



**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	3
1 Iterazione 0	7
1.1 Introduzione	7
1.2 Requisiti funzionali	8
1.2.1 Use case stories: Amministratore	8
1.2.2 Use case stories: Cliente	9
1.2.3 Use case stories: Cuoco	9
1.2.4 Use case stories: Cassiere	10
1.2.5 Priorità dei casi d'uso	10
1.2.6 Use case diagram	11
1.3 Requisiti non funzionali	13
1.3.1 Performance	13
1.3.2 Integrabilità	13
1.3.3 Modificabilità	13
1.3.4 Testabilità	13
1.3.5 Sicurezza	13
1.4 Topologia	14
1.5 Toolchain	15
1.5.1 Modellazione	15
1.5.2 Stack applicativo	15
1.5.3 Deployment	15
1.5.4 Gestore repository	15
1.5.5 Continuous Integration	15
1.5.6 Analisi statica	15
1.5.7 Analisi dinamica	15
1.5.8 Documentazione e organizzazione del team	16
1.5.9 Modello di sviluppo	16
2 Iterazione 1	17
2.1 Introduzione	17
2.2 Use Cases	18
2.2.1 Gruppo Sistema	18
2.2.2 Gruppo cliente	19
2.2.3 Gruppo cuoco	19

2.3 Component Diagram	20
2.3.1 Sistema ServeEasy	20
2.3.2 Gestione Comanda	21
2.3.3 Gestione Cliente	26
2.3.4 Gestione Cucina	29
2.4 Database	32
2.4.1 Modello Entità-Relazione	32
2.4.2 Modello logico	33
2.5 Interface Class Diagram	35
2.5.1 Interfacce Gestione Comanda	35
2.5.2 Interfacce Gestione Cucina	36
2.5.3 Interfacce Gestione Cliente	37
2.6 Costruzione dello scheletro	38
2.6.1 Organizzazione area di lavoro	38
2.6.2 Definizione Interfacce	39
2.6.3 Adattatore Kafka	41
2.6.4 Creazione del Database	52
2.6.5 Adattatore JPA	53
2.6.6 Continuous Integration	60
2.6.7 DTO	62
2.6.8 Interfaccia di Test	66
2.7 Documentazione delle API	78
3 Iterazione 2	97
3.1 Algoritmo	97
3.1.1 Briefing	97
3.1.2 Organizzazione	97
3.1.3 Struttura dati	98
3.1.4 Funzione di priorità	101
3.1.5 Diagramma di flusso	104

Elenco delle figure

1.1	Diagramma dei casi d'uso	12
1.2	Topologia del sistema	14
2.1	Casi d'uso presi in considerazione nell'iterazione 1	18
2.2	Component diagram - ServeEasy	20
2.3	Component diagram - System	20
2.4	Architettura esagonale per il microservizio Gestione comanda	22
2.5	Component diagram - Gestione Comanda	23
2.6	Component diagram - Gestione Comanda - Infrasructure	23
2.7	Component diagram - Gestione Comanda - Domain	24
2.8	Component diagram - Gestione Comanda - Interface	25
2.9	Component diagram - Gestione Cliente	26
2.10	Component diagram - Gestione Cliente - Infrastructure	27
2.11	Component diagram - Gestione Cliente - Domain	28
2.12	Component diagram - Gestione Cliente - Interface	28
2.13	Component diagram - Gestione Cucina	29
2.14	Component diagram - Gestione Cucina - Infrastructure	29
2.15	Component diagram - Gestione Cucina - Domain	30
2.16	Component diagram - Gestione Cucina - Interface	31
2.17	Modello Entità-relazione	32
2.18	Modello Logico	33
2.19	Interface class diagram - Gestione Comanda	35
2.20	Interface class diagram - Gestione Cucina	36
2.21	Interface class diagram - Gestione Cliente	37
2.22	Organizzazione del lavoro cloud e locale, CI/CD e deployment	39
2.23	Esempio funzionamento kafdrop	46
2.24	CI merge a pull-request	61
2.25	CI main check	61
2.26	Component diagram - Gestione Comanda - Interface con Test	67
2.27	Esempio applicativo Postman	77
3.1	Strutture dati dell'algoritmo	99
3.2	Diagramma di flusso	105

Elenco dei codici

2.1	Query del database in SQL	34
2.2	Interfaccia MessagePort	39
2.3	Interfaccia DataPort	40
2.4	Interfaccia NotifyOrderEvent	41
2.5	Setup del docker-compose.yaml per l'adattatore Kafka	42
2.6	Aggiornamento dipendenze nel pom.xml per includere spring-kafka	42
2.7	Aggiornamento del file ‘application.yml‘ per il producer Kafka	43
2.8	Classe di configurazione KafkaConfig.java	43
2.9	Classe di configurazione JsonConfig.java	43
2.10	Classe del producer kafka CucinaPubProducer.java	44
2.11	Operazione di post sul topic kafka	45
2.12	Operazione di get sul topic kafka	45
2.13	Aggiornamento del docker-compose.yaml perl per Kafdrop	45
2.14	Aggiornamento dipendenze nel pom.xml per includere spring-kafka-test .	46
2.15	Aggiornamento del file ‘application.properties‘ di test per il producer kafka	47
2.16	Test di integrazione per il producer CucinaPubProducer	47
2.17	Aggiornamento del file ‘application.yml‘ per il consumer Kafka	49
2.18	Classe del consumer kafka SubClienteAdapter.java	49
2.19	Aggiornamento del file ‘application.properties‘ di test per il consumer Kafka	51
2.20	Test di integrazione per il consumer SubClienteAdapter	51
2.21	Aggiornamento dipendenze nel pom.xml per includere spring-data-jpa .	53
2.22	Aggiornamento del file ‘application.yml‘ per Spring Data JPA	53
2.23	Classe entità OrdineEntity.java	53
2.24	Classe repository OrdineRepository.java	55
2.25	Classe adattatore JPA JPADBAdapter.java	56
2.26	Aggiornamento del file pom.xml per la dipendenza di H2	57
2.27	Aggiornamento del file application.properties per i test con H2	57
2.28	Classe adattatore JPA JPADBAdapter.java	58
2.29	Creazione del file maven.yml	60
2.30	Aggiornamento dipendenze nel pom.xml per includere model-mapper .	62
2.31	Classe di configurazione MapperConfig.java	62
2.32	Classe DTO per l’entità ordine OrderDTO.java	63
2.33	Interfaccia Mapper.java	64
2.34	Classe OrdineMapper.java implementazione di ModelMapper.java	64
2.35	Test di integrazione OrdineMapperTests.java	65

2.36 Aggiornamento dipendenze nel pom.xml per includere spring-web	67
2.37 Interfaccia TestAPI.java	68
2.38 Classe REST Controller di test TestController.java	71
2.39 Classe Test per il REST Controller di test TestControllerTests.java	75

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).

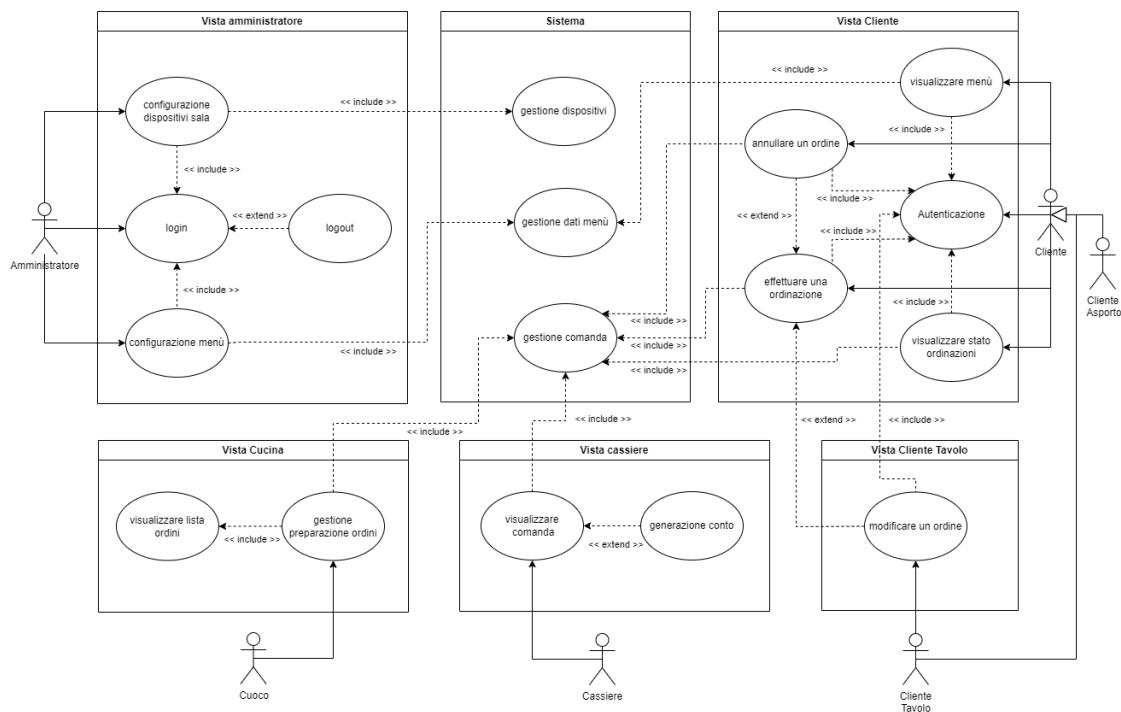


Figura 1.1: Diagramma dei casi d'uso

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.

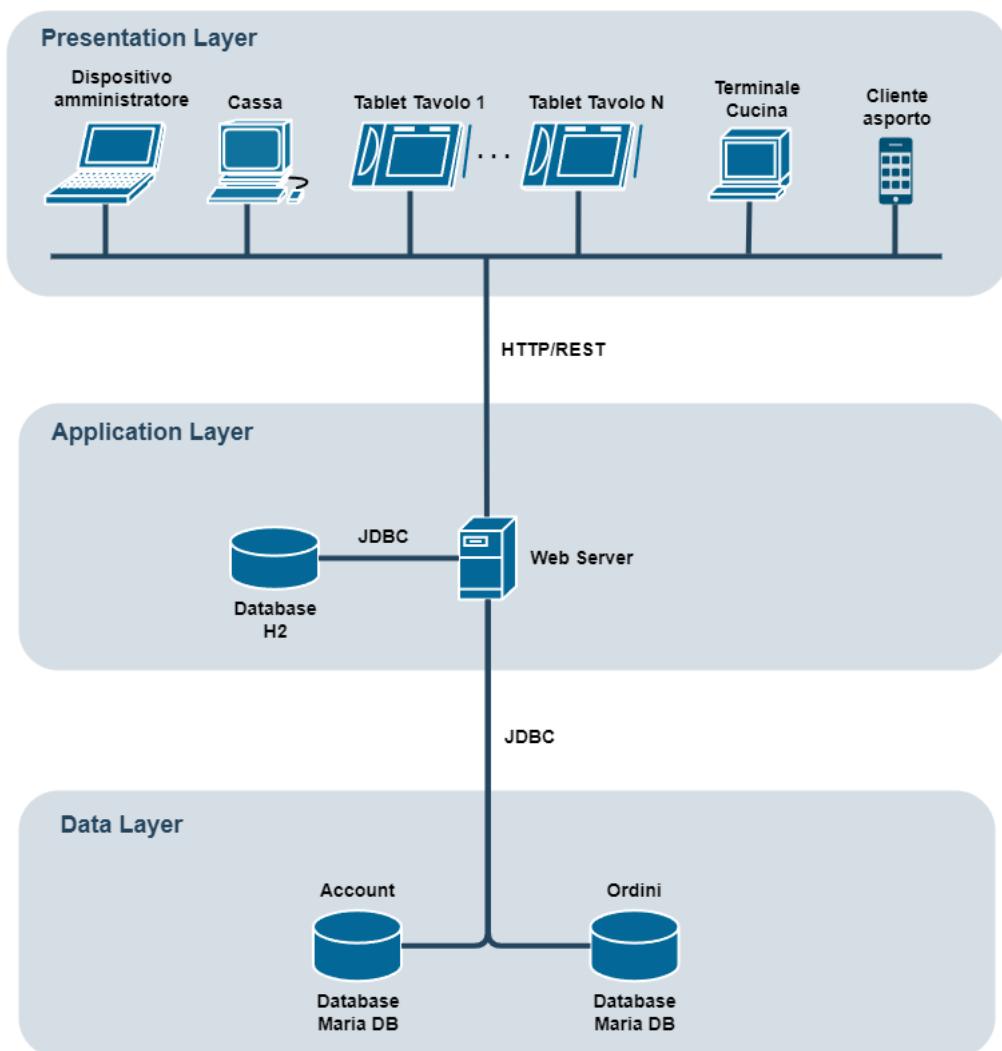


Figura 1.2: Topologia del sistema

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 e topologia;

1.5.2 Stack applicativo

- Java Spring Boot 3.2.4: back-end;
- MariaDB: database per l'archiviazione;
- H2: database in-memory;

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
- 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, del database e dell'algoritmo. 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

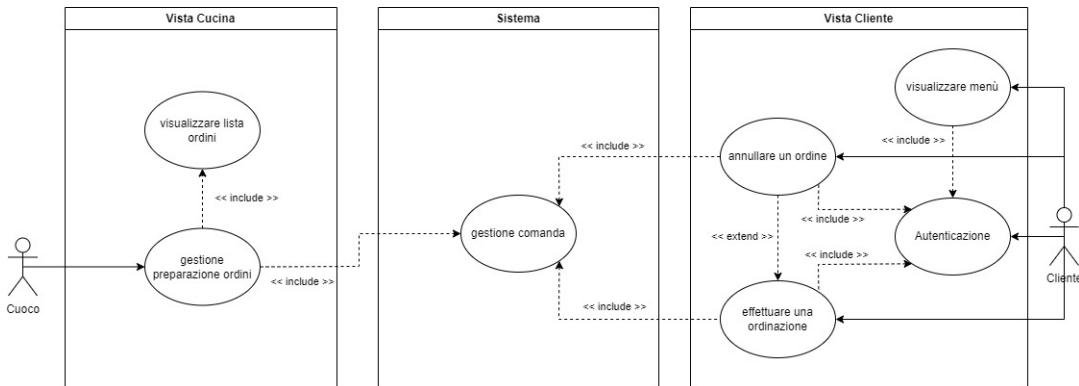


Figura 2.1: Casi d'uso presi in considerazione nell'iterazione 1

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

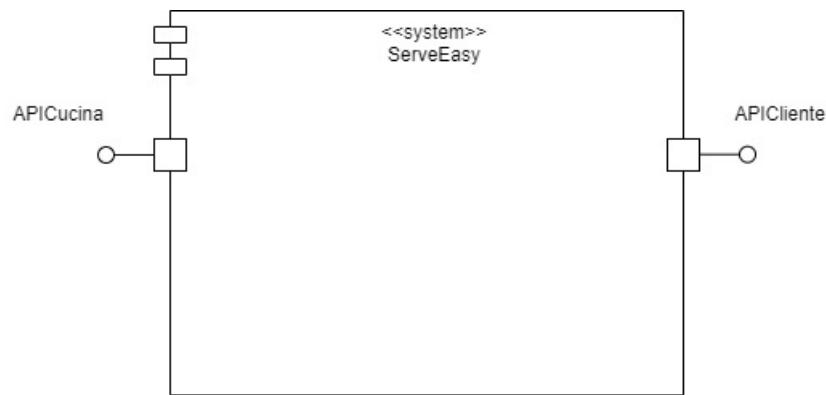


Figura 2.2: Component diagram - ServeEasy

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

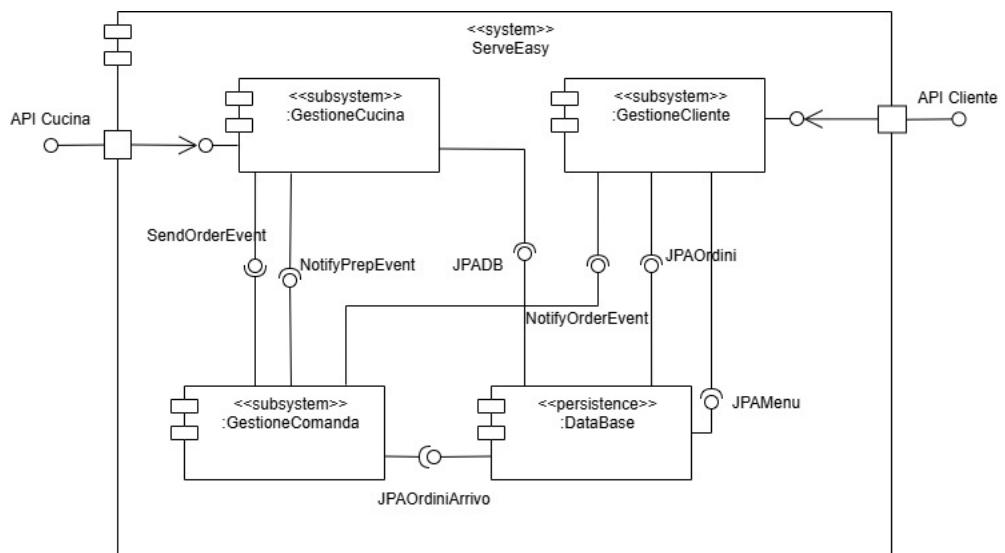


Figura 2.3: Component diagram - System

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.

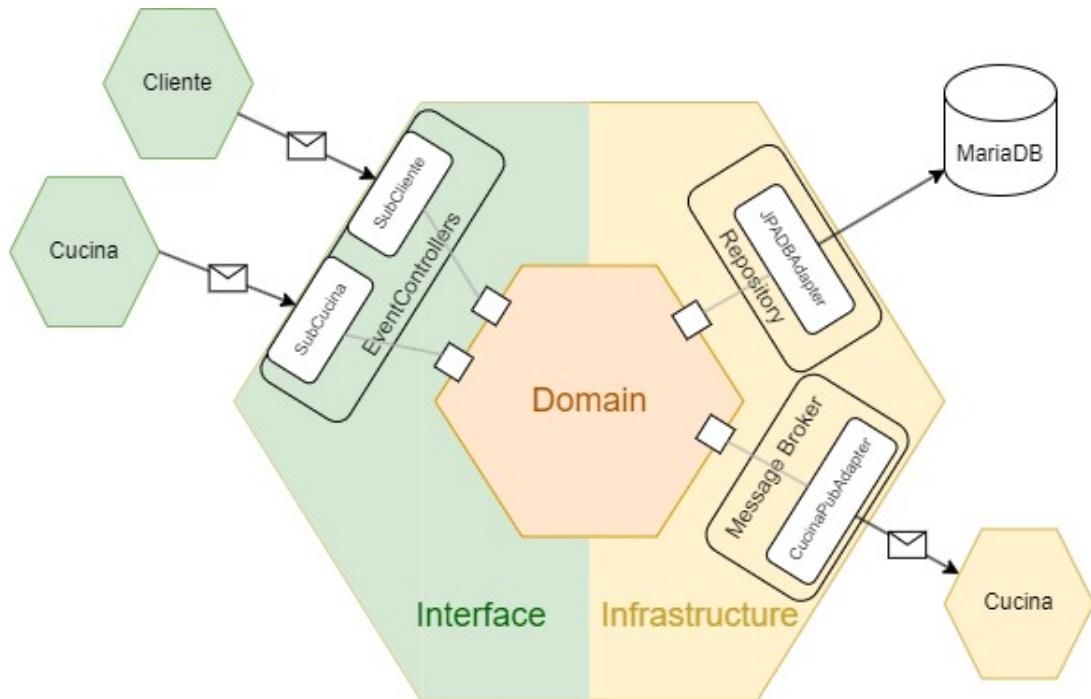
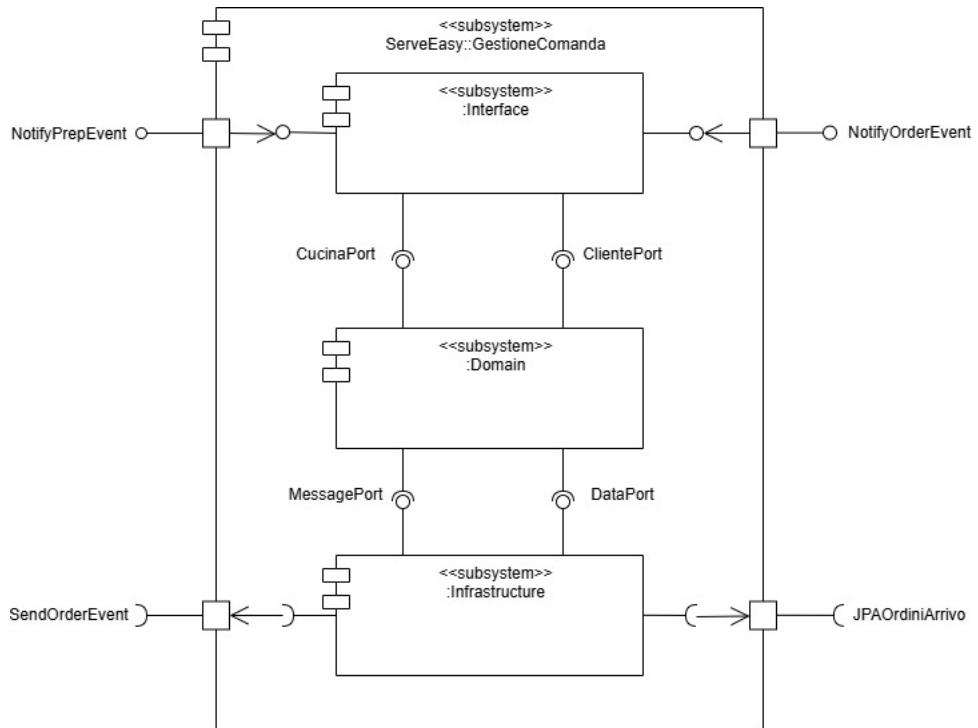
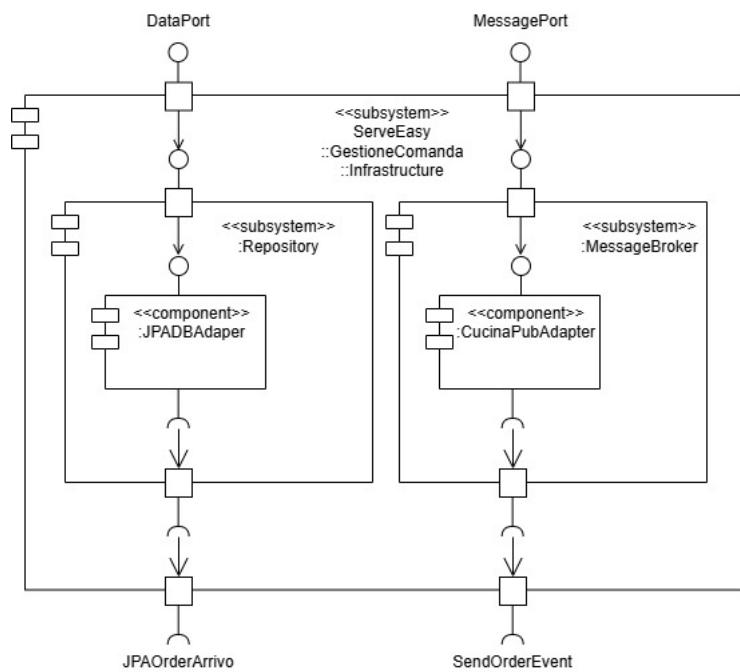


Figura 2.4: Architettura esagonale per il microservizio Gestione comanda

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

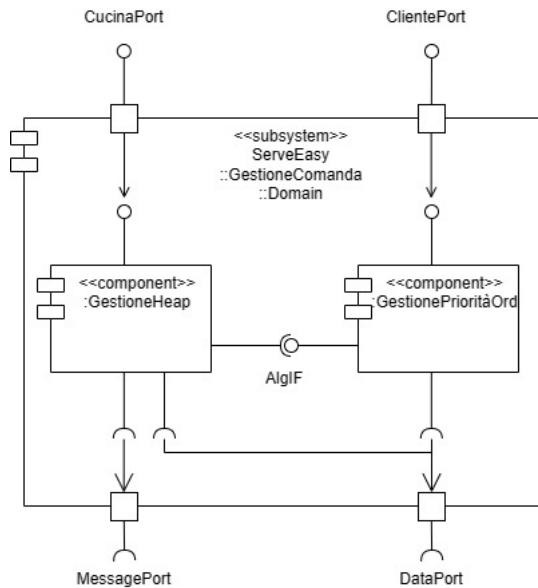
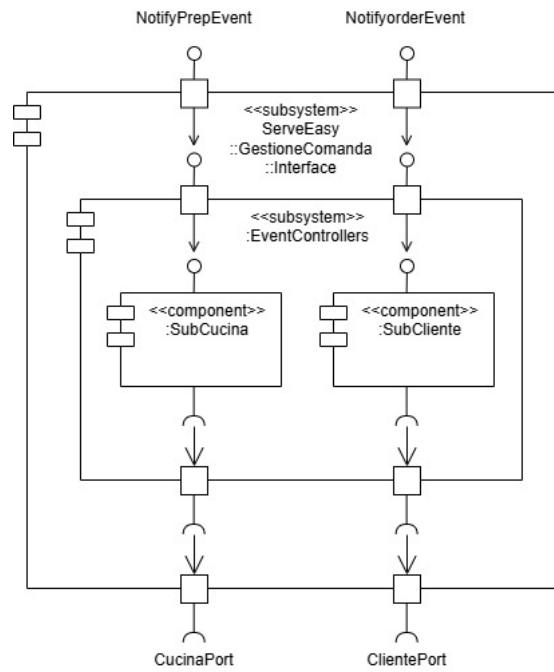


Figura 2.7: Component diagram - Gestione Comanda - Domain

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

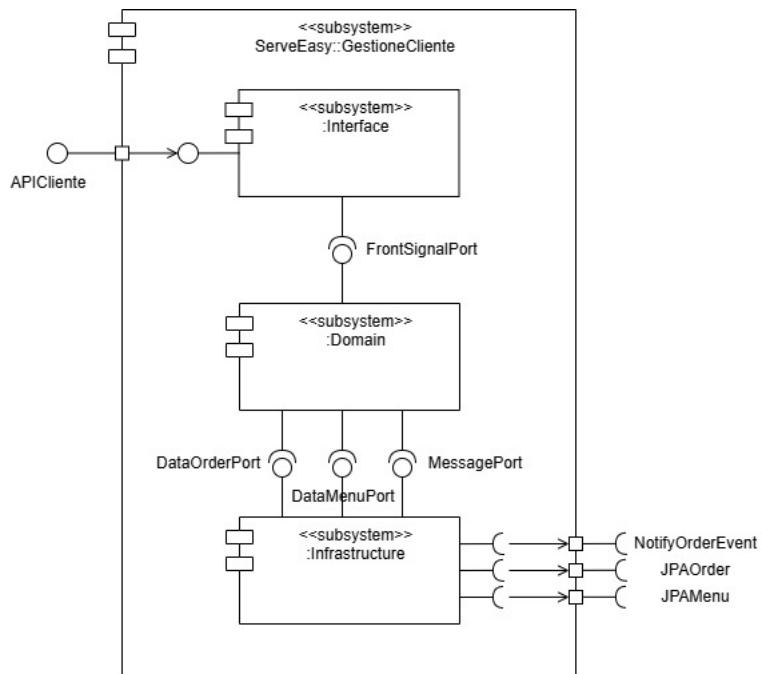


Figura 2.9: Component diagram - Gestione Cliente

zoom-in infrastructure di gestione cliente

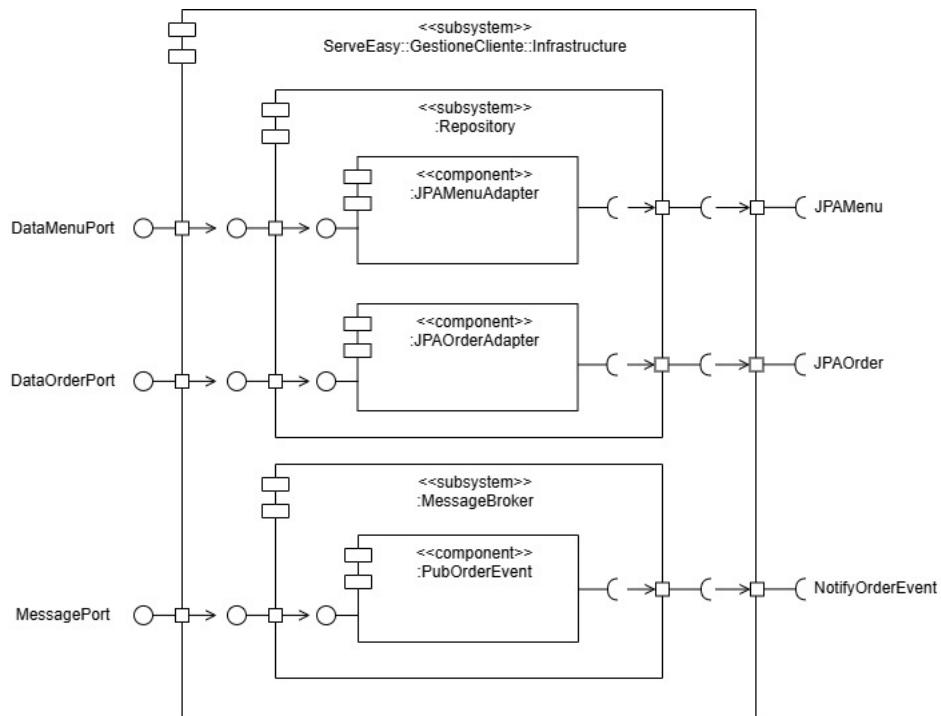


Figura 2.10: Component diagram - Gestione Cliente - Infrastructure

- 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

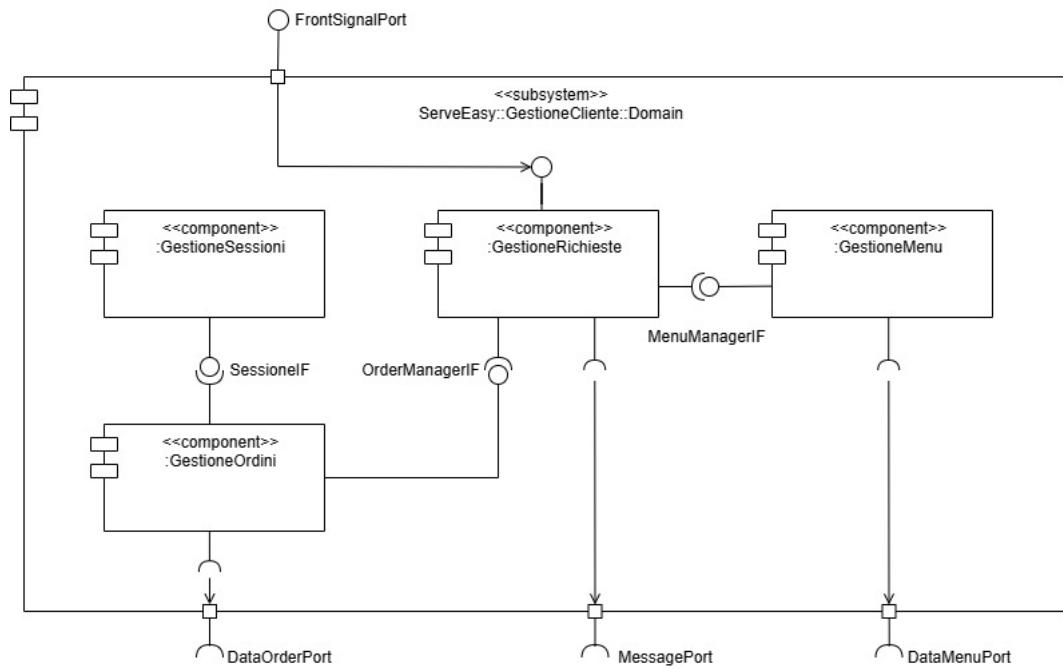


Figura 2.11: Component diagram - Gestione Cliente - Domain

zoom-in interface di gestione cliente

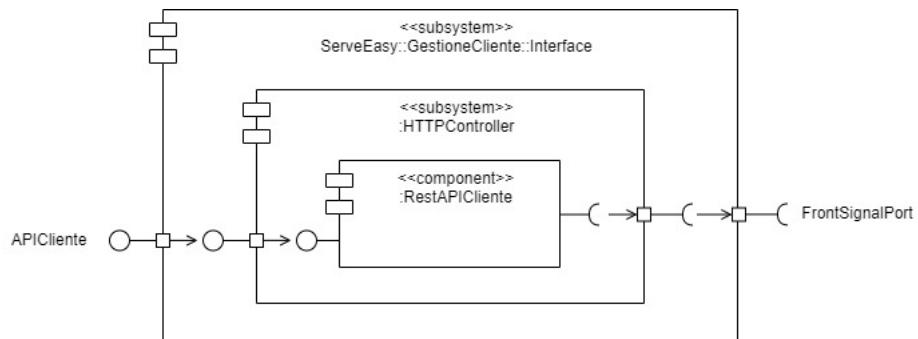


Figura 2.12: Component diagram - Gestione Cliente - Interface

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

2.3.4 Gestione Cucina

zoom-in gestione cucina

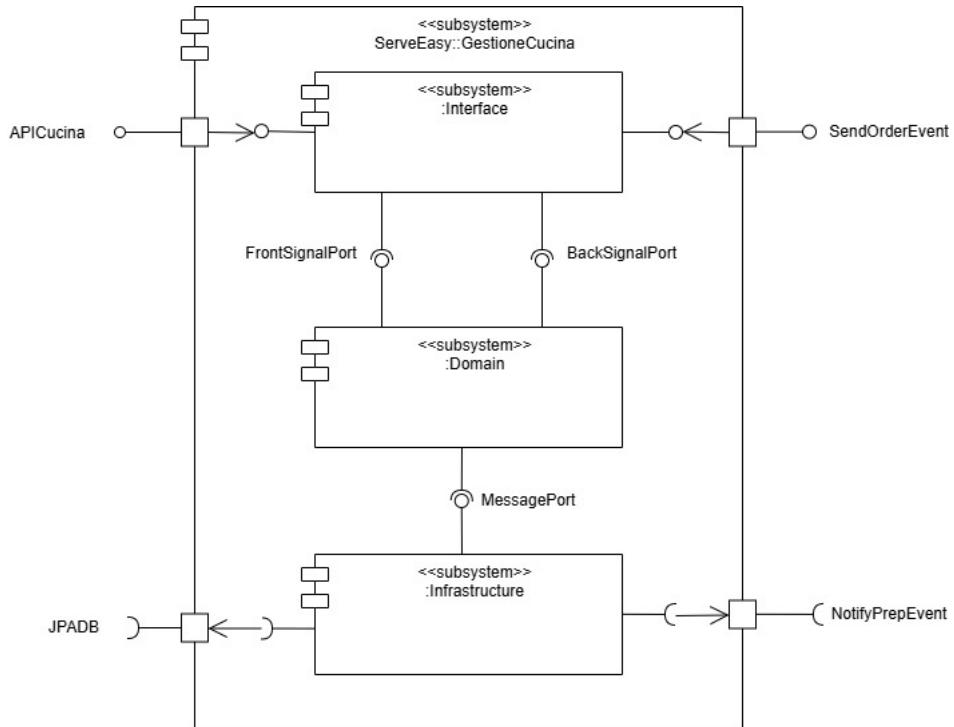


Figura 2.13: Component diagram - Gestione Cucina

zoom-in infrastructure di gestione cucina

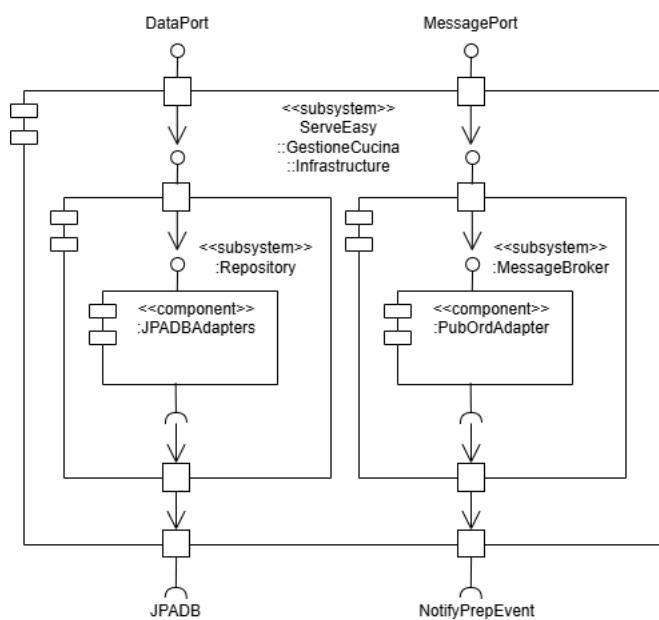


Figura 2.14: Component diagram - Gestione Cucina - Infrastructure

- 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

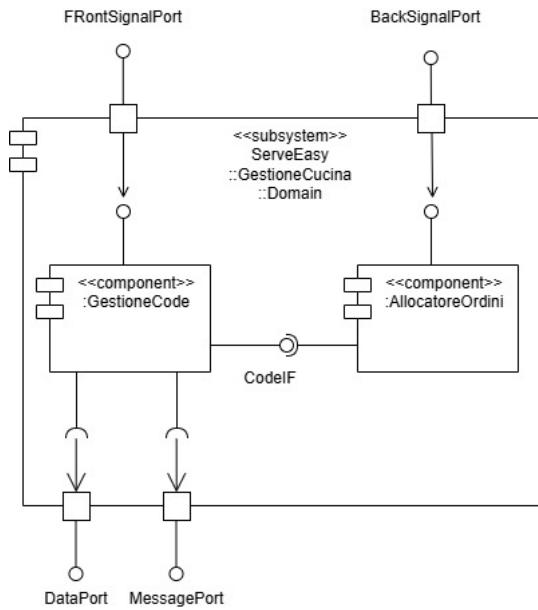
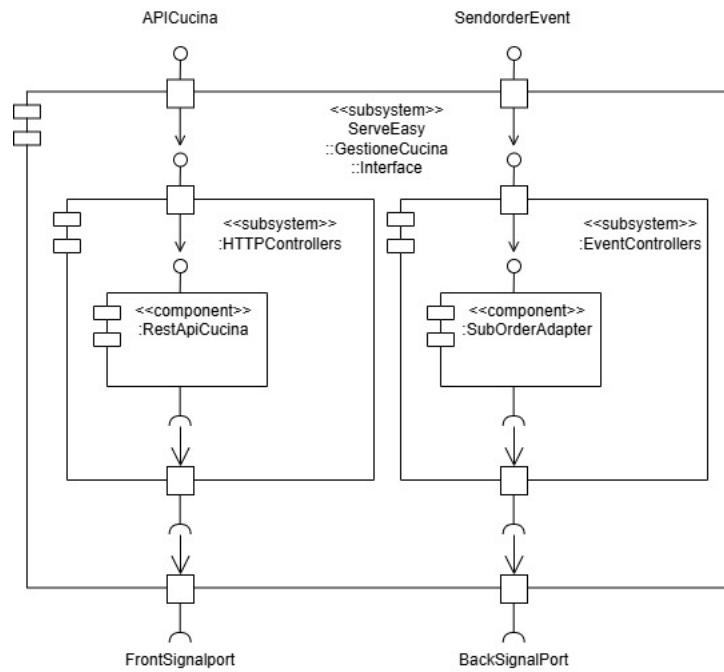


Figura 2.15: Component diagram - Gestione Cucina - Domain

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.

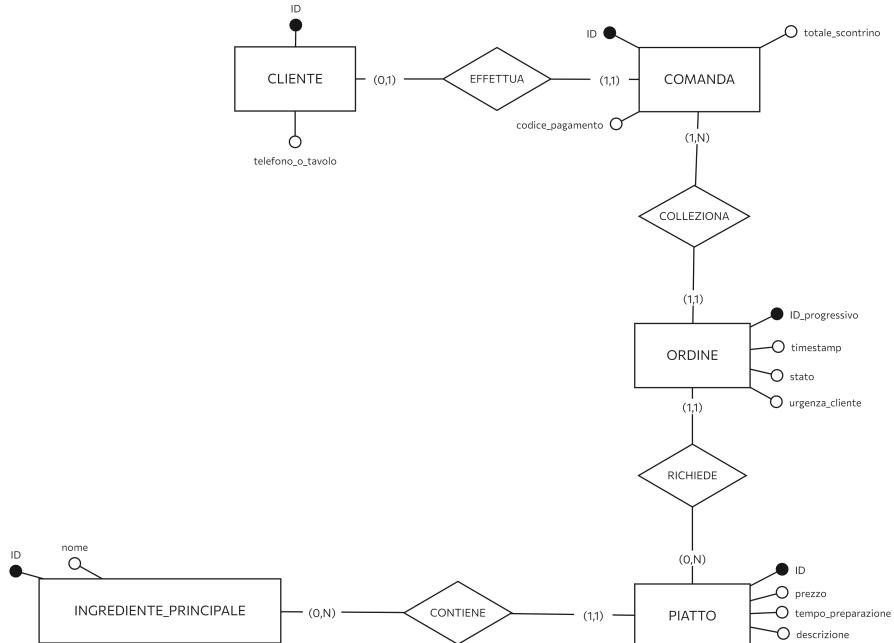


Figura 2.17: Modello Entità-relazione

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;

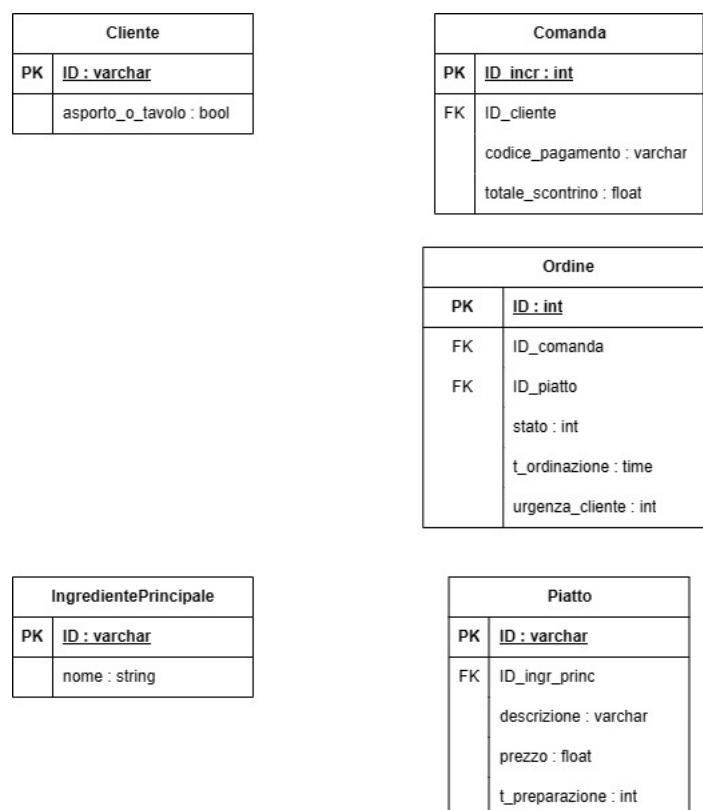


Figura 2.18: Modello Logico

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 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.

2.5.1 Interfacce Gestione Comanda

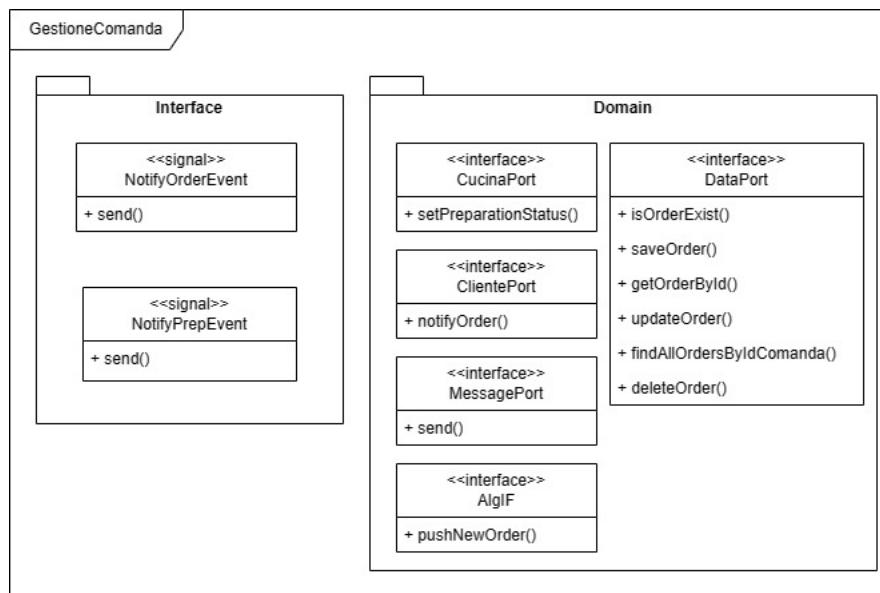


Figura 2.19: Interface class diagram - Gestione Comanda

2.5.2 Interfacce Gestione Cucina

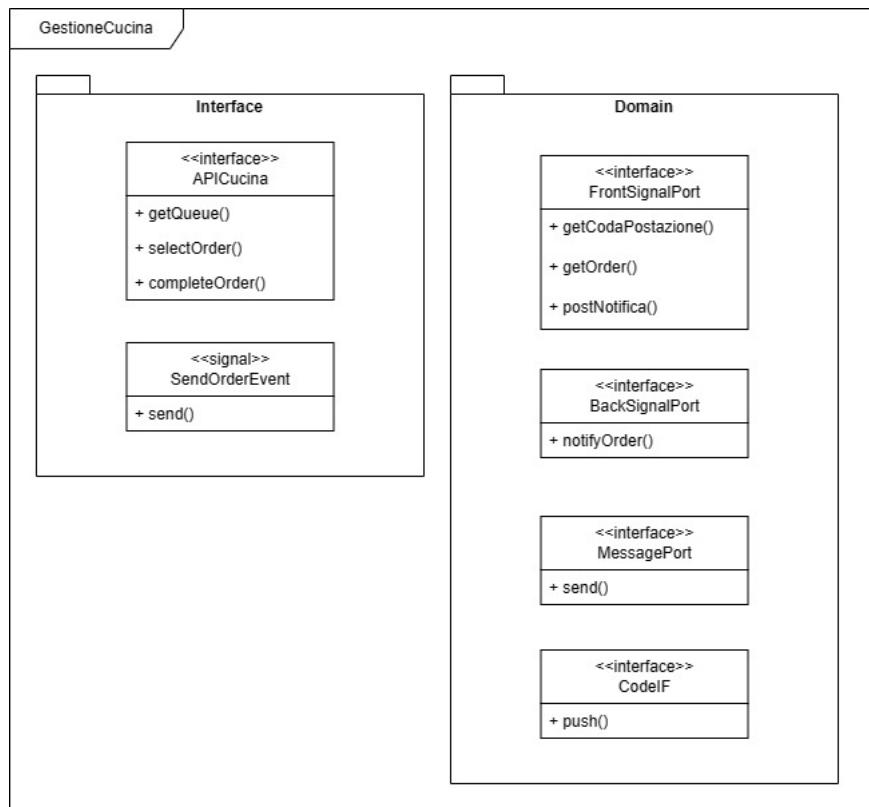


Figura 2.20: Interface class diagram - Gestione Cucina

2.5.3 Interfacce Gestione Cliente

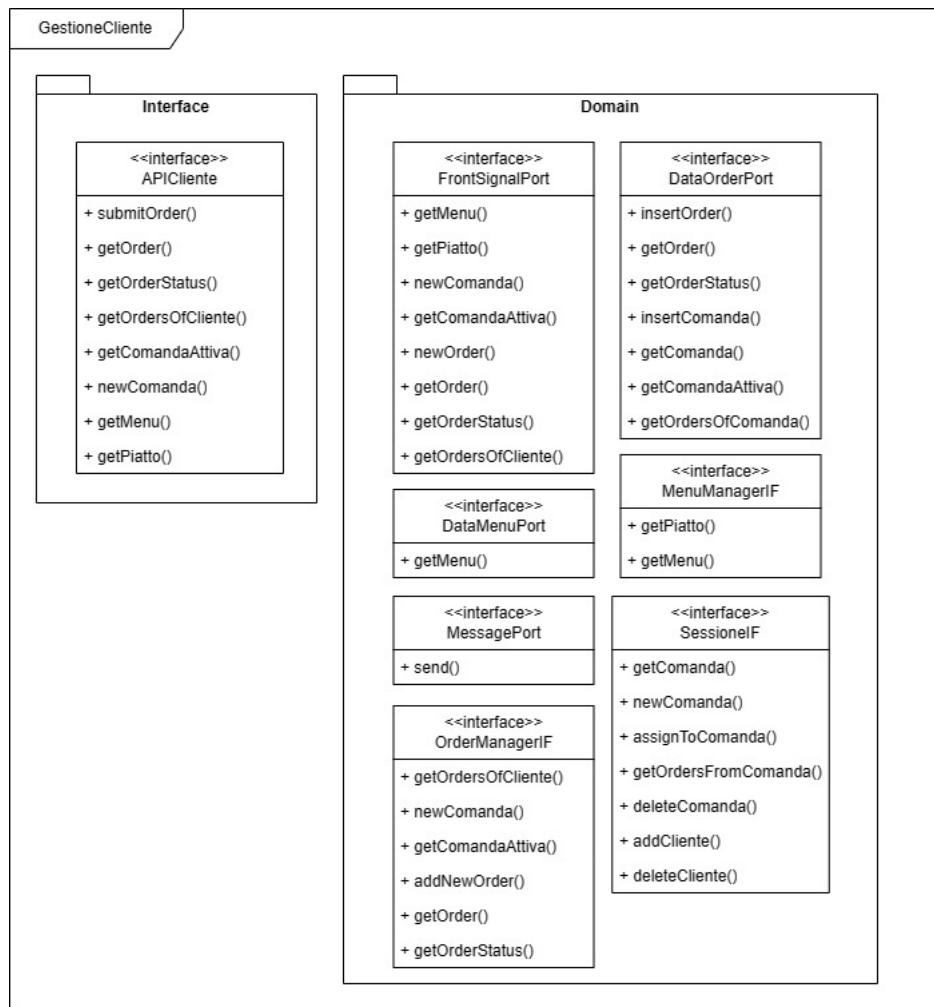


Figura 2.21: Interface class diagram - Gestione Cliente

2.6 Costruzione dello scheletro

2.6.1 Organizzazione area di lavoro

Per l'implementazione dei microservizi che compongono il sistema ServeEasy, attualmente delineato al primo zoom-in, è stato applicato un approccio “polyrepo”[12], 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 lo shipping dell'applicazione, per mezzo di un Dockerfile, salvandola in remoto su registry DockerHub.

In particolare, nel punto 2 si parla di Continuous Delivery e non Continuous Deployment, in quanto il deployment della nuova immagine va eseguita manualmente [16].

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.

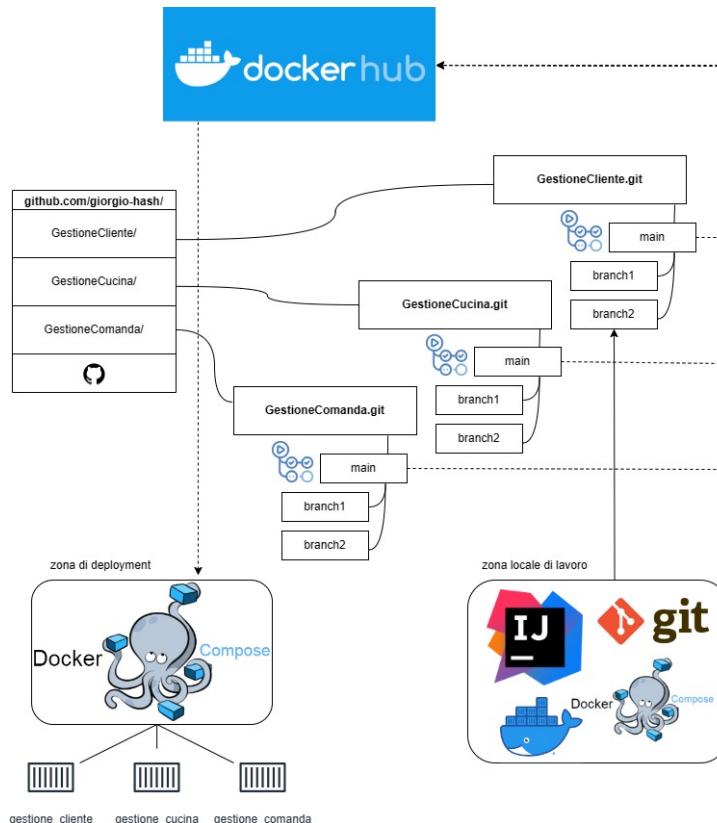


Figura 2.22: Organizzazione del lavoro cloud e locale, CI/CD e deployment

2.6.2 Definizione Interfacce

```

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 2.2: 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     */  
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);
```

```

44
45     /**
46      * Cancella l'ordine con il dato ID dal database
47      *
48      * @param id id dell'ordine da eliminare
49      */
50     void deleteOrder(int id);
51 }
```

Codice 2.3: Interfaccia DataPort

```

1 public interface NotifyOrderEvent {
2
3     /**
4      * Riceve un messaggio tramite Kafka dal servizio gestioneCliente
5      * in merito all'avvenuta ordinazione
6      * da parte di un cliente
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    * @param partition numero di partizione sul quale si riceve il
12     * messaggio
13    * @param offset numero di offset che presenta il messaggio
14     * ricevuto
15    */
16    @KafkaListener(id = "${spring.kafka.consumer.gestioneCliente.group-
17    id}", topics = "${spring.kafka.consumer.gestioneCliente.topic}")
18    void receive(String message, String topic, Integer partition, Long
19    offset) throws JsonProcessingException;
20
21    /**
22     * Restituisce l'ultima notifica letta dal listener
23     *
24     * @return oggetto notifica letto dal listener
25     */
26    NotificaOrdineDTO getLastMessageReceived();
27 }
```

Codice 2.4: Interfaccia NotifyOrderEvent

2.6.3 Adattatore Kafka

Lo sviluppo del componente adibito all'adattatore Kafa 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 [21] 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 2.5):

```

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
26       KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
27       KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1

```

Codice 2.5: Setup del docker-compose.yaml per l'adattatore Kafka

Fatto ciò si è passati ad aggiornare il file "pom.xml" (Codice 2.6) con la seguente dipendenza per Kafka:

```

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

```

Codice 2.6: Aggiornamento dipendenze nel pom.xml per includere spring-kafka

Successivamente nel file "application.yml" (Codice 2.7) 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 2.7: Aggiornamento del file ‘application.yml’ per il producer Kafka

Sono stati poi creati i primi bean di configurazione nella classe "KafkaConfig.java" (Codice 2.8):

```

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 2.8: Classe di configurazione KafkaConfig.java

E di Object Mapper (Codice 2.9) 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:
mm:ss.SSS"));
10        return objectMapper;
11    }
12}

```

Codice 2.9: 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 2.10) che implementa la MessagePort (Codice 2.2 a pagina 39):

```

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(
26         ordineDTO);
27
28     // invia messaggio sul topic specificato
29     CompletableFuture<SendResult<String, String>> future =
30     kafkaTemplate.send(topic, payload);
31     future.whenComplete((result, ex)->{
32         if(ex == null){
33             log.info("Sent Message=[ " + payload + " ] with offset=[ "
34             + result.getRecordMetadata().offset() + " ]");
35         }
36         else{
37             log.info("Unable to send message=[ " + payload + " ] due
38             to : " + ex.getMessage());
39         }
40     });
41 }

```

Codice 2.10: Classe del producer kafka CucinaPubProducer.java

Si può notare che questa classe richiede un bean di ObjectMapper (Codice 2.9 a pagina 43) e KafkaTemplate (Codice 2.8 a pagina 43) definiti al passo precedente, mentre richiede dall'application.yml (Codice 2.7 a pagina 43) 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 2.11: 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 2.12: Operazione di get sul topic kafka

Step 4 - User Interface Verificato il funzionamento tramite log si è passati ad usare Kafdrop[15] ossia una User Interface (UI) per i topic di Kafka, integrandolo nella nostra applicazione con le seguenti istruzioni nel docker-compose.yml:(Codice 2.13):

```

1 services:

```

```

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

```

Codice 2.13: Aggiornamento del docker-compose.yaml per Kafdrop

Di seguito (Figura 2.23) viene mostrato un esempio della UI in funzione sulla porta localhost:9000 che mostra tutti i messaggi postati sul topic *sendOrderEvent* ordinati in base all'offset:

Offset	Key	Timestamp	Headers	Message
0	empty	2024-05-03 11:10:22.444	empty	{"id":17,"idComanda":7,"idPiatto":"SUH724","stato":1,"urgenzaCliente":0,"tordinazione":"2024-05-03 13:10:21.359"}
1	empty	2024-05-03 11:54:02.799	empty	{"id":18,"idComanda":4,"idPiatto":"CAR123","stato":0,"urgenzaCliente":1,"tordinazione":"2024-05-03 13:54:02.555"}
2	empty	2024-05-03 11:54:35.125	empty	{"id":19,"idComanda":3,"idPiatto":"RIS001","stato":1,"urgenzaCliente":0,"tordinazione":"2024-05-03 13:54:35.095"}

Figura 2.23: 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 [6]. 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 2.14) la seguente dipendenza:

```

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

```

Codice 2.14: Aggiornamento dipendenze nel pom.xml per includere spring-kafka-test

E aggiornato l'application.properties specifico dei test (Codice 2.15):

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

Codice 2.15: Aggiornamento del file ‘application.properties’ di test per il producer kafka

Fatto ciò si può procedere a scrivere la seguente classe di test (Codice 2.16):

```
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 testSendMessage() throws JsonProcessingException {
```

```

33     OrdineDTO ordineDTO = TestDataUtil.createOrdineDtoA();
34
35     producer.send(ordineDTO);
36     String message = objectMapper.writeValueAsString(ordineDTO);
37
38     // configurazione del consumer di Embedded Kafka
39     Map<String, Object> consumerProps = KafkaTestUtils.
40     consumerProps("testT", "false", embeddedKafka);
41     DefaultKafkaConsumerFactory<Integer, String> cf = new
42     DefaultKafkaConsumerFactory<>(consumerProps);
43     Consumer<Integer, String> consumer = cf.createConsumer();
44     embeddedKafka.consumeFromAnEmbeddedTopic(consumer, topic);
45     ConsumerRecord<Integer, String> received = KafkaTestUtils.
46     getSingleRecord(consumer, topic);
47
48     // testo il corretto invio
49     assertFalse(testAppender.events.isEmpty());
50     assertEquals("Sent Message=[ " + message + " ] with offset=[0]", 
51     testAppender.events.get(0).getFormattedMessage());
52
53     // testo la corretta ricezione
54     assertThat(received.offset()).isEqualTo(0);
55     assertThat(received.topic()).isEqualTo(topic);
56     assertThat(received.partition()).isEqualTo(0);
57     OrdineDTO ordineDTOReceived = objectMapper.readValue(received.
58     value(), OrdineDTO.class);
59     assertThat(ordineDTOReceived).isEqualTo(ordineDTO);
60
61     logger.detachAppender(testAppender);
62 }
63 }
```

Codice 2.16: 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 2.17) con la configurazione del consumer in questione:

```

1 kafka:
2   consumer:
3     gestioneCliente:
4       topic: notifyOrderEvent
5       group-id: gestioneCliente

```

Codice 2.17: Aggiornamento del file ‘application.yml‘ per il consumer Kafka

Poi si è passati direttamente ad implementare la classe SubClienteAdapter (Codice: 2.18) che implementa l’interfaccia NotifyOrderEvent (Codice: 2.4 a pagina 41)

```

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   private CountDownLatch latch = new CountDownLatch(1);
13   private final Logger logger = LoggerFactory.getLogger(
14     SubCucinaAdapter.class);
15   private NotificaOrdineDTO lastMessageReceived;
16
17   @Autowired
18   public SubClienteAdapter(final ClientePort clientePort, final
19   ObjectMapper objectMapper) {
20     this.clientePort = clientePort;
21     this.objectMapper = objectMapper;
22   }
23
24   @Override
25   public void receive(@Payload String message,
26                       @Header(KafkaHeaders.RECEIVED_TOPIC) String
27   topic,
28                       @Header(KafkaHeaders.RECEIVED_PARTITION)
29   Integer partition,
30                       @Header(KafkaHeaders.OFFSET) Long offset)
31   throws JsonProcessingException {
32     logger.info("Received a message {}, from {} topic, " +
33     message, topic, partition, offset);
34   }
35 }
```

```

27         "{} partition, and {} offset", message.toString(),
28     topic, partition, offset);
29     NotificaOrdineDTO notificaOrdineDTO = objectMapper.readValue(
30     message, NotificaOrdineDTO.class);
31     clientePort.notifyOrder(notificaOrdineDTO);
32     lastMessageReceived = notificaOrdineDTO;
33     latch.countDown();
34 }
35 /**
36  * resetta il valore del latch
37 */
38 public void resetLatch() {
39     latch = new CountDownLatch(1);
40 }
41 /**
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  * Restituisce l'ultimo messaggio letto dal listener
53  *
54  * @return l'ultimo messaggio letto dal listener
55  */
56 @Override
57 public NotificaOrdineDTO getLastMessageReceived() {
58     return lastMessageReceived;
59 }
60 }

```

Codice 2.18: 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 2.19) di test con il topic di test del consumer:

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

Codice 2.19: 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 2.20) 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", "
8 port=9092" },
9 topics = {"${spring.kafka.consumer.gestioneCliente.topic}"}))
10 public class SubClienteAdapterTests {
11     @Autowired
12     private EmbeddedKafkaBroker embeddedKafka;
13     @Autowired
14     private SubClienteAdapter subClienteAdapter; // kafka consumer
15     @Value("${spring.kafka.consumer.gestioneCliente.topic}")
16     private String topic;
17     private Logger logger;
18     private TestAppender testAppender;
19 
20    @BeforeEach
21     public void setup() {
22         subClienteAdapter.resetLatch();
23         logger = (Logger) LoggerFactory.getLogger(SubCucinaAdapter.class)
24     };
25         testAppender = new TestAppender();
26         testAppender.start();
27         logger.addAppender(testAppender);
28     }
29 
30    @Test
31     public void testOutput1() throws Exception {
```

```

30     NotificaOrdineDTO notificaOrdineDTO = TestDataUtil.
31     createNotificaOrdineDTOA();
32     String notifica = TestUtil.serialize(notificaOrdineDTO);
33
34     CompletableFuture<SendResult<Integer, String>> future = TestUtil
35       .sendMessageToTopic(topic, notifica, embeddedKafka);
36     log.info("Sent Message=[ " + notifica + " ] with offset=[0]");
37
38     boolean messageConsumed = subClienteAdapter.getLatch().await(10,
39       TimeUnit.SECONDS);
40
41     //testo il corretto invio
42     future.whenComplete((result, ex)->{
43       assertThat(ex).isNull();
44       // verifica che non sia stata sollevata alcuna eccezione
45     });
46
47     // testo la corretta ricezione
48     assertTrue(messageConsumed);
49     assertFalse(testAppender.events.isEmpty());
50     assertEquals("Received a message " + notifica + ", from " +
51       topic + " topic, 0 partition, and 0 offset", testAppender.events.
52       get(0).getFormattedMessage());
53
54     NotificaOrdineDTO notificaOrdineDTOReceived = subClienteAdapter.
55     getLastMessageReceived();
56     assertEquals(notificaOrdineDTO, notificaOrdineDTOReceived);
57     logger.detachAppender(testAppender);
58   }
59 }
```

Codice 2.20: 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.

2.6.4 Creazione del Database

TODO

2.6.5 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 [20] 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 2.21) 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 2.21: Aggiornamento dipendenze nel pom.xml per includere spring-data-jpa

Successivamente nel file "application.yml" (Codice 2.22) 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 2.22: 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 2.23) 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     /**
24      * Identificatore del piatto ordinato dal cliente
25     */
26     @Basic
27     @Column(name = "ID_piatto", nullable = false, length = 20)
28     private String idPiatto;
29
30     /**
31      * Stato dell'ordine
32      * 0: Ordine preso in carico
33      * 1: Ordine in coda di preparazione
34      * 2: Ordine in preparazione
35      * 3: Ordine preparato
36     */
37     @Basic
38     @Column(name = "stato", updatable = true)
39     private Integer stato;
40
41     /**
42      * Istante temporale in cui viene effettuata l'ordinazione
43      * pattern : "yyyy-MM-dd HH:mm:ss.SSS"
44     */
45     @Basic
46     @CreationTimestamp
47     @Column(name = "t_ordinazione", updatable = false)
48     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 2.23: 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 2.24: Classe repository OrdineRepository.java

La classe `OrdineRepository` (Codice 2.24) 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.

mentazione 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 2.25) che implementa la DataPort (Codice 2.3 a pagina 40):

```
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    }
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    {
29        ordineEntity.setId(id);
30
31        return ordineRepository.findById(ordineEntity.getId()).map(
32            existingOrder -> {
33                Optional.ofNullable(ordineEntity.getStato()).ifPresent(
34                    existingOrder::setStato);
35                Optional.ofNullable(ordineEntity.getUrgenzaCliente()).ifPresent(
36                    existingOrder::setUrgenzaCliente);
37                return ordineRepository.save(existingOrder);
38            }
39        );
40    }
41}
```

```

34         }).orElseThrow(() -> new RuntimeException("Order does not exist
35         "));
36     }
37
38     @Override
39     public List<OrdineEntity> findAllOrdersByIdComanda(int idComanda) {
40         return StreamSupport.stream(ordineRepository.
41             findOrdineEntitiesByIdComanda(idComanda).spliterator(), false).
42             collect(Collectors.toList());
43     }
44
45     @Override
46     public void deleteOrder(int id) {
47         ordineRepository.deleteById(id);
48     }
49 }
```

Codice 2.25: 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 2.24 a pagina 55) 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[10] 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 2.26) in questo modo:

```

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

Codice 2.26: Aggiornamento del file pom.xml per la dipendenza di H2

Poi si aggiorna il file application.properties (Codice 2.27) 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=MySQL;DATABASE_TO_LOWER
   =TRUE
6 spring.datasource.username=user
7 spring.datasource.password=user
8 spring.datasource.driver-class-name=org.h2.Driver

```

Codice 2.27: 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 2.28):

```

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

```

```

28     public void testMultipleSaveOrderAndFindByIdComanda(){
29         OrdineEntity ordineEntityA = TestDataUtil.createOrdineEntity()
30         ;
31         OrdineEntity savedOrdineEntityA = dataPort.saveOrder(
32             ordineEntityA);
33         OrdineEntity ordineEntityB = TestDataUtil.createOrdineEntityB()
34         ;
35         OrdineEntity savedOrdineEntityB = dataPort.saveOrder(
36             ordineEntityB);
37         OrdineEntity ordineEntityC = TestDataUtil.createOrdineEntityC()
38         ;
39         OrdineEntity savedOrdineEntityC = dataPort.saveOrder(
40             ordineEntityC);

41         Iterable<OrdineEntity> result = dataPort.
42             findAllOrdersByIdComanda(ordineEntityA.getIdComanda());
43         assertThat(result).hasSize(2).containsExactly(
44             savedOrdineEntityA, savedOrdineEntityB);
45     }

46     @Test
47     public void testOrderPartialUpdate() {
48         OrdineEntity ordineEntity = TestDataUtil.createOrdineEntityA();
49         OrdineEntity savedOrdineEntity = dataPort.saveOrder(
50             ordineEntity);
51         savedOrdineEntity.setStato(0);
52         savedOrdineEntity.setUrgenzaCliente(1);
53         dataPort.saveOrder(savedOrdineEntity);
54         Optional<OrdineEntity> result = dataPort.getOrderById(
55             savedOrdineEntity.getId());
56         assertThat(result).isPresent();
57         assertThat(result.get()).isEqualTo(savedOrdineEntity);
58     }

59     @Test
60     public void testDeleteOrder() {
61         OrdineEntity ordineEntityA = TestDataUtil.createOrdineEntityA()
62         ;
63         OrdineEntity savedOrdineEntityA = dataPort.saveOrder(
64             ordineEntityA);
65         dataPort.deleteOrder(savedOrdineEntityA.getId());
66         Optional<OrdineEntity> result = dataPort.getOrderById(
67             savedOrdineEntityA.getId());
68         assertThat(result).isEmpty();
69     }

```

60 }

Codice 2.28: 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 (2.25 a pagina 56) 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.

2.6.6 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 [8].

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 2.29) 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   if: github.ref != 'refs/heads/main'
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'
```

```

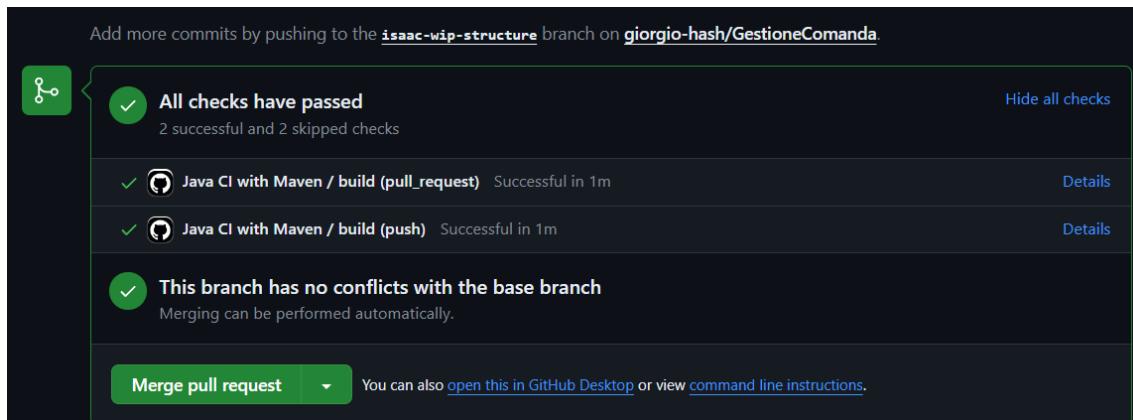
21   cache: maven
22     - name: Run the Maven verify phase
23       run: mvn --batch-mode clean verify

```

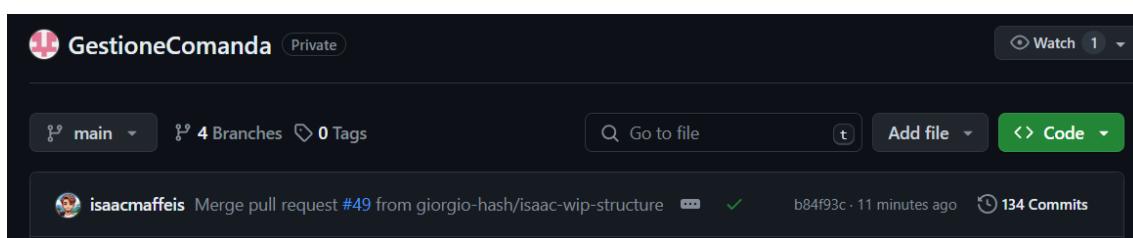
Codice 2.29: Creazione del file maven.yml

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 2.24:** CI merge a pull-request

Dalla Figura 2.24 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 2.25:** CI main check

Una volta completato il merge con successo è possibile notare una spunta verde nella repository principale del progetto (Figura 2.25) che ci informa che allo stato corrente il codice sul main ha passato tutti i test proposti.

2.6.7 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 [14], 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 2.30) 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 2.30: Aggiornamento dipendenze nel pom.xml per includere model-mapper

E successivamente è stato creato il bean di configurazione nella classe "MapperConfig.java" (Codice 2.31):

```

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 2.31: 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 2.23 a pagina 53), semplicemente creando una nuova classe Java (Codice 2.32) 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 2.32: 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 2.33):

```

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 2.33: 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 2.34):

```

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 2.34: 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 2.35) 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 2.35: 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.

2.6.8 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 [18].

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 2.26 nella pagina successiva):

zoom-in interface di gestione comanda

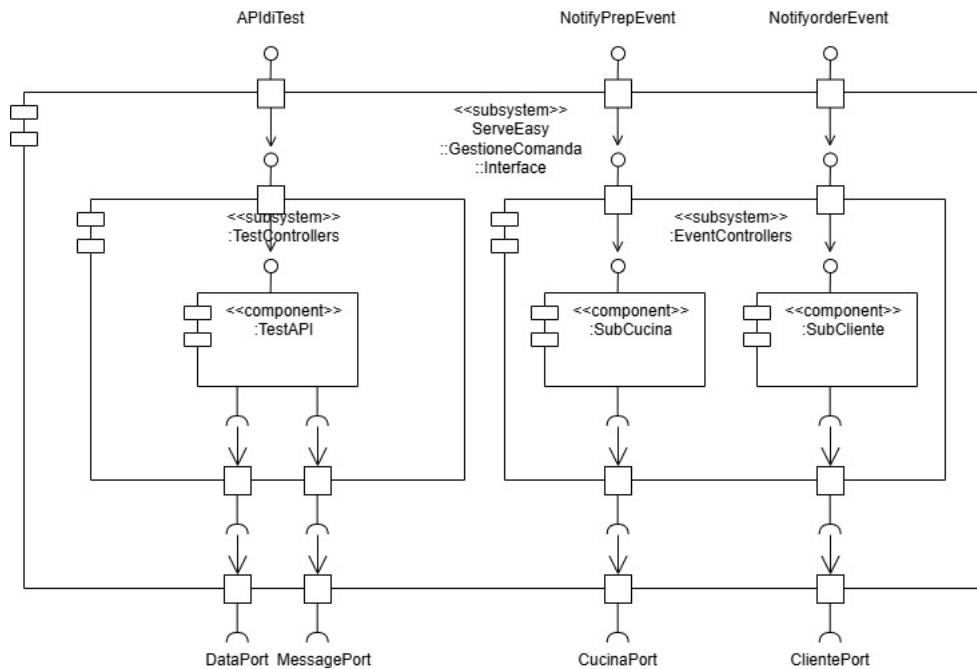


Figura 2.26: Component diagram - Gestione Comanda - Interface con Test

- 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 2.36) 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 2.36: 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 2.37) 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 2.37: 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 2.37 a pagina 68) andando a creare un controller REST di test (Codice 2.38):

```

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 2.38: 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 2.7 a pagina 78.

Step 5 - Testing: In questo passo ci siamo occupati di testare il REST Controller appena creato, lo abbiamo fatto utilizzando MockMVC [17], 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             OrdineEntity ordineEntity = TestDataUtil.createOrdineEntityB();
31             dataPort.saveOrder(ordineEntity);
32             mockMvc.perform(MockMvcRequestBuilders.get("/test/order/1").
33                 contentType(MediaType.APPLICATION_JSON)).andExpect(
34                 MockMvcResultMatchers.status().isOk());
35         }
36
37     @Test
38     public void testThatGetOrderReturnsHttpStatus404WhenNoOrderExists()
39         throws Exception {
40             mockMvc.perform(MockMvcRequestBuilders.get("/test/order/99").
41                 contentType(MediaType.APPLICATION_JSON)).andExpect(
42                 MockMvcResultMatchers.status().isNotFound());
43         }
44
45     @Test
46     public void testThatGetOrderReturnsOrderWhenOrderExist() throws
47         Exception {
48         OrdineEntity ordineEntity = TestDataUtil.createOrdineEntityB();
49         dataPort.saveOrder(ordineEntity);
50
51         mockMvc.perform(
52             MockMvcRequestBuilders.get("/test/order/1")
53                 .contentType(MediaType.APPLICATION_JSON)
54         ).andExpect(
55             MockMvcResultMatchers.jsonPath("$.id").value(1)
56         ).andExpect(
57             MockMvcResultMatchers.jsonPath("$.idComanda").value(
58                 ordineEntity.getIdComanda())
59         ).andExpect(
60             MockMvcResultMatchers.jsonPath("$.idPiatto").value(
61                 ordineEntity.getIdPiatto())
62         ).andExpect(
63             MockMvcResultMatchers.jsonPath("$.stato").value(
64                 ordineEntity.getStato())
65         ).andExpect(
66             MockMvcResultMatchers.jsonPath("$.urgenzaCliente").
67                 value(ordineEntity.getUrgenzaCliente())));
68     }
69     ...
70 }
```

Codice 2.39: 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 2.7 nella pagina successiva 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 2.27):

The screenshot shows the Postman interface with the following details:

- Left Sidebar (Collections):** Shows a tree structure of API collections:
 - Gestione Cliente
 - Gestione Comanda
 - Message Broker
 - POST Post to topic sendOrderEvent
 - GET Get from topic sendOrderEv...
 - POST Post to topic notifyOrderEvent
 - GET Get from topic notifyOrderEv...
 - POST Post to topic notifyPrepEvent
 - GET Get from topic notifyPepEv...
 - Database
 - POST Post order
 - GET Get order by id
 - PATCH Patch order
 - GET Get orders by IdComanda
 - DEL Delete order
 - Gestione Cucina
 - Test a gruppi
- Central Area (Request Details):**
 - Method:** PATCH
 - URL:** localhost:8080/test/order/1
 - Body:** JSON (Raw JSON input)


```
1 {
2     ...
3     "stato": 1,
4     ...
5     "urgenzaCliente": 1
6 }
```
- Bottom Area (Response Preview):**
 - Status Code: 200 OK
 - JSON Response (Pretty):


```
1 {
2     "id": 1,
3     "idComanda": 4,
4     "idPiatto": "PIA779",
5     "stato": 1,
6     "urgenzaCliente": 1,
7     "tordinazione": "2924-04-30 23:38:42.814"
```

Figura 2.27: 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.

2.7 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 2.6.8 a pagina 66, 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  
}
```


Iterazione 2

3.1 Algoritmo

3.1.1 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.

3.1.2 Organizzazione

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

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.

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;
- ogni postazione di lavoro può avere più cuochi (la presenza di più cuochi aumenta la velocità di preparazione della postazione), i cuochi possono spostarsi tra le postazioni;
- una postazione può essere vuota, esiste un massimo numero di cuochi per postazione; ogni postazione ha una coda di ordini da preparare:
 - soglia minima di ordini in coda per poter attivare la postazione;
 - soglia massima di ordini in coda (oltre la quale si può richiede un cuoco aggiuntivo oppure di rallentare aggiornando il parametro);
 - tempo massimo in cui gli ordini possono stare in coda di preparazione.
- In preparazione possono stare un numero di ordini pari al numero di cuochi;
- la somma degli ordini in coda di preparazione è sempre minore della lunghezza della coda di preparazione più piccola.

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, ogni postazione presenta un numero massimo di cuochi che possono lavorare contemporaneamente e può essere attivata solo con un numero minimo di ordini in coda (può essere vuota senza cuochi).

- 1 ingrediente principale;
- N cuochi ($N < M$ max cuochi per postazione);
- 1 struttura dati (coda);
- stato (vuota, regolare, intasata).

3.1.3 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.

Si rendono quindi necessarie due tipi di strutture dati:

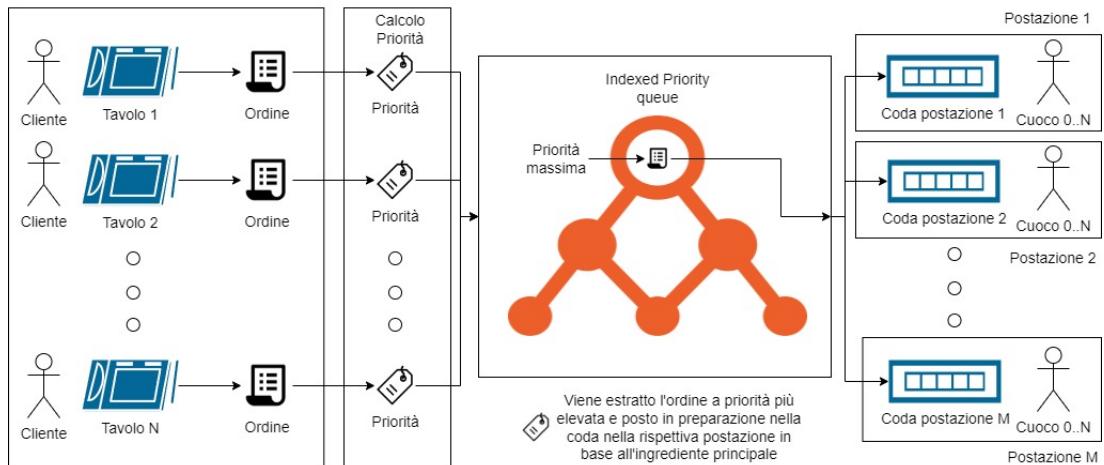


Figura 3.1: Strutture dati dell'algoritmo

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

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.

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.

3.1.4 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

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.

- condizione iniziale ogni ingrediente ha valore 0.5
- se la cucina è satura ridurre il valore (min 0)
- se la cucina è scarica aumentare il valore (max 1)

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

- normalizzazione:

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

con tp : tempo di preparazione,

tp_{\max} : tempo di preparazione massimo,

tp_{\min} : tempo di preparazione minimo;

- considerare $x2 = tp_{\text{norm}}$ per prioritizzare ordini più lunghi,
oppure $x2 = 1 - tp_{\text{norm}}$ 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 specificare di non avere fretta o non dire nulla e tenere un valore di urgenza di default

- 1 se il cliente ha espresso urgenza;
- 0 se ha espresso di ritardare o fare con calma;
- valore neutro standard 0.5.

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.

- il primo ordine effettuato ha priorità maggiore, mentre i successivi hanno priorità decrescente;

- normalizzazione:

$$x4 = \frac{noe - 1}{\max_{noe} - 1}$$

con noe : numero ordini effettuati,

\max_{noe} : massimo numero ordini effettuabili;

- considerare un valore massimo di ordini (es.5 gli ordini dopo il quinto sono comunque consentiti e prenderanno la stessa priorità del 5° ordine).

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

- normalizzazione:

$$x5 = \frac{\text{tempo in attesa}}{\text{tempo max in attesa}}$$

- considerare un valore massimo di tempo in attesa consentito, in prossimità del quale si ha la priorità più elevata.

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.

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. Per questo motivo è richiesta una certa flessibilità in modo da consentire l'aggiornamento dei pesi dei parametri in modo dinamico.

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.

$$y = p_1 * x_1 + p_2 * x_2 + p_3 * x_3 + p_4 * x_4 + p_5 * x_5$$

3.1.5 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.

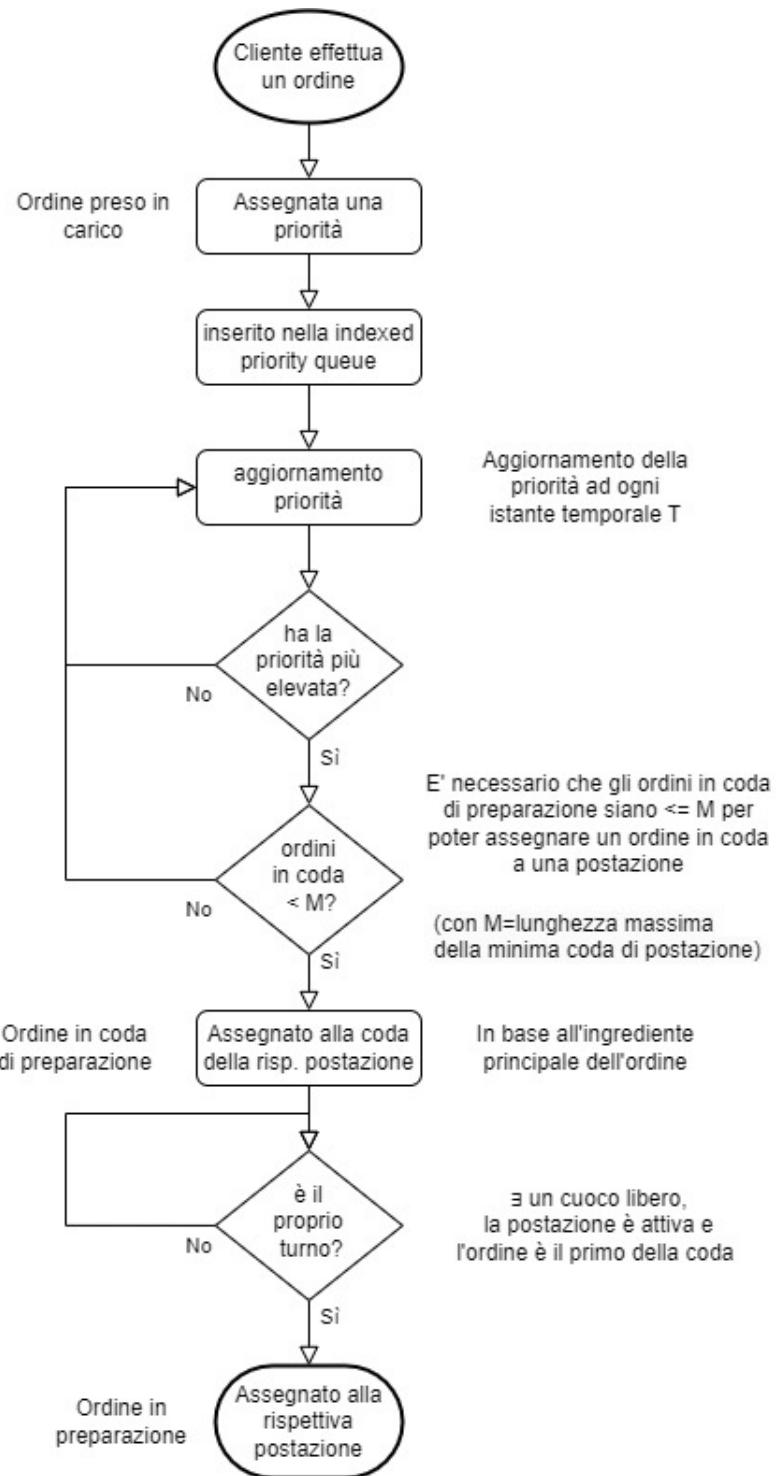


Figura 3.2: Diagramma di flusso

Bibliografia

- [1] Baeldung. *Guide on Loading Initial Data with Spring Boot*. URL: <https://www.baeldung.com/spring-boot-data-sql-and-schema-sql> (visitato il 03/05/2024).
- [2] Baeldung. *Interface Driven Controllers*. URL: <https://www.baeldung.com/spring-interface-driven-controllers> (visitato il 02/05/2024).
- [3] Baeldung. *MockMvc Integration Tests*. URL: <https://www.baeldung.com/integration-testing-in-spring> (visitato il 12/04/2024).
- [4] Baeldung. *Object Mapper*. URL: <https://www.baeldung.com/jackson-object-mapper-tutorial> (visitato il 10/04/2024).
- [5] Baeldung. *Spring Data JPA @Query*. URL: <https://www.baeldung.com/spring-data-jpa-query> (visitato il 02/05/2024).
- [6] Baeldung. *Testing Kafka*. URL: <https://www.baeldung.com/spring-boot-kafka-testing> (visitato il 02/05/2024).
- [7] Happy Coders. *Filosofia Esagonale e Microservizi*. URL: <https://www.happycoders.eu/software-craftsmanship/hexagonal-architecture/> (visitato il 02/05/2024).
- [8] Continuous Integration with GitHub Actions. URL: <https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration> (visitato il 22/04/2024).
- [9] 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 02/05/2024).
- [10] H2 DB. URL: https://www.h2database.com/html/features.html#in-memory_databases (visitato il 02/05/2024).
- [11] Arho Huttunen. *Testing Jackson*. URL: <https://www.arhohuttunen.com/spring-boot-json-test/> (visitato il 29/04/2024).
- [12] 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 06/05/2024).

- [13] 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 22/04/2024).
- [14] ModelMapper. *ModelMapper - Getting Started*. URL: <https://modelmapper.org/getting-started/> (visitato il 01/05/2024).
- [15] Obsidian Dynamics. *Kafdrop*. <https://github.com/obsidiandynamics/kafdrop>. (Visitato il 25/04/2024).
- [16] Sandro Pinna. *Continuous Delivery & Continuous Deployment: dal rilascio manuale a docker*. URL: <https://newsroom.spindox.it/continuous-delivery-continuous-deployment-docker/> (visitato il 06/05/2024).
- [17] Inc. Pivotal Software. *Spring MVC Test Framework*. 2024. URL: <https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html> (visitato il 05/05/2024).
- [18] Spring. *Building a RESTful Web Service*. 2024. URL: <https://spring.io/guides/gs/rest-service> (visitato il 14/04/2024).
- [19] Spring Boot. URL: <https://spring.io/projects/spring-boot> (visitato il 02/05/2024).
- [20] Spring Data JPA Reference Documentation. URL: <https://docs.spring.io/spring-data/jpa/reference/> (visitato il 04/05/2024).
- [21] Spring Kafka. URL: <https://docs.spring.io/spring-kafka/reference/index.html> (visitato il 02/05/2024).
- [22] Web Layer Test. URL: <https://spring.io/guides/gs/testing-web> (visitato il 15/04/2024).