice è efficiente e non presenta problemi di complessità temporale o spaziale.

3.4.7 x5 tempo in attesa:

Il parametro x5 permette di verificare quanto tempo è passato dall'ordinazione del cliente, dando maggior priorità a chi ha atteso maggiormente dall'invio dell'ordine.

Algorithm 7 Funzione che calcola il parametro x5 riferito al tempo di attesa del cliente

```

function CALCOLATEMPODIATTESA(timestamp)
    x5 ← Parametro tempo di attesa
    tpMaxDiAttesa ← valore massimo di attesa scelto apriori
    tpAttuale ← t rappresenta l'istante di tempo attuale
    t ← tpAttuale – timestamp                                ▷ Tempo trascorso
    if t = 0 then
        x5 ← 0                                              ▷ Evita divisione per zero
    else if t ≤ tpMaxDiAttesa then
        x5 ←  $\frac{t}{tpMaxDiAttesa}$ 
    else
        x5 ← 1                                              ▷ Troppo tempo atteso, priorità massima
    end if
    return x5
end function
```

Costo temporale: La complessità temporale di questo pseudocodice è **O(1)**, poiché esegue un numero costante di operazioni.

Costo spaziale: La complessità spaziale è **O(1)**, poiché utilizza un numero costante di variabili.

3.4.8 Gestore Code:

Dopo aver assegnato una priorità a ciascun ordine, sarà possibile organizzare gli ordini effettuati utilizzando la nostra **PriorityQueue**. Una volta completata questa procedura, gestiremo l'inserimento all'interno delle M code di preparazione. Questo processo è stato organizzato in modo tale da evitare un eccessivo afflusso di ordini all'interno delle code di preparazione, prevenendo la saturazione, per questo è stato pensato in parametro **C** che potrà essere gestito così da non distribuire contemporaneamente tutti gli ordini.

Algorithm 8 Funzione che inserisce ordini in codaPostazione solo se essa non è troppo piena e se il numero totale di ordini in cucina non supera una determinata soglia

```
procedure INSERIMENTOCODAPOSTAZIONE(ordinePQ)
    if codaPostazione.isFull() = false ∧ numeroOrdiniTotaliInCucina() < C then
        codaPostazione.add(OrdinePQ)
    end if
end procedure
```

Costo temporale: La complessità temporale di questo pseudocodice è **O(1)**.

Costo spaziale: La complessità spaziale è **O(1)**.

3.5 Struttura dati

Per quanto riguarda il flusso di un ordine all'interno del sistema si considera che immediatamente dopo l'ordinazione da parte del cliente viene assegnata una priorità per tale ordine, gli ordini vengono così raccolti nella struttura dati principale con l'etichetta della priorità. Successivamente, se la cucina lo richiede, l'ordine con la priorità più elevata viene spostato nella coda di preparazione della rispettiva postazione di lavoro.

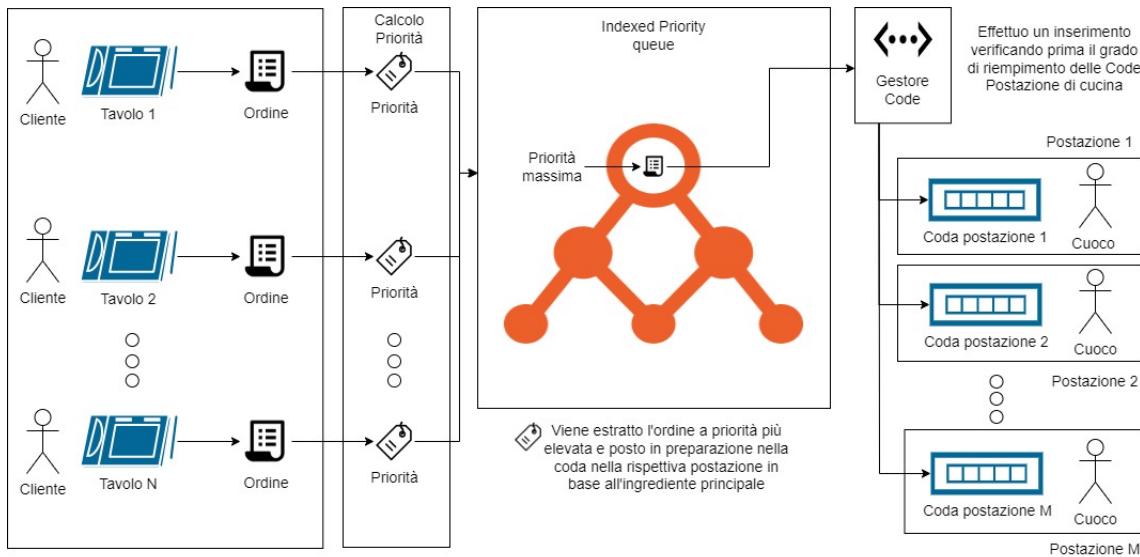


Figura 3.1: Strutture dati dell'algoritmo

Si rendono quindi necessarie due tipi di strutture dati:

- Struttura dati principale: Indexed priority queue[13];
- Struttura dati delle postazioni: Coda (queue).

3.5.1 Indexed priority queue

Struttura dati che estende il concetto di coda con priorità aggiungendo la possibilità di accedere in tempo costante agli elementi presenti in coda per compiere operazioni quali la modifica dei parametri, l'aggiornamento della priorità o la rimozione dell'ordine (che altrimenti presenterebbe costo lineare). Viene implementata per mezzo di una combinazione di una coda con priorità (max heap) e un dizionario (hashtable) che tiene traccia della posizione di ogni elemento all'interno della coda.

Analisi complessità: La complessità temporale è correlata a quella di un heap binario, potenziato dall'accesso diretto agli elementi tramite dizionario, di conseguenza:

- creazione: $O(n)$;
- inserimento e rimozione: $O(\log n)$;
- modifica priorità: $O(\log n)$;
- accedere a un elemento: $O(1)$.

Requisiti funzionali:

- Gestione degli ordini con priorità: funzionalità chiave della struttura dati, gli ordini ricevono una priorità prima di entrare nella coda a priorità indicizzata;
- Fornire l'ordine con priorità più elevata: la struttura dati deve essere in grado di fornire alla cucina l'ordine con la priorità più alta quando richiesto;
- Accesso, modifica e rimozione degli ordini: la coda a priorità deve poter fornire la possibilità di implementare la funzionalità che consente ai clienti di accedere, modificare o rimuovere il proprio ordine (nelle prossime iterazioni);
- Flessibilità nella modifica delle priorità: la struttura deve garantire una certa flessibilità alla modifica delle priorità degli ordini, poiché le priorità possono cambiare per conto dei clienti, della cucina e a intervalli regolari di tempo.

Requisiti non funzionali:

- Tempo di risposta rapido: l'ordine con priorità più elevata deve essere fornito in tempo rapido alla cucina senza ritardi;
- Tempo di accesso, modifica e rimozione ragionevole: il cliente deve poter effettuare operazioni senza complicazioni in tempi ragionevoli, mantenendo un'esperienza di utilizzo piacevole;
- Scalabilità: La struttura dati deve essere in grado di gestire un grande volume di ordini, adattandosi alle variazioni nella domanda senza compromettere le prestazioni;
- Flessibilità alle modifiche: requisito non funzionale relativo alla flessibilità e alla manutenibilità del sistema.

3.5.2 Coda (queue)

Struttura dati lineare che segue il principio "First In, First Out" (FIFO), ossia il principio per il quale il primo elemento che entra nella coda è poi il primo che esce.

Analisi complessità:

- inserimento in coda: $O(1)$;
- rimozione della testa: $O(1)$;
- verifica stato: $O(1)$ se vuota, $O(n)$ altrimenti.

Requisiti funzionali:

- Funzionamento FIFO: La coda deve garantire il corretto funzionamento FIFO (First In, First Out), indipendentemente dalle priorità degli ordini;
- Soglia di attivazione: Il sistema deve permettere di configurare una soglia di valore minimo di attivazione per la coda, al di sotto della quale la postazione non viene attivata;
- Soglia critica di intasamento: Il sistema deve permettere di configurare una soglia di valore critico, oltre la quale la postazione diventa intasata e richiede operazioni per ridurre il carico.

Requisiti non funzionali:

- Lunghezza finita della coda: Il sistema deve gestire una coda con una lunghezza finita, limitata dalla capacità della postazione di lavoro;
- Tempo massimo di attesa in coda: Il sistema deve garantire che gli ordini non rimangano in coda di preparazione per troppo tempo prima di essere elaborati;
- Attivazione anticipata della postazione: In casi di eccessivo ritardo nella preparazione degli ordini, il sistema può attivare una postazione di lavoro anche se è al di sotto della soglia minima di attivazione.

3.5.3 Diagramma di flusso

Per comprendere meglio il processo dell'algoritmo viene mostrato il diagramma di flusso in Figura 3.2, nel quale viene mostrato il flusso di un ordine dal momento in cui viene effettuato dal cliente a quando viene assegnato alla postazione di lavoro in cucina.

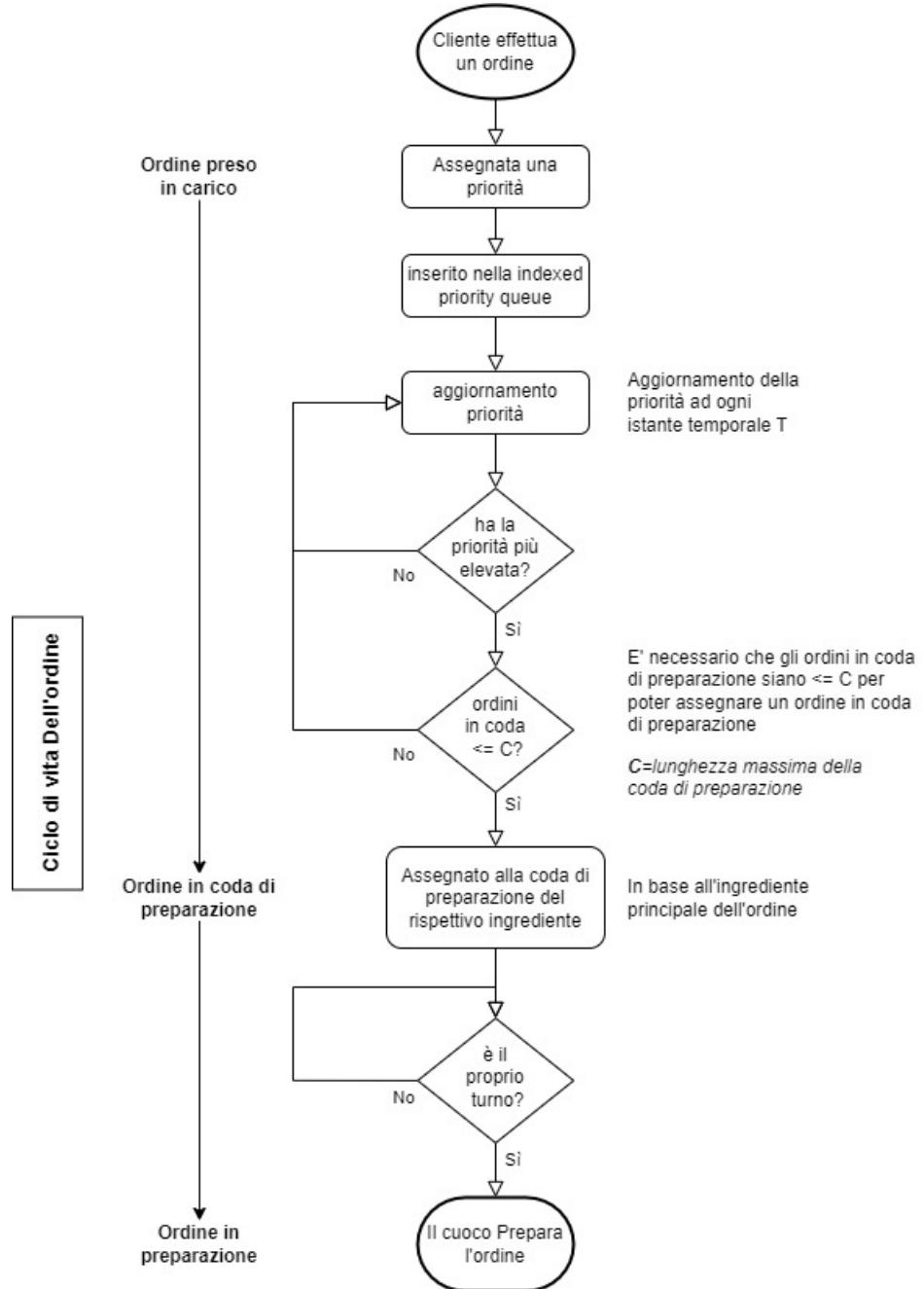


Figura 3.2: Diagramma di flusso

Iterazione 3

4.1 Introduzione

Nell’ambito della terza iterazione del progetto, è stata fornita una prima implementazione dell’algoritmo descritto in pseudocodice durante la seconda iterazione. In questa fase, abbiamo selezionato attentamente i linguaggi di programmazione e le strutture dati più adatte per garantire efficienza e robustezza al sistema. Inoltre, abbiamo sviluppato e integrato il gateway di rete, un componente cruciale per la comunicazione tra i vari moduli del sistema.

4.2 Gateway

Per unificare le chiamate da e verso l'esterno su un'unica porta, si è deciso di implementare un gateway. Questo può comportare significativi vantaggi anche per sviluppi futuri:

- Può agire come un punto di ingresso sicuro per l'applicazione, filtrando e autenticando le richieste provenienti dall'esterno;
- Permette meccanismi di caching;

Permette inoltre di astrarre la composizione interna del servizio dal punto di vista del client, il quale vedrà il sistema come un'unica entità con un unico punto d'accesso.

4.2.1 Component Diagram

A seguito dell'aggiunta di questo microservizio, viene quindi aggiornato il diagramma dei componenti.

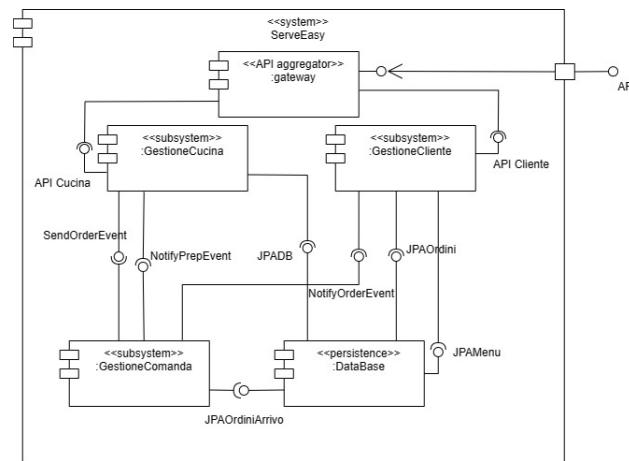
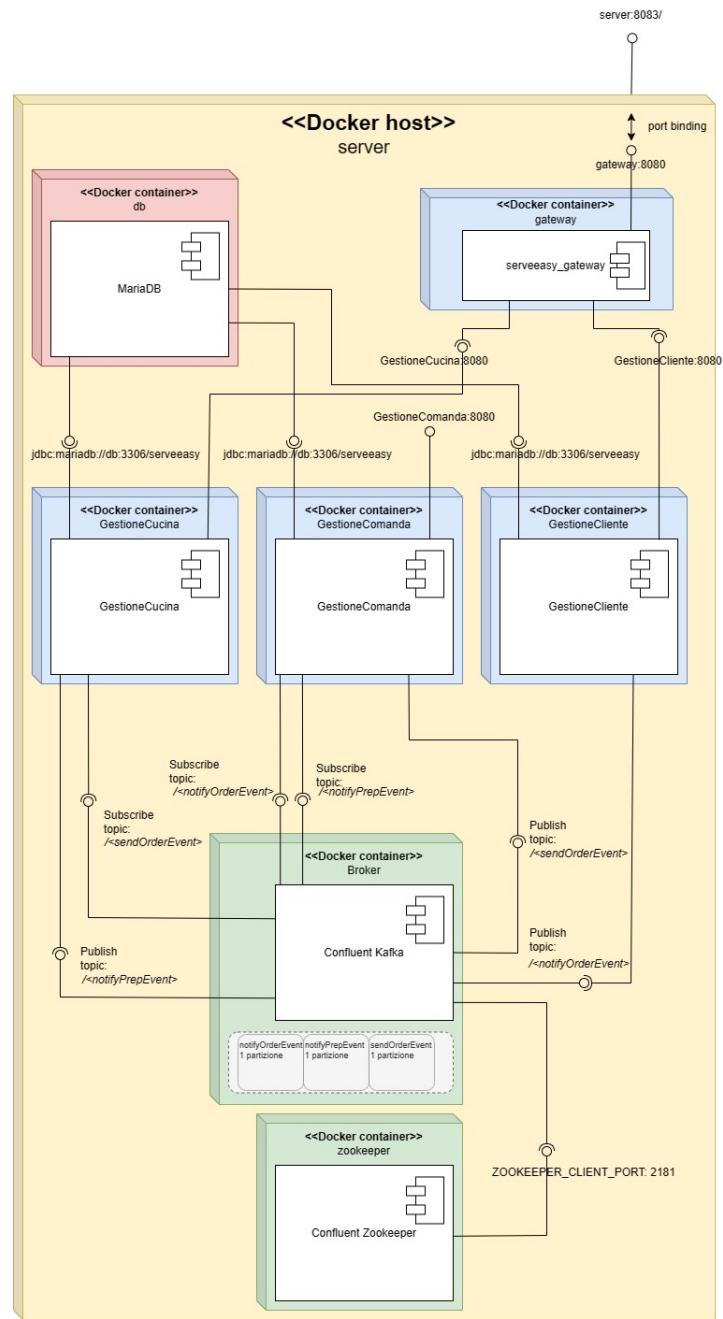


Figura 4.1: Component Diagram con Gateway

4.2.2 Deployment Diagram

Viene aggiunto un nodo alla rete di container, il quale è l'unico ad esporre un'interfaccia all'esterno.

**Figura 4.2:** Deployment Diagram con gateway

4.3 Algoritmo

Al fine di implementare l'algoritmo, si è inizialmente pensato di svilupparlo in un ambiente separato dai microservizi così da avere uno sviluppo più semplificato e di più facile da testare. Si è quindi utilizzato un ambiente di sviluppo Java con l'ide IntelliJ.

4.3.1 brainstorming

La fase iniziale dell'implementazione è stata affrontata su draw.io dove facendo principalmente riferimento al flowchart ed alla struttura data all'algoritmo sono stati individuati i componenti che si adattassero meglio al nostro pseudocodice. L'Output ottenuto da questo brainstorming è infine rappresentato nella seguente immagine:

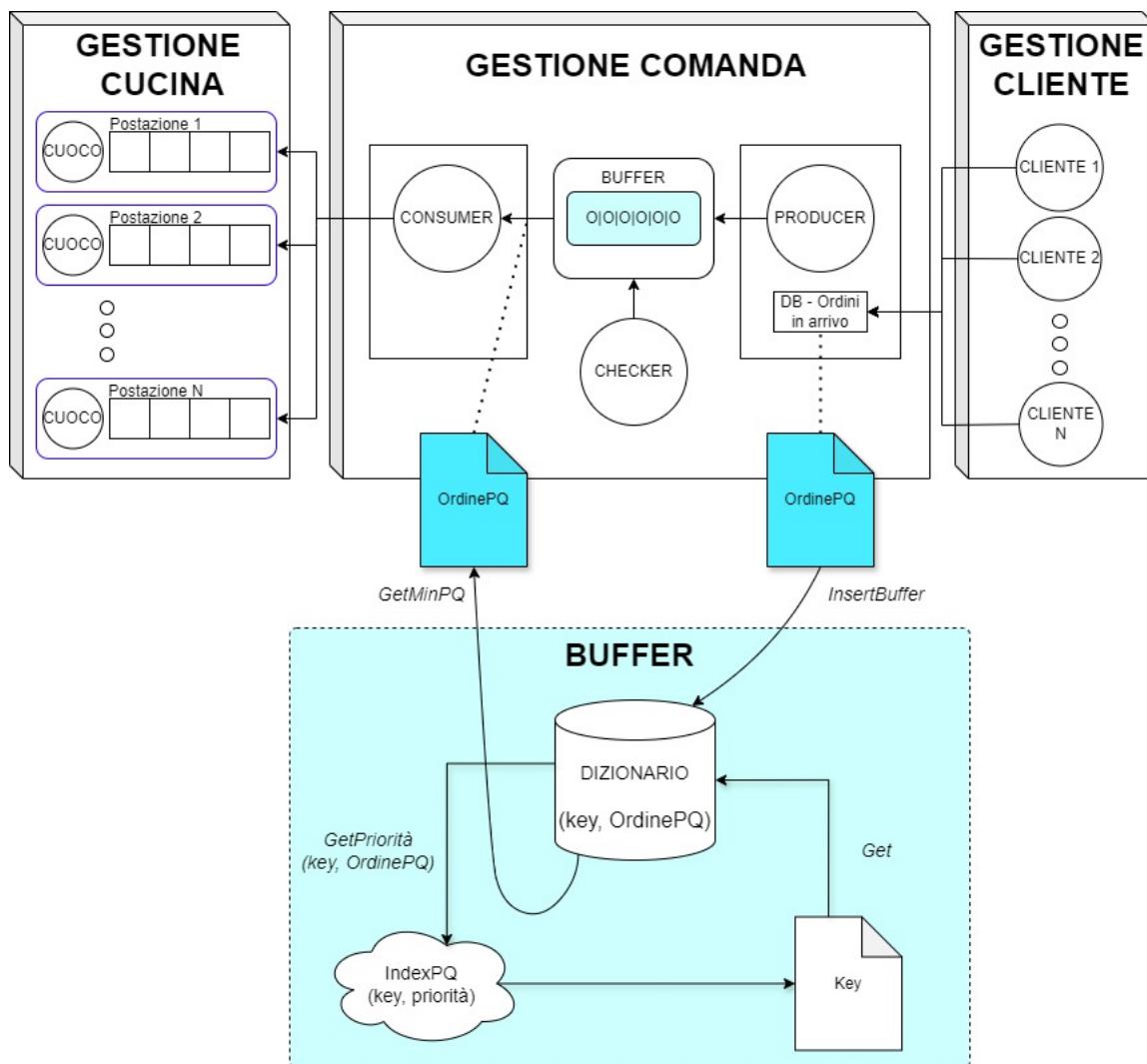


Figura 4.3: logico-struttura dell'algoritmo da implementare

4.3.2 Adattamento struttura dati

La scelta implementativa presa ha portato alla ricerca di una struttura equivalente alla **IndexPriorityQueue**.

Dopo una attenta ricerca la struttura scelta è stata la **IndexMinPQ** [28], basata su una PriorityQueue indicizzata che permette di costuire un heap dove la minima priorità è a radice. Successivamente, è stata sviluppata una classe **Dizionario** contenente un dizionario (hashMap) ed un array (ArrayList) di supporto per indicizzare univocamente le chiavi della IndexPriorityQueue ed inoltre mantene riutilizzabili gli stessi indici.

Vi era infatti un problema implementativo per il quale ad ogni elemento aggiunto all'heap avrebbe avuto un Id identificativo incrementale causando una esponenzialità di quest'ultimo non desiderata, attraverso il dizionario di supporto è stato quindi possibile riutilizzare degli indici univoci all'interno dell'heap ordinato della indexMinPQ ed avere un accesso diretto all'elemento da estrarre.

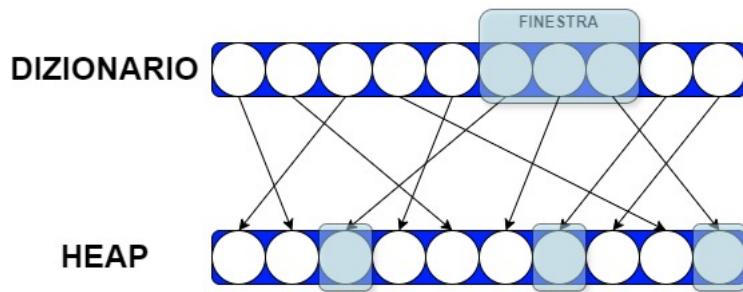


Figura 4.4: associazione dizionario ed heap

4.3.3 Implementazione

L'effettiva implementazione si è basata su un approccio di sviluppo bottom-up delle varie componenti.

Inizialmente è stata sviluppata la struttura dati attraverso i seguenti passaggi:

- verificato il corretto funzionamento della IndexMinPQ ed adattato il funzionamento al nostro caso di utilizzo;
- implementato il dizionario composto da hashMap e ArrayList associato all'heap della struttura dati;
- verificato il corretto funzionamento dell'associazione imposta fra le strutture.

Per lo Sviluppo della gestione dei thread:

- Si è creato un Buffer semaphore gestito inizialmente da tre thread:
 - **Producer:** verifica la disponibilità di posti all'interno del buffer ed inserisce gli ordini con una priorità iniziale già assegnata in esso;
 - **Consumer:** estraе l'elemento con priorità più elevata dalla struttura dati per poi inserirlo all'interno delle code di postazione attraverso una FIFO;
 - **Checker:** verifica attraverso una finestra limitata l'aggiornamento di priorità dei vari ordini presenti nel buffer.
- adattato il buffer alle strutture dati utilizzate;
- verificato il corretto utilizzo del buffer con le strutture dati con alcuni oggetti test.

Infine sono state sviluppate alcune entità e code utili per simulare il percorso degli ordini effettuati dal cliente fino alla coda di postazione designata per ingrediente principale. Fra esse abbiamo:

- **Cliente:** ogni cliente rappresenta un thread che genera un ordine;
- **Cuoco:** ogni cuoco rappresenta un thread che riceve degli ordini;
- **Coda postazione:** coda singola che permette di ricevere ordini e processarli come una coda fifo;
- **OrdinePQ:** classe che rappresenta l'ordine e tutti i suoi attributi;
- **Gestione priorità:** classe che permette di gestire i pesi **p** e **x** che portano al calcolo del **valore priorità** di ogni ordine.

4.3.4 Simulazione

Viene di seguito mostrato un esempio di funzionamento per via dei messaggi di log sulla console

Setup:

Per questa simulazione si sono scelti i seguenti valori:

- Producer: sleep ogni 500 millisecondi;
- Consumer: sleep ogni 2 secondi;
- Checker: sleep ogni 3 secondi;
- Cliente: 4 thread con sleep random tra 10 e 60 secondi;
- Cuoco: 4 thread con sleep in base all'ordine variabile tra 10 e 60 secondi;
- Buffer size: 10 posti;
- Buffer cucina: 10 posti (numero di ordini totali in cucina);
- Capacità singola coda postazione: 5 posti;
- 4 Code di postazione: PASTA, RISO, CARNE, PESCE.

Cliente e Producer:

```
Cliente: aggiungo un ordine nel sistema: OrdinePQ{ id=6094, PASTA }
uuid:cl18-secs:28855519531900-id:6094-type:PASTA-thread:clienteC-state:end:::
Producer: rilevato ordine in arrivo.
uuid:p38-secs:28855642022900-id:0-type:0-thread:producer-state:begin:::
Producer: calcolo della priorità per: OrdinePQ{ id=6094, PASTA }
Gestione Priorita': x1 = 0.6
Gestione Priorita': x2 = 0.92
Gestione Priorita': x3 = 1
Gestione Priorita': x4 = 1.0
Gestione Priorita': x5 = 6.83333333333333E-4
Gestione Priorita': y = 0.6381708477576574
Gestione Priorita iniziale': y' = 0.8506666858990988
uuid:p38-secs:28855642500100-id:6094-type:PASTA-thread:producer-state:end:::
Producer: inserimento avvenuto con successo: OrdinePQ{ id=6094, PASTA, valorePriorità=0,638 }
```

Figura 4.5: Funzionamento Thread Cliente e Producer

In Figura4.5 è mostrata l'interazione tra il thread Cliente e il Producer, si può notare come il Cliente effettua un'ordinazione aggiungendo un ordine nel sistema, viene mostrato l'id

dell'ordine e il tipo di ingrediente principale, successivamente il Producer si attiva e rileva un ordine all'interno della sua coda, prende questo ordine, gli calcola una priorità passando per la classe Gestione Priorità, e lo inserisce nel buffer. Vengono inoltre mostrati sulla console i passi che si compiono per calcolare la priorità. La priorità iniziale y' differisce dalla y poiché è calcolata senza il contributo del parametro x_5 visto che nullo all'entrata nel buffer (quindi viene calcolata facendo la somma pesata solo dei primi 4 parametri).

Checker:

```
uuid:ch30-secs:28857218086200-id:0-type:0-thread:checker-state:begin:::  
***** Checker controlla il buffer *****  
Gestione Priorita': x1 = 0.6  
Gestione Priorita': x2 = 0.92  
Gestione Priorita': x3 = 1  
Gestione Priorita': x4 = 1.0  
Gestione Priorita': x5 = 0.009461111111111111  
Gestione Priorita': y = 0.6403652922021018  
Checker: aggiorno la priorità di: OrdinePQ{ id=6094, PASTA, valorePriorità=0,640 }  
Checker: priorità aggiornata: OrdinePQ{ id=6094, PASTA, valorePriorità=0,640 }  
Gestione Priorita': x1 = 0.6  
Gestione Priorita': x2 = 0.4  
Gestione Priorita': x3 = 0  
Gestione Priorita': x4 = 0.5  
Gestione Priorita': x5 = 0.05057777777777778  
Gestione Priorita': y = 0.2976444498088625  
Checker: aggiorno la priorità di: OrdinePQ{ id=7979, PASTA, valorePriorità=0,298 }  
Checker: priorità aggiornata: OrdinePQ{ id=7979, PASTA, valorePriorità=0,298 }  
Gestione Priorita': x1 = 0.8  
Gestione Priorita': x2 = 0.7  
Gestione Priorita': x3 = 0  
Gestione Priorita': x4 = 1.0  
Gestione Priorita': x5 = 0.16209444444444446  
Gestione Priorita': y = 0.49552362124390076  
Checker: aggiorno la priorità di: OrdinePQ{ id=7337, RISO, valorePriorità=0,496 }  
Checker: priorità aggiornata: OrdinePQ{ id=7337, RISO, valorePriorità=0,496 }  
***** Checker rilascia il buffer *****
```

Figura 4.6: Funzionamento Thread Checker

In Figura 4.6 è mostrato il comportamento del Checker, il quale si attiva, controlla in maniera mutualmente esclusiva il buffer e seleziona tramite una finestra mobile gli ordini da controllare ed aggiornare la priorità. Ogni aggiornamento di priorità è seguito da un aggiornamento del buffer im modo da riportare la modifica appena apportata all'ordine.

Consumer e Producer:

```

Consumer: chiedo l'ordine a priorità più alta dal buffer...
uuid:c21-secs:30906932283600-id:0-type:0-thread:consumer-state:begin:::
GestioneCode: Capacità massima di ordini in cucina raggiunta: 10
GestioneCode: stampa di tutte le code:
RISO, entities.CodaPostazione{ingredientePrincipale=RISO, numeroOrdiniPresenti=1, gradoRiempimento=0.2, capacita=5,
queue=[OrdinePQ{ id=3421, RISO, valorePriorità=0,673 }]}
CARNE, entities.CodaPostazione{ingredientePrincipale=CARNE, numeroOrdiniPresenti=2, gradoRiempimento=0.4, capacita=5,
queue=[OrdinePQ{ id=6853, CARNE, valorePriorità=0,277 }, OrdinePQ{ id=5473, CARNE, valorePriorità=0,251 }]}
PESCE, entities.CodaPostazione{ingredientePrincipale=PESCE, numeroOrdiniPresenti=3, gradoRiempimento=0.6, capacita=5,
queue=[OrdinePQ{ id=5312, PESCE, valorePriorità=0,410 }, OrdinePQ{ id=9157, PESCE, valorePriorità=0,523 }, OrdinePQ{ id=1799,
PESCE, valorePriorità=0,278 }]}
PASTA, entities.CodaPostazione{ingredientePrincipale=PASTA, numeroOrdiniPresenti=4, gradoRiempimento=0.8, capacita=5,
queue=[OrdinePQ{ id=8290, PASTA, valorePriorità=0,404 }, OrdinePQ{ id=5225, PASTA, valorePriorità=0,632 }, OrdinePQ{ id=3608,
PASTA, valorePriorità=0,377 }, OrdinePQ{ id=8837, PASTA, valorePriorità=0,396 }]}
Consumer: problemi nell'inserimento di: OrdinePQ{ id=8932, RISO, valorePriorità=0,209 }
Consumer: reinserirò nel buffer l'ordine: OrdinePQ{ id=8932, RISO, valorePriorità=0,209 }
uuid:c21-secs:30906934319200-id:8932-type:RISO-thread:consumer-state:end:::
Producer: rilevato ordine da reinserire nel buffer.
uuid:p22-secs:30907434770500-id:0-type:0-thread:producer-state:begin:::
uuid:p22-secs:30907434921100-id:8932-type:RISO-thread:producer-state:end:::
Producer: reinserimento avvenuto con successo: OrdinePQ{ id=8932, RISO, valorePriorità=0,209 }

```

Figura 4.7: Funzionamento Thread Consumer e Producer

In Figura4.7 è mostrato il caso in cui il Consumer estrae l'ordine a priorità più elevata dal buffer ma non può inviarlo in cucina poiché è stato raggiunto il numero massimo di ordini che possono esistere contemporaneamente in cucina, per questo motivo lo passa al Producer per poterlo reinserire all'interno del buffer. Viene stampato lo stato di ogni coda di postazione per una verifica (infatti si nota che la somma degli ordini in cucina è 10 come segnalato). Il Producer riceve quindi l'ordine e lo reinserisce immediatamente nel buffer.

Consumer:

```

Consumer: chiedo l'ordine a priorità più alta dal buffer...
uuid:c4-secs:29335113779700-id:0-type:0-thread:consumer-state:begin:::
GestioneCode: Postazione Aggiornata: entities.CodaPostazione{ingredientePrincipale=CARNE, numeroOrdiniPresenti=2,
gradoRiempimento=0.4, capacita=5, queue=[OrdinePQ{ id=2154, CARNE, valorePriorità=0,660 }, OrdinePQ{ id=2129, CARNE,
valorePriorità=0,564 }]}
GestioneCode: stampa di tutte le code:
RISO, entities.CodaPostazione{ingredientePrincipale=RISO, numeroOrdiniPresenti=1, gradoRiempimento=0.2, capacita=5,
queue=[OrdinePQ{ id=5458, RISO, valorePriorità=0,668 }]}
CARNE, entities.CodaPostazione{ingredientePrincipale=CARNE, numeroOrdiniPresenti=2, gradoRiempimento=0.4, capacita=5,
queue=[OrdinePQ{ id=2154, CARNE, valorePriorità=0,660 }, OrdinePQ{ id=2129, CARNE, valorePriorità=0,564 }]}
PESCE, entities.CodaPostazione{ingredientePrincipale=PESCE, numeroOrdiniPresenti=0, gradoRiempimento=0.0, capacita=5, queue=[]}
PASTA, entities.CodaPostazione{ingredientePrincipale=PASTA, numeroOrdiniPresenti=1, gradoRiempimento=0.2, capacita=5,
queue=[OrdinePQ{ id=1000, PASTA, valorePriorità=0,665 }]}
uuid:c4-secs:29335116407800-id:2129-type:CARNE-thread:consumer-state:end:::
Consumer: estratto: OrdinePQ{ id=2129, CARNE, valorePriorità=0,564 }

```

Figura 4.8: Funzionamento Thread Consumer

In Figura4.8 è mostrato il caso di funzionamento ordinale del Consumer, cioè il caso in cui estrae l'ordine a priorità più elevata dal buffer e lo inserisce nella corrispettiva coda di postazione per ingrediente principale in cucina. Anche in questo caso viene stampato lo stato di ogni coda di postazione per una semplice verifica.

Cuoco:

```
Cuoco PASTA: ordine completato: Optional[OrdinePQ{ id=1000, PASTA, valorePriorità=0,665 }]
uuid:1cu1-secs:29361192979000-id:1000-type:PASTA-thread:cuoco_pasta-state:end:::
Cuoco PASTA: osservo la coda di postazione: Optional[entities.CodaPostazione{ingredientePrincipale=PASTA, numeroOrdiniPresenti=1,
gradoRiempimento=0,2, capacita=5, queue=[OrdinePQ{ id=2030, PASTA, valorePriorità=0,686 }]]]
uuid:1cu2-secs:29416250440200-id:0-type:0-thread:cuoco_pasta-state:begin:::
Cuoco PASTA: preparando l'ordine: Optional[OrdinePQ{ id=2030, PASTA, valorePriorità=0,686 }]
```

Figura 4.9: Funzionamento Thread Cuoco

In Figura4.9 è mostro il lavoro del cuoco, nello specifico quello adibito alla postazione dell'ingrediente principale PASTA, il quale finisce di preparare un ordine, invia di conseguenza una notifica di avvenuta preparazione e si mette in attesa sulla propria coda di postazione, rileva quindi un ordine in coda e comincia la preparazione quel nuovo ordine rilevato.

Dati simulazione:

Per quanto riguarda i dati della simulazione e l'analisi di questi per valutare la performance dell'algoritmo si fa riferimento alla sezione 4.4 nella pagina successiva.

4.4 Analisi dei Dati

Per fare l'operazione di analisi dei dati si è utilizzato Google Colab [15], un ambiente di sviluppo gratuito basato su cloud che offre l'accesso a potenti risorse di calcolo tramite browser, consentendo agli utenti di scrivere, eseguire e condividere codice Python.

Questa operazione di analisi si è concentrata sulle prestazioni dell'algoritmo e sui thread che lavorano all'interno di esso.

4.4.1 Prestazioni algoritmo

La seguente documentazione è presente in formato originale con il codice Python utilizzato, come notebook di Google Colab al seguente link:

- versione 1:

<https://colab.research.google.com/drive/1PZyxKQf85-XFKB7PsUqYRUQSwh0I-cuO?usp=sharing>

- versione 2 (mostrata di seguito):

https://colab.research.google.com/drive/1vJyO7_P-xOQ7RNWgx4I4YvhqxD3cgvAF?usp=sharing

Analisi prestazioni algoritmo

Il seguente notebook viene utilizzato per fare un'analisi dei dati per verificare il requisito non funzionale della performance:

L'algoritmo di priorità impiegato dalla cucina per la selezione degli ordini deve fornire risultati in un tempo utile.

Analisi file di report

Viene caricato e visualizzato il file di report in formato csv che da in output l'algoritmo:

	Numero	ID	tempo_attesa	ingr_principale	t_prep	urgenza	numero_ordine_eff	t_coda	priorita_iniz	priorita_fin
0	1	2154	31.994	CARNE	30	True	1	1	88	66
1	2	1000	33.985	PASTA	30	True	1	3	88	66
2	3	5458	35.998	RISO	30	True	1	5	88	67
3	4	4955	43.045	CARNE	11	True	2	7	69	52
4	5	9577	63.049	CARNE	20	True	4	9	62	43
...
341	342	5906	113.324	PESCE	24	False	1	89	52	49
342	343	2341	148.765	PASTA	56	True	4	83	77	67
343	344	5077	327.236	RISO	30	False	5	213	15	46
344	345	8809	245.492	CARNE	47	True	3	72	51	48
345	346	5628	128.755	PESCE	37	False	2	75	52	49

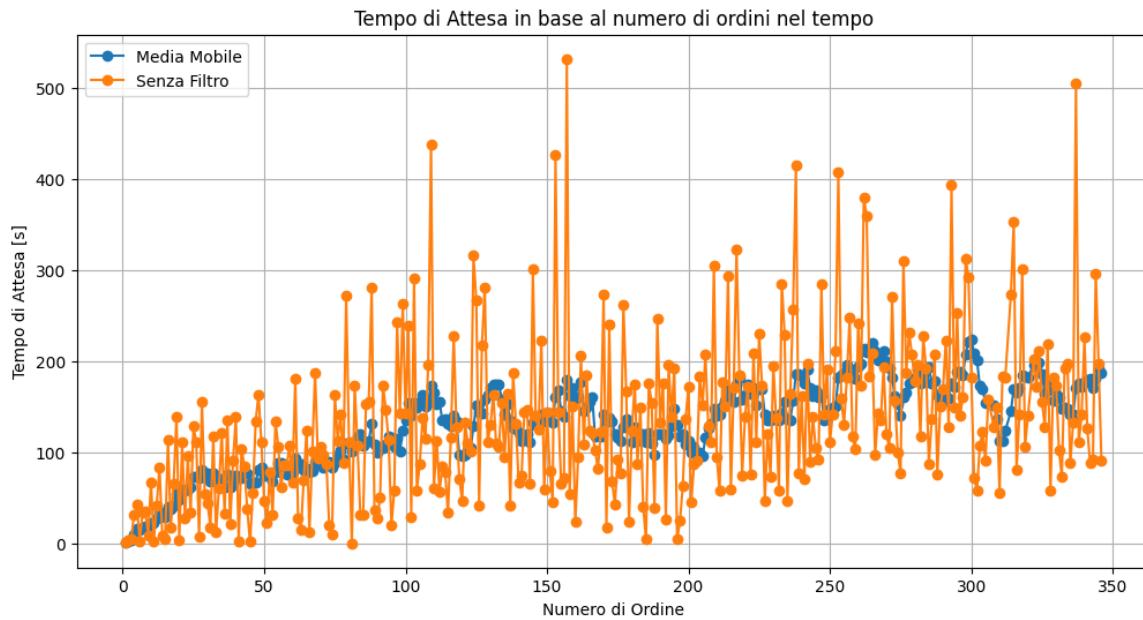
346 rows × 10 columns

Figura 4.10: File csv di report

Analisi Tempo di attesa in base al numero degli ordini

Come primo passo viene analizzato come il tempo di attesa varia con l'avanzare del numero di ordini.

Viene utilizzato un grafico a linea per mostrare (in arancione) il tempo di attesa per numero di ordine e (in blu) i valori filtrati con un filtro a media mobile con finestra = 20 valori

**Figura 4.11:** Analisi Tempo di attesa in base al numero degli ordini

Si può notare come inizialmente la crescita del tempo di attesa sia praticamente costante per poi stabilizzarsi all'interno della banda che va dai 100s ai 200s anche se con un trend crescente. Si può quindi supporre che inizialmente la cucina non riesca a servire

la richiesta dei clienti e quindi accumula ordini in coda, ma successivamente riesce a stabilizzarsi verso un valore medio di tempo di attesa costante.

Analisi distribuzione del tempo di attesa di preparazione degli ordini

In questa sezione viene analizzato il tempo di attesa di preparazione degli ordini, ossia il tempo che passa da quando un ordine viene effettuato a quando viene preso in carico dal cuoco, non viene considerato quindi il tempo di preparazione poiché caratteristico del singolo ordine

Viene ripulito il dataset da possibili outlier usando il z-score e rimuovendo le prime 50 osservazioni perché poco rilevanti (coda scarica).

Viene utilizzato un istogramma con il quale si mostra la frequenza con la quale i vari ordini presentano un certo slot di tempo di attesa, vengono inoltre calcolate e mostrate: media, deviazione standard, mediana, 25° e 75° percentili per una maggiore comprensione.

In un ulteriore istogramma poi si raggruppano queste frequenze in base alla priorità, con valori di Alta, Media e Bassa priorità per capire come l'assegnazione della priorità da parte dell'algoritmo influenza il tempo di attesa di preparazione.

$$\text{Mediatempodiattesa} = 146.326$$

$$\text{Deviazionestandardtempodiattesa} = 88.721$$

$$\text{Medianastandardtempodiattesa} = 133.917$$

$$25\text{percentiletempodiattesa} = 86.847$$

$$75\text{percentiletempodiattesa} = 187.355$$

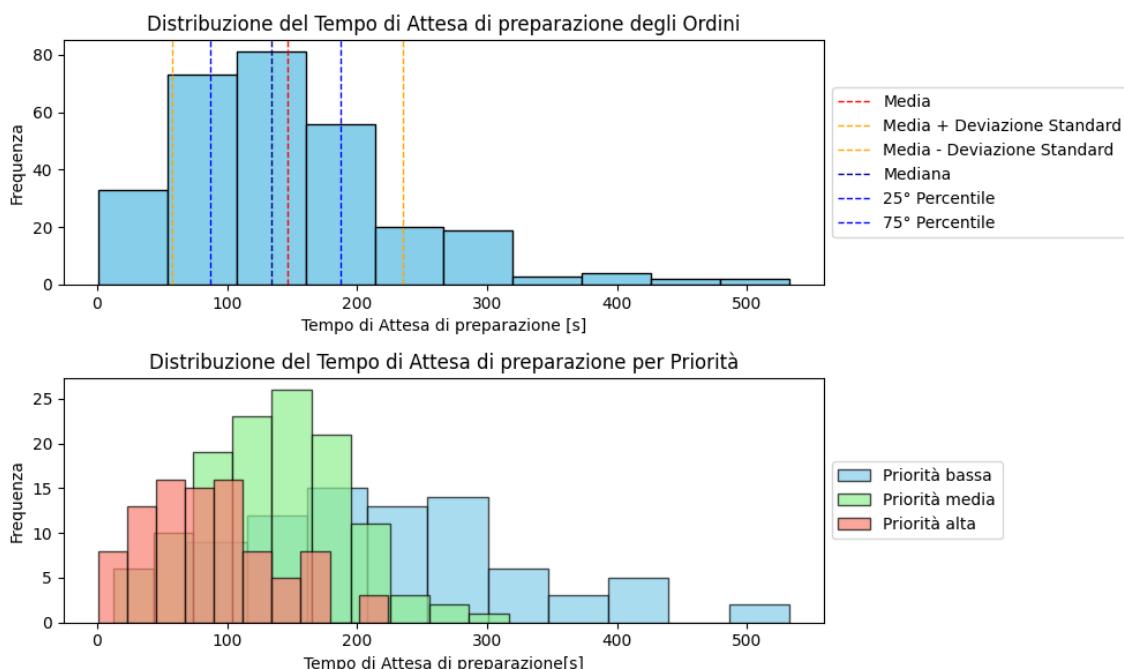


Figura 4.12: Distribuzione del Tempo di Attesa di preparazione degli Ordini

Dal primo istogramma è possibile notare che:

- la media è di poco superiore alla mediana, ciò può fare pensare che la distribuzione sia approssimativamente simmetrica;
- significativa deviazione standard, ciò indica che c'è una variabilità nei tempi di attesa;
- presenza di outlier: dati molto al di fuori dell'intervallo tra il 25° e il 75° percentile, potrebbero essere considerati outlier o dati anomali.

Dal secondo istogramma è invece possibile notare come tendenzialmente le priorità siano rispettate, con ordini che hanno una priorità maggiore che presentano un tempo di attesa di preparazione minore.

Analisi distribuzione del tempo di attesa in coda degli ordini

In questa sezione viene analizzato il tempo di attesa in coda degli ordini, ossia il tempo che passa da quando un ordine viene effettuato a quando viene esce dalla coda per andare in cucina, non viene considerato quindi il tempo di preparazione e il tempo di attesa in cucina.

Viene ripulito il dataset da possibili outlier usando il z-score e rimuovendo le prime 50 osservazioni perché poco rilevanti (coda scarica).

Viene utilizzato un istogramma con il quale si mostra la frequenza con la quale i vari ordini presentano un certo slot di tempo di attesa, vengono inoltre calcolate e mostrate: media, deviazione standard, mediana, 25° e 75° percentili per una maggiore comprensione.

In un ulteriore istogramma poi si raggruppano queste frequenze in base alla priorità, con valori di Alta, Media e Bassa priorità per capire come l'assegnazione della priorità da parte dell'algoritmo influenza il tempo di attesa di preparazione.

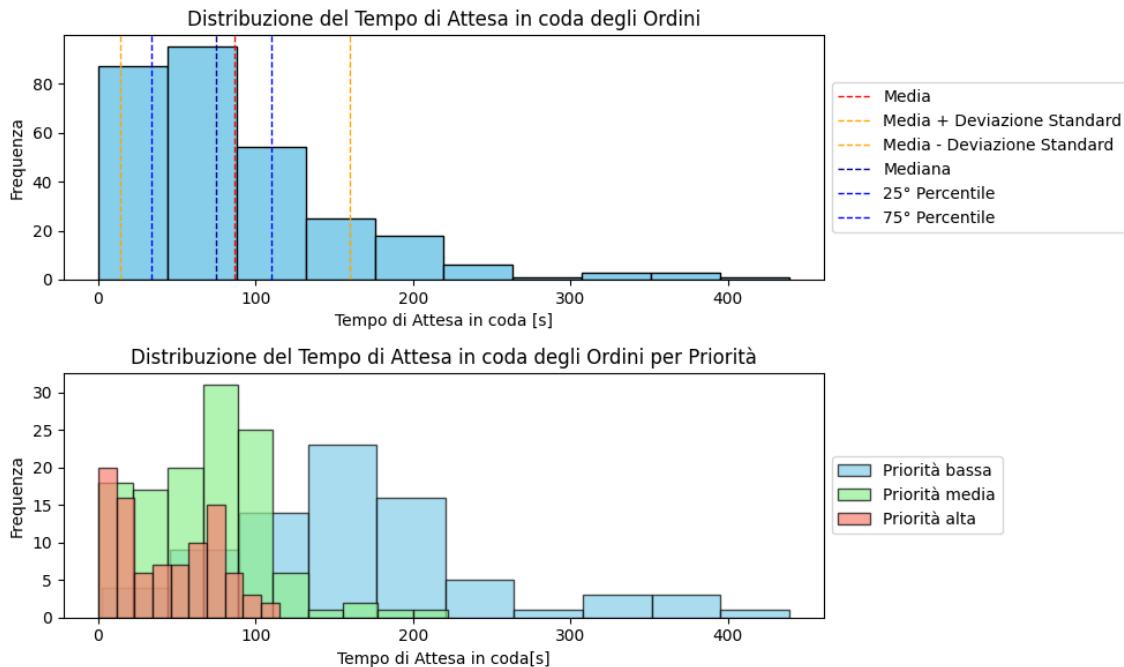
Mediatempodiattesa = 86.819

Deviazionestandardtempodiattesa = 73.051

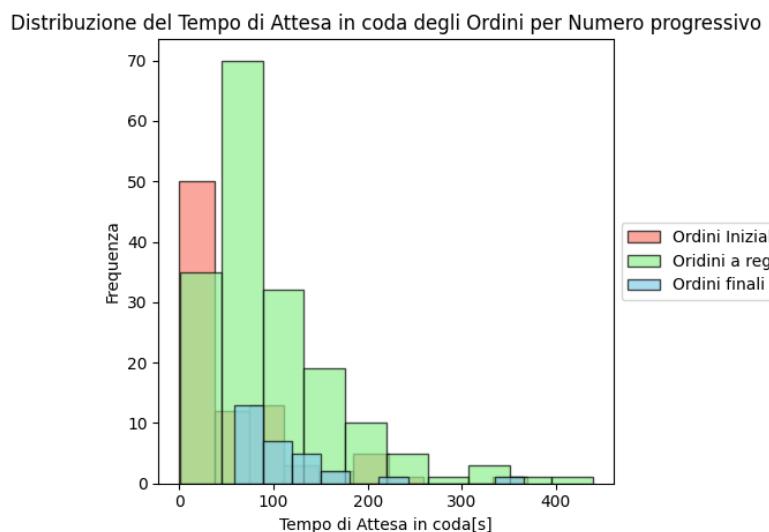
Medianastandardtempodiattesa = 75.000

25percentiletempodiattesa = 34.000

75percentiletempodiattesa = 110.000

**Figura 4.13:** Distribuzione del Tempo di Attesa in coda degli Ordini

Questi grafici sono simili ai grafici calcolati precedentemente per il tempo di attesa di preparazione, per cui si possono tirare le stesse conclusioni, si può inoltre notare che ci sono molti ordini a priorità media e alta che presentano tempi di attesa in coda nulli o poco superiori allo zero, possiamo ipotizzare che l'alta frequenza di ordini con basso tempo di attesa in coda sia data dagli ordini effettuati inizialmente questo perché è possibile che al momento della loro ordinazione trovino il buffer vuoto o comunque poco carico e vengano quindi subito spediti alla cucina.

**Figura 4.14:** Distribuzione del Tempo di Attesa in coda degli Ordini per Numero progressivo

Viene infatti confermata l'ipotesi del passo precedente e cioè che gli ordini che presentano tempi di attesa in coda nulli o poco superiori allo zero siano dati principalmente da ordini effettuati nelle fasi iniziali.

Analisi dei parametri in base al tempo totale in attesa

In questo passo viene analizzato come i 5 parametri su cui si basa la priorità influiscano sul tempo di attesa di un ordine nel sistema, cioè da quando viene effettuato a quando viene completato. I parametri in considerazione sono:

- ingrediente principale, indica la postazione della cucina riservata a quel particolare ordine;
- tempo di preparazione, rappresenta la durata stimata necessaria per preparare un determinato ordine;
- urgenza del cliente, consente ai clienti di specificare la tempestività con cui desiderano ricevere il proprio ordine;
- numero ordine effettuato, specifica il numero dell'ordine del cliente in ordine temporale (posizione relativa di un ordine all'interno della sequenza di ordini effettuati da egli stesso);
- tempo in attesa in coda, rappresenta il periodo di tempo trascorso da quando un ordine è stato effettuato fino al momento in cui viene elaborato.

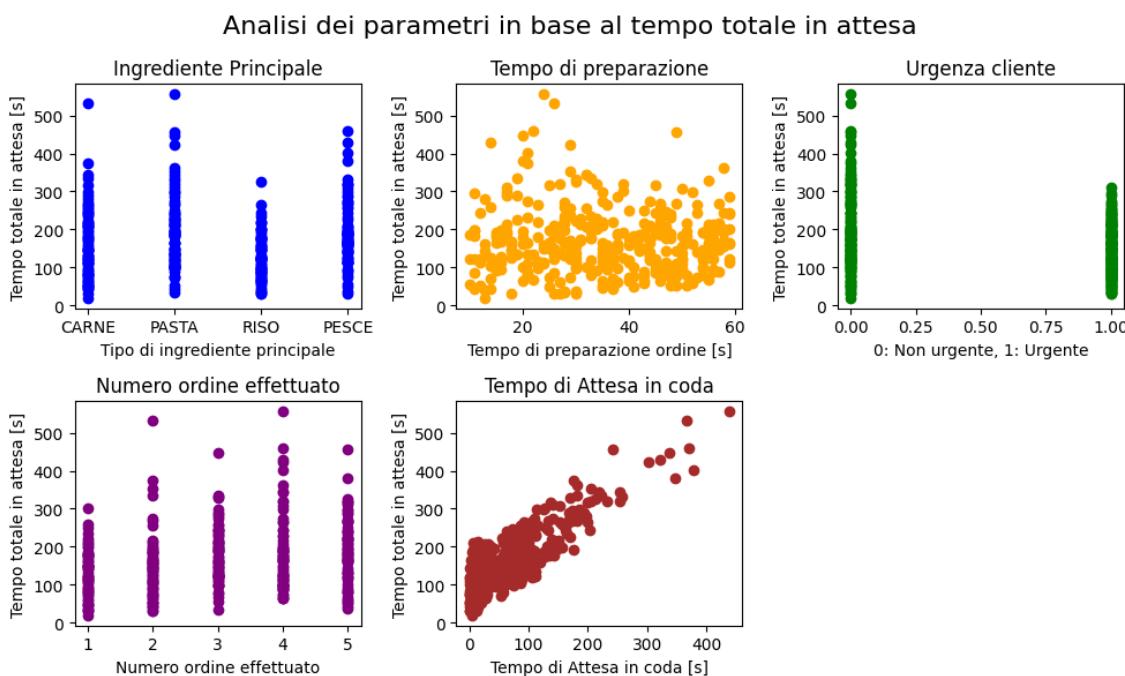


Figura 4.15: Analisi dei parametri in base al tempo totale in attesa

4.4.2 Analisi dei thread

La analisi delle attività dei thread è stata svolta servendosi delle informazioni fornite dai log.

Il lavoro svolto per ottenere dati interpretabili ha seguito una sequenza di passaggi e script:

1. Vengono create delle stampe personalizzate nella classe Printer e, alla voce *Run* della toolbar IntelliJ IDEA, viene imposto il salvataggio dei log nel file *logs.txt*;
2. Ottenuto *logs.txt*, questo viene dato input allo script Python3 *log_to_dict.py*: questo filtra le sole righe del log interessate, distinte dalla substring ":", e, per ogni riga, estrae i dati e organizza la tupla in un dizionario Python. Si ottiene quindi una lista di dizionari che viene salvata su un file *input.txt*, creato a scopo di log, e *input.csv*.
3. Il file *input.csv* viene dato in input al notebook Google Colab *Grant.ipynb* che esegue elaborazioni con modulo Pandas e visualizzazioni con modulo Plotly.

Successivamente è stata concepita una classificazione, sulla base del tempo, per distinguere diverse categorie di dati. Al fine di fornire quindi diverse prospettive sulle simulazioni, i dati sono stati resi disponibili sulla repository GitHub per poter essere resi disponibili automaticamente, con l'ausilio di `git clone`, in una cartella dell'area di lavoro Notebook [Google Colab](#).

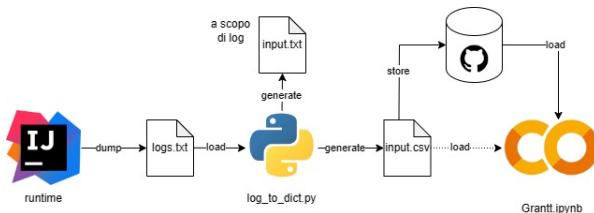
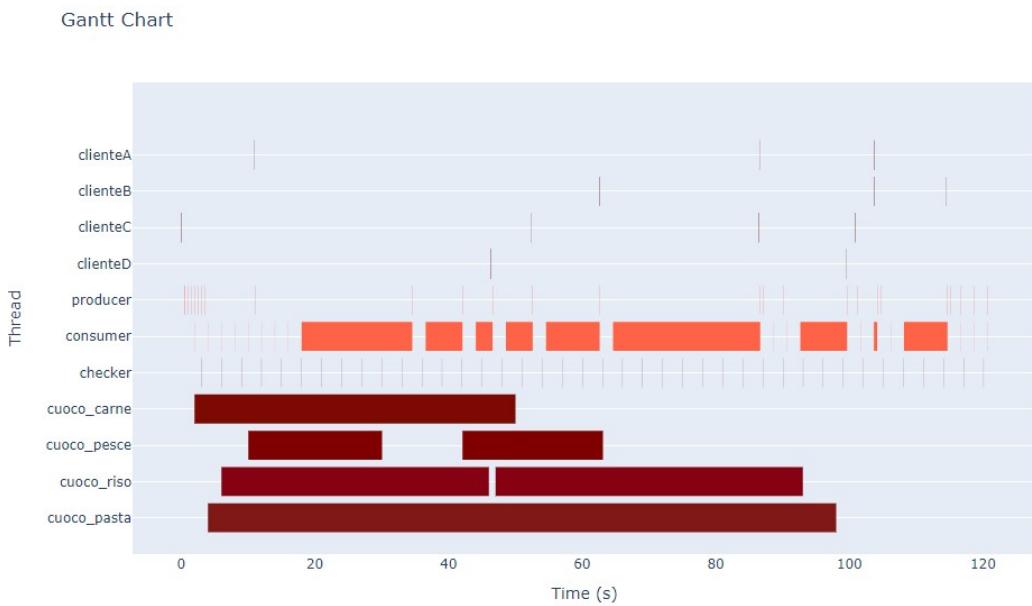
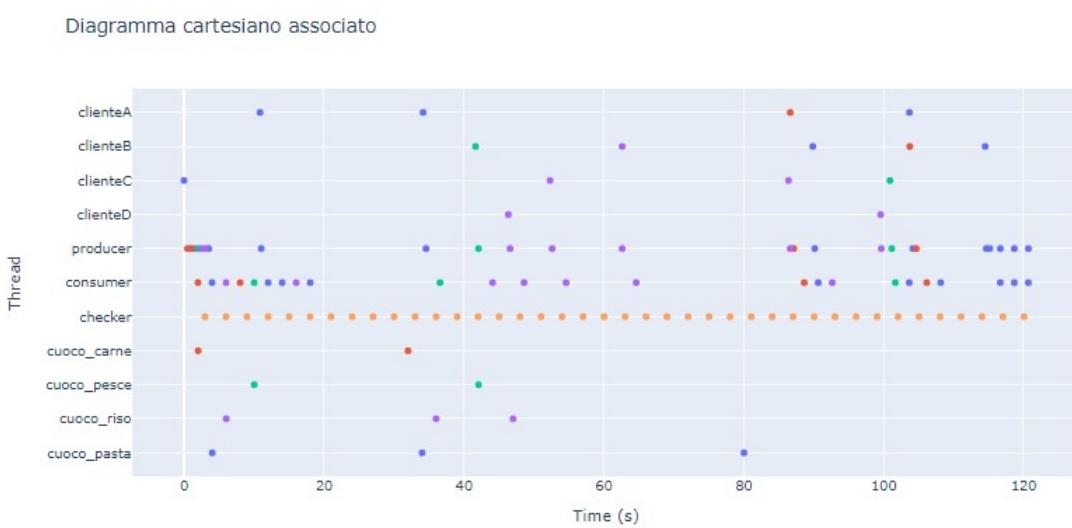


Figura 4.16: workflow dalla raccolta ai display dei dati

Diagramma di Gantt

Il diagramma di Gantt è stato utilizzato per visualizzare, nel tempo, i periodi di attività dei thread, dall’accesso al buffer all’elaborazione ordini. Si nota che il checker si attiva periodicamente per l’applicazione delle modifiche del valore di priorità delle ordinazioni accodate, con conseguente riordinamento del buffer. Inoltre, è possibile notare che, sebbene complessivamente vi sia netta separazione dei periodi di accesso, ci sono momenti di concorrenzialità tra consumer e checker.

Nel diagramma cartesiano associato viene offerta una prospettiva sui dati coinvolti nell’esecuzione dei thread: in particolare, si può notare che la priorità con cui un ordine entra in coda può alle volte determinarne la sua uscita anticipata dal buffer.

**Figura 4.17:** grafo di Gantt**Figura 4.18:** grafo cartesiano

Bibliografia

- [1] Ambroise C. aka.Hartesic. *GitHub Readme SonarQube*. Accessed on May 13, 2024. GitHub. 2024. URL: <https://github.com/SonarSource/sonarqube/blob/master/README.md>.
- [2] Baeldung. *Guide on Loading Initial Data with Spring Boot*. URL: <https://www.baeldung.com/spring-boot-data-sql-and-schema-sql> (visitato il giorno 03/05/2024).
- [3] Baeldung. *Interface Driven Controllers*. URL: <https://www.baeldung.com/spring-interface-driven-controllers> (visitato il giorno 02/05/2024).
- [4] Baeldung. *MockMvc Integration Tests*. URL: <https://www.baeldung.com/integration-testing-in-spring> (visitato il giorno 12/04/2024).
- [5] Baeldung. *Object Mapper*. URL: <https://www.baeldung.com/jackson-object-mapper-tutorial> (visitato il giorno 10/04/2024).
- [6] Baeldung. *Spring Data JPA @Query*. URL: <https://www.baeldung.com/spring-data-jpa-query> (visitato il giorno 02/05/2024).
- [7] Baeldung. *Testing Kafka*. URL: <https://www.baeldung.com/spring-boot-kafka-testing> (visitato il giorno 02/05/2024).
- [8] Happy Coders. *Filosofia Esagonale e Microservizi*. URL: <https://www.happycoders.eu/software-craftsmanship/hexagonal-architecture/> (visitato il giorno 02/05/2024).
- [9] *Continuous Integration with GitHub Actions*. URL: <https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration> (visitato il giorno 22/04/2024).
- [10] *Creating a GitHub Pages site*. Accessed on May 9, 2024. GitHub. 2024. URL: <https://docs.github.com/en/pages/getting-started-with-github-pages/creating-a-github-pages-site>.
- [11] *Docker overview*. URL: <https://docs.docker.com/get-started/overview/> (visitato il giorno 06/05/2024).
- [12] *Documentazione Jacoco*. Accessed on May 13, 2024. Jacoco. 2024. URL: <https://www.jacoco.org/jacoco/trunk/index.html>.
- [13] geeksforgeeks. *Indexed Priority Queue with Implementation*. 2022. URL: <https://www.geeksforgeeks.org/indexed-priority-queue-with-implementation/> (visitato il giorno 29/11/2022).

- [14] Roman Glushach. *Hexagonal Architecture: The Secret to Scalable and Maintainable Code for Modern Software*. URL: <https://romanglushach.medium.com/hexagonal-architecture-the-secret-to-scalable-and-maintainable-code-for-modern-software-d345fdb47347> (visitato il giorno 02/05/2024).
- [15] Google Colab. URL: <https://colab.google/> (visitato il giorno 14/05/2024).
- [16] H2 DB. URL: https://www.h2database.com/html/features.html#in-memory_databases (visitato il giorno 02/05/2024).
- [17] Arho Huttunen. *Testing Jackson*. URL: <https://www.arhohuttunen.com/spring-boot-jsontest/> (visitato il giorno 29/04/2024).
- [18] Jamie Shiell. *Checkstyle plugin for intelliJ IDE*. 2020. URL: <https://plugins.jetbrains.com/plugin/1065-checkstyle-idea> (visitato il giorno 09/05/2024).
- [19] joelparkhenderson. *Monorepo vs. Polyrepo: architecture for source code management (SCM)*. URL: <https://github.com/joelparkerhenderson/monorepo-vs-polyrepo?tab=readme-ov-file#what-is-polyrepo> (visitato il giorno 06/05/2024).
- [20] Kevin. *How to Implement Port and Adapters in Hexagonal Architecture with Java*. URL: <https://1kevinson.com/how-to-implement-port-and-adapters-in-hexagonal-architecture-with-java/> (visitato il giorno 22/04/2024).
- [21] Kevin Kouomeu. *SOLID Principles*. URL: <https://1kevinson.com/solid-principles-timeless-wisdom-on-building-high-quality-software/> (visitato il giorno 08/05/2024).
- [22] ModelMapper. *ModelMapper - Getting Started*. URL: <https://modelmapper.org/getting-started/> (visitato il giorno 01/05/2024).
- [23] Baeldung Mona Mohamadinia. *Health Indicators in Spring Boot*. URL: <https://www.baeldung.com/spring-boot-health-indicators> (visitato il giorno 08/05/2024).
- [24] Obsidian Dynamics. *Kafdrop*. <https://github.com/obsidiandynamics/kafdrop>. (Visitato il giorno 25/04/2024).
- [25] PhoenixNAP. *How to Install Python 3 on Windows*. 2022. URL: <https://phoenixnap.com/kb/how-to-install-python-3-windows> (visitato il giorno 09/05/2024).
- [26] Sandro Pinna. *Continuous Delivery & Continuous Deployment: dal rilascio manuale a docker*. URL: <https://newsroom.spindox.it/continuous-delivery-continuous-deployment-docker/> (visitato il giorno 06/05/2024).

- [27] Inc. Pivotal Software. *Spring MVC Test Framework*. 2024. URL: <https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html> (visitato il giorno 05/05/2024).
- [28] Robert Sedgewick, Kevin Wayne. *Index priority queue of princeton.edu*. Accessed on May 9, 2024. javadoc. 2012. URL: <https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/IndexMinPQ.html>.
- [29] Spring. *Building a RESTful Web Service*. 2024. URL: <https://spring.io/guides/gs/rest-service> (visitato il giorno 14/04/2024).
- [30] Spring Boot. URL: <https://spring.io/projects/spring-boot> (visitato il giorno 02/05/2024).
- [31] Spring Data JPA Reference Documentation. URL: <https://docs.spring.io/spring-data/jpa/reference/> (visitato il giorno 04/05/2024).
- [32] Spring Kafka. URL: <https://docs.spring.io/spring-kafka/reference/index.html> (visitato il giorno 02/05/2024).
- [33] Web Layer Test. URL: <https://spring.io/guides/gs/testing-web> (visitato il giorno 15/04/2024).

