

# PROGETTAZIONE ARCHITETTURA SOFTWARE

---

PROGETTAZIONE E SVILUPPO DI SISTEMI  
SOFTWARE

M63/1069 Pisano Francescantonio  
00117822 Poziello Alessandro  
M63/1007 Puccinelli Gennaro  
M63/1078 Ricciardi Armando



UNIVERSITÀ DEGLI STUDI DI NAPOLI  
**FEDERICO II**

## Sommario

Design .....	2
Diagramma a componenti .....	2
Diagramma di deploy sui nodi .....	3
Stile architetturale utilizzato .....	4
Client-Server .....	4
Pattern Utilizzati .....	4
Model-View-Controller (MVC) .....	4
Data Transfer Object (DTO) .....	6
Repository .....	7
Architettura Software.....	8
Framework Spring MVC.....	8
Architettura 2Service (Vista Cliente-Autenticato).....	9
Ulteriori viste architetturali.....	10
Architettura 2Service (Vista Proprietario).....	10
Architettura 2Service (Vista Cameriere).....	11
Sequence Diagram raffinati.....	12
SD raffinato Crea Conto.....	12
SD raffinato Effettua Ordine.....	13
SD raffinato Chiedi Conto .....	13
SD raffinato Salda Conto Attivo.....	14
Schema del database: Diagramma ER.....	15
Implementazioni future.....	16

# Design

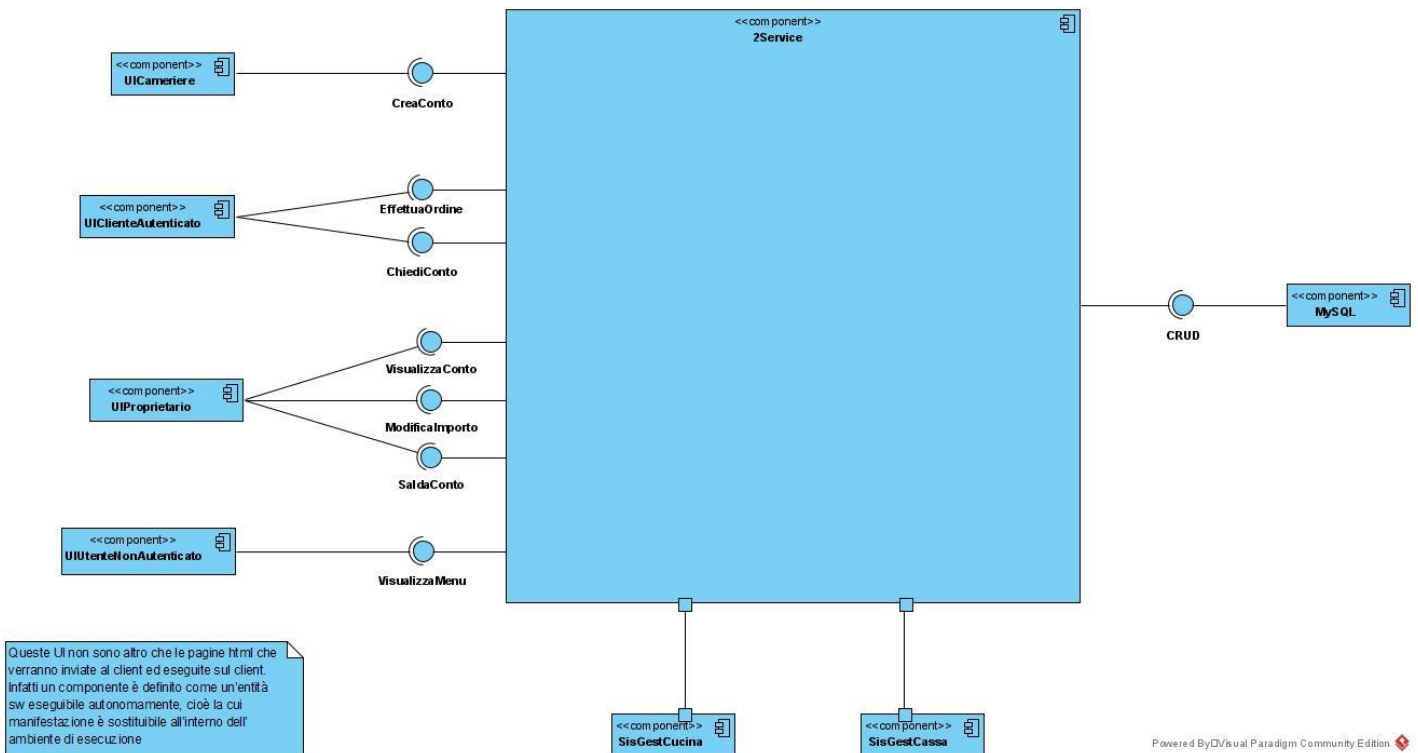
## Diagramma a componenti

Il sistema 2Service si presenta come un unico componente, ossia un unico modulo software rilasciabile ed eseguibile autonomamente.

Ad esso possono essere aggiunti ulteriori componenti se compatibili, cioè se offrono e richiedono le stesse interfacce.

Nel dettaglio si ha:

- Un componente che permette di gestire la persistenza dei dati del sistema (Database MySQL Server)
- Due componenti, uno per la gestione della cucina ed un altro per la gestione della cassa, corrispondenti a sistemi software tipici dei locali di ristorazione
- Un componente per ogni tipologia di utente con le relative interfacce per interagire con 2Service. Questi ultimi corrispondono a pagine HTML che verranno inviate al Client ed eseguite dal browser engine.



## Diagramma di deploy sui nodi

Il sistema 2Service può essere "*deployato*" o distribuito su un unico nodo di elaborazione per funzionare.

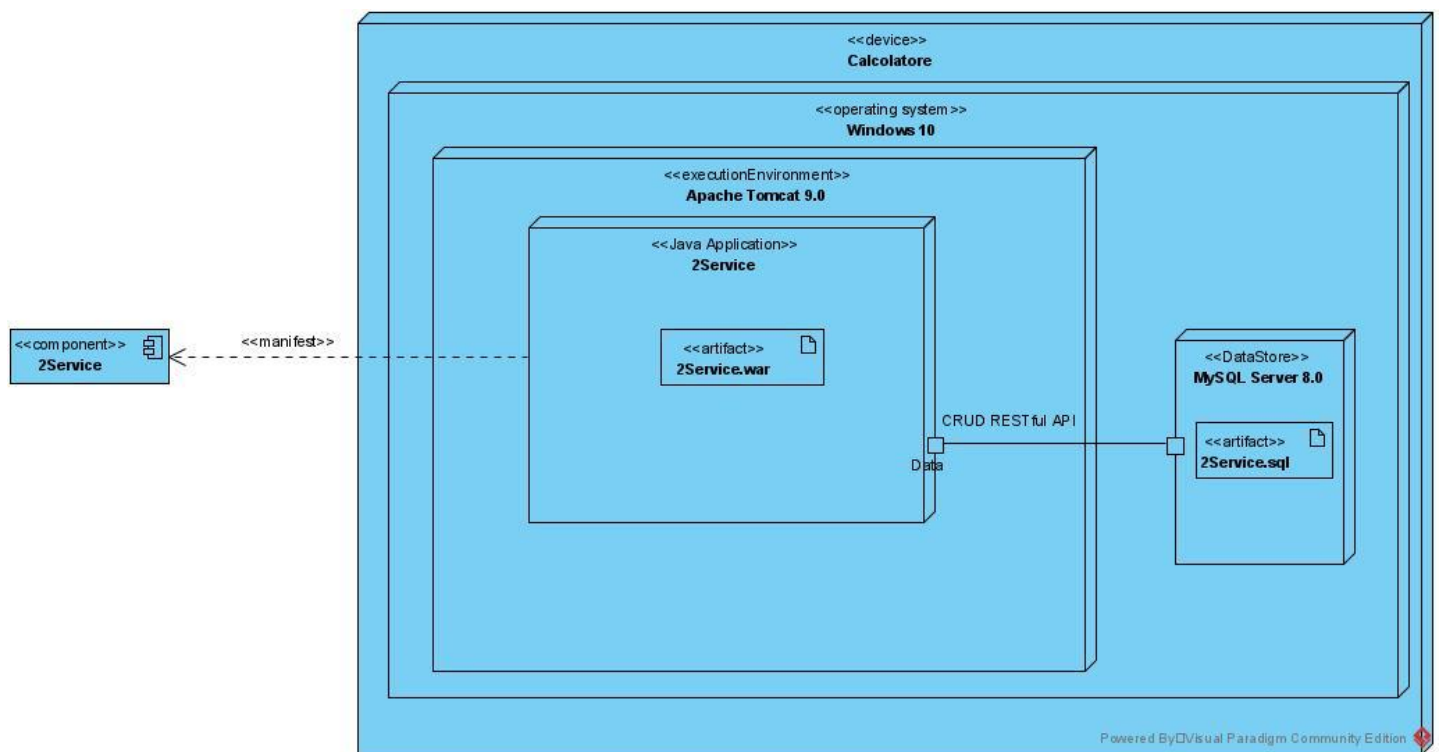
Per consentire il funzionamento sul calcolatore (che funge da server) si dovranno installare e configurare i seguenti software:

- Un **Web Server Apache Tomcat** versione 9.0
- un database **MySQL Server** 8.0.

In particolare, in *Tomcat* andrà installato il file "*war*" (*Web application ARchive*), mentre in *MySQL* il file "*sql*" (*Structured Query Language*) che contiene lo schema del database ed i dati necessari alla corretta esecuzione del sistema.

La comunicazione tra i due software avviene tramite **MySQL JDBC driver**, che fornisce l'accesso al database e le operazioni CRUD di base (Create, Read, Update, Delete) e quindi permette all'applicazione di essere RESTful.

Infine, la manifestazione dell'artifact "2Service.war" viene rappresentata dal componente 2Service.

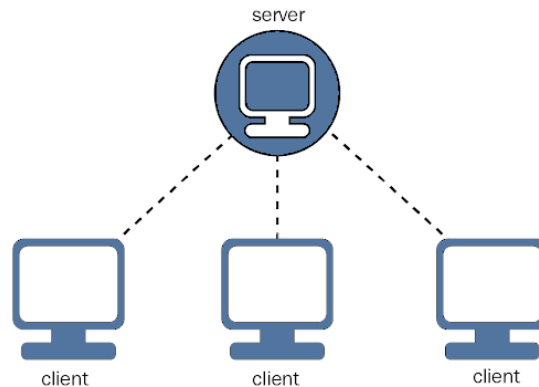


## Stile architetturale utilizzato

### Client-Server

Ad un alto livello di dettaglio l'architettura del sistema 2Service, segue uno stile Client-Server distribuito, in cui sia il client che il server sono in esecuzione su nodi differenti.

Nel dettaglio si ha un *"thin"* client che richiede servizi al *"fat"* server.



### Pattern utilizzati

Di seguito verranno discussi i pattern (architetturali e di design) usati per l'applicazione.

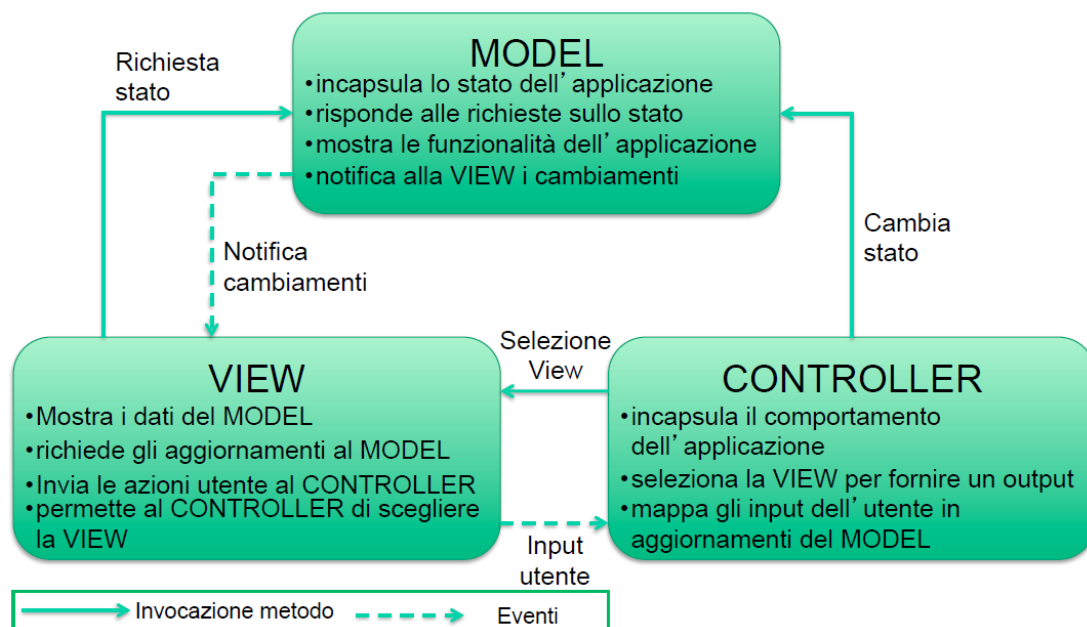
### Model-View-Controller (MVC)

Per la realizzazione del server è stato scelto il pattern architetturale Model-View-Controller.

La scelta è ricaduta su di esso vista la necessità di mantenere separata la logica di visualizzazione dalle logiche di business e di accesso ai dati.

In particolare, ricordiamo i principali ruoli di questi componenti:

- Il **Model** gestisce i dati e le regole di business per poter interagire con essi. Fornisce alla view ed al controller le funzionalità per l'accesso e l'aggiornamento.
- La **View** gestisce la logica di presentazione dei dati e permette di implementare diverse viste su di essi
- Il **Controller** trasforma gli input utente della view in azioni da eseguire sul model e seleziona le schermate da mostrare tramite la view.

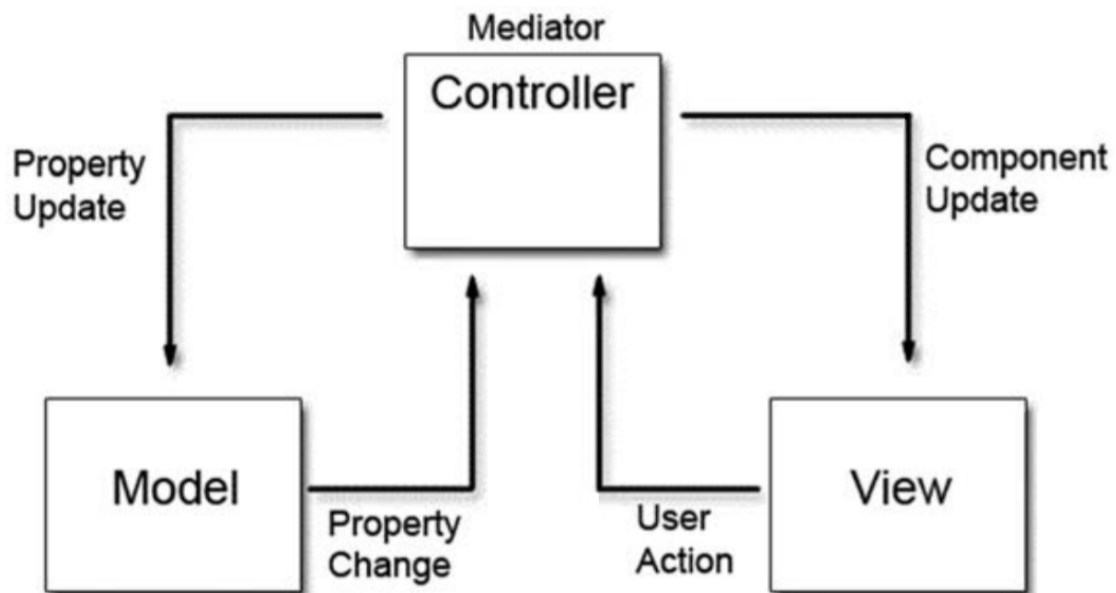


A questa tipologia di MVC detta di tipo *push*, si affianca una seconda tipologia di tipo *pull* tipicamente utilizzata nelle soluzioni distribuite.

Nel Model View Controller di tipo *pull* la sequenza di azioni tra i tre componenti è leggermente diversa.

In particolare:

- La view riceve un input utente e lo invia al controller
- Il controller seleziona l'azione corrispondente all'input, che dovrà essere eseguita sul model
- Il model esegue l'azione, si aggiorna e restituisce al controller il nuovo stato
- Il controller è a sua volta delegato di restituire alla view il nuovo stato del model.



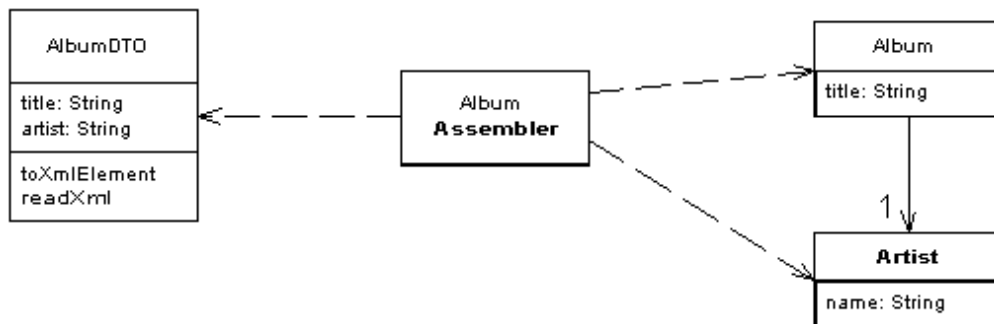
## Data Transfer Object (DTO)

Il DTO è un **Design Pattern**, definito anche come *Distribution Pattern* da Martin Fowler [1], utilizzato principalmente in caso di comunicazione tra nodi di elaborazioni distribuiti.

Questo pattern nasce per risolvere il problema relativo alla condivisione di un oggetto tra due nodi, ad esempio un Client ed un Server.

Infatti, volendo seguire i principi dell'Object-Orientation un oggetto per funzionare correttamente si basa su altri oggetti; se però una copia del primo oggetto viene inviata su un nodo remoto, tale oggetto non avrà a disposizione gli altri oggetti di cui ha bisogno per funzionare.

Ed è qui che interviene il DTO con cui non ci si limita ad inviare una copia dell'oggetto, bensì un nuovo oggetto che ha tutte le informazioni al suo interno per poter funzionare in modo ottimale nel nodo remoto.



Nel caso dell'applicazione "2Service" si è deciso di utilizzare questo pattern per permettere di trasferire liste di oggetti tra il Controller, che è sul server, ed una View, che si trova sul Client, in maniera automatica utilizzando Spring MVC e Thymeleaf.

Il DTO realizza ciò tramite una conversione delle informazioni che provengono dal client, tramite input utente, ed arrivano al Controller.

Quindi anziché inviare un oggetto Ordine con molteplici attributi si utilizza un oggetto OrdineCreationDTO; esso invierà solamente le informazioni necessarie al client e riotterrà in maniera automatica le informazioni richieste sotto forma di oggetto DTO.

I vantaggi offerti da questo pattern sono:

- Consente di ridurre il numero di chiamate remote tra due nodi di elaborazione;
- Permette di nascondere i dettagli implementativi degli oggetti di dominio dell'applicazione.

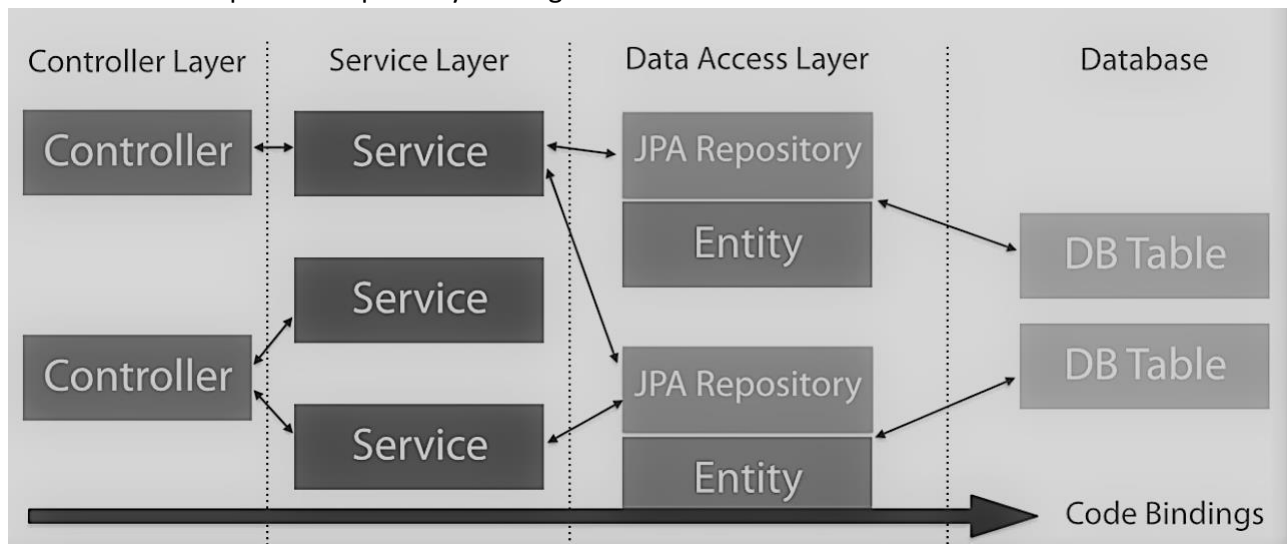
Il principale svantaggio nell'utilizzo del DTO è quello di avere una replica dell'oggetto, o di una gerarchia di oggetti, che non riguarda la logica di business. Ed inoltre, necessita dell'implementazione di specifiche funzioni volte alla conversione degli oggetti DTO nei corrispettivi oggetti del dominio.

## Repository

Il pattern Repository è un design pattern, ormai consolidato per gestire la persistenza dei dati. Esso nasce per garantire una separazione tra la logica di business e la logica di persistenza. Infatti, il package Repository contiene tutto il codice relativo alla persistenza, ma nessuna logica di business. Rende più facile la scrittura e la leggibilità del codice, favorisce il riuso, e permette di concentrarsi sull'implementazione dei requisiti anziché sull'interazione con il database.

Precisamente fornisce metodi per inserire, aggiornare e rimuovere le entità nei database ed anche metodi che istanziano ed eseguono query specifiche sul database.

L'architettura del pattern Repository è la seguente:



- Il **Controller Layer** contiene la logica di business tramite cui coordina l'esecuzione di una certa funzionalità o servizio del sistema;
- il **Service Layer** mette a disposizione operazioni di base sulle entità del sistema, come ad esempio quelle di creazione, aggiornamento, cancellazione o ricerca;
- il **Data Access Layer**, composto dalle interfacce JPA più le classi di *entity* con annotazioni Hibernate. Questo layer definisce le entità del sistema e ne permette il "*mapping*" con il database relazionale e le sue tabelle.



## Architettura Software

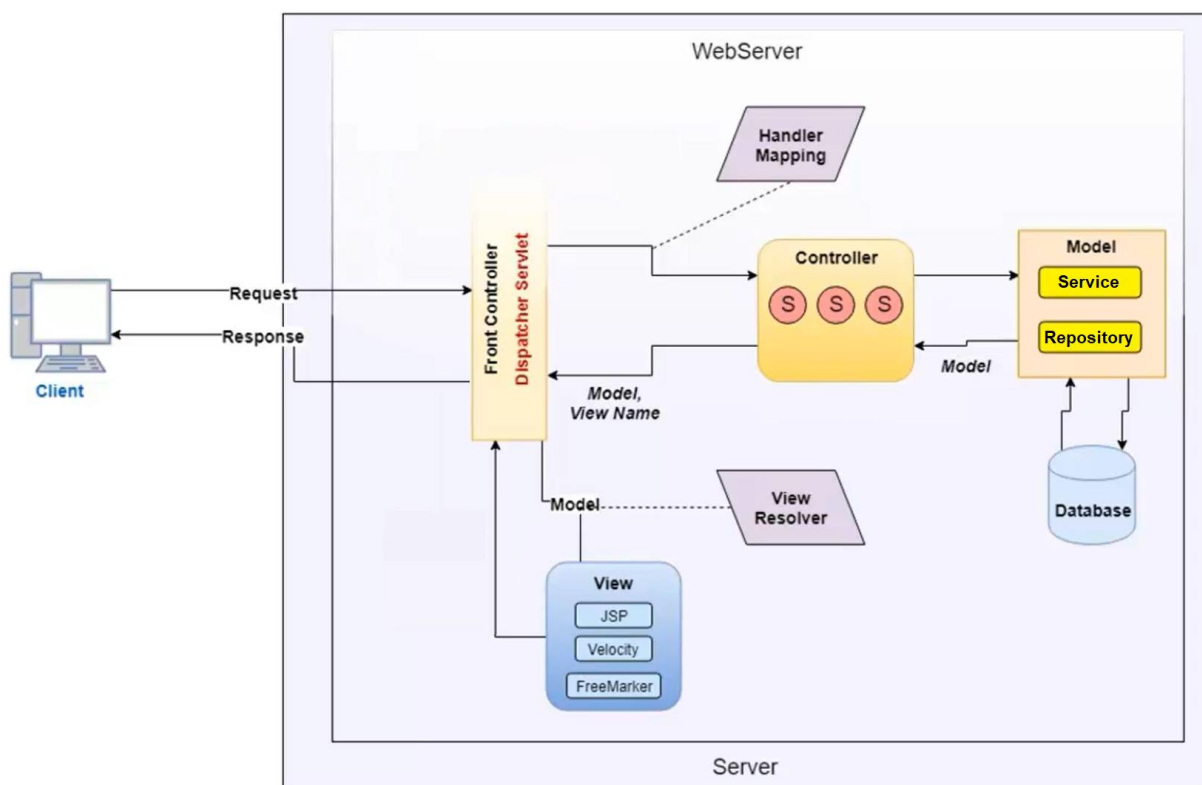
L'architettura software dell'applicativo 2Service usa lo stile architetturale Client-Server e il pattern architetturale MVC per la progettazione del server.

Durante lo sviluppo, e vista la natura distribuita dell'applicazione, si è utilizzato il Framework Spring MVC che implementa un MVC di tipo pull.

## Framework Spring MVC

In 2Service gli input utente corrispondono a richieste HTTP verso il server.

Queste richieste sono catturate dal *Front Controller*, il quale le distribuisce ai vari controller del sistema tramite la configurazione del modulo di *Handler Mapping*. In questo modo il *Front Controller* svolge una funzione di *Dispatcher Servlet*.



A questo punto i vari controller si occupano della logica principale dell'applicazione e comunicano con il model per poter interagire con i dati.

Infatti, il Model si dovrà occupare della persistenza dei dati e dei servizi di base ad essi relativi per poterli utilizzare e manipolare. Quindi la struttura interna del Model presenterà:

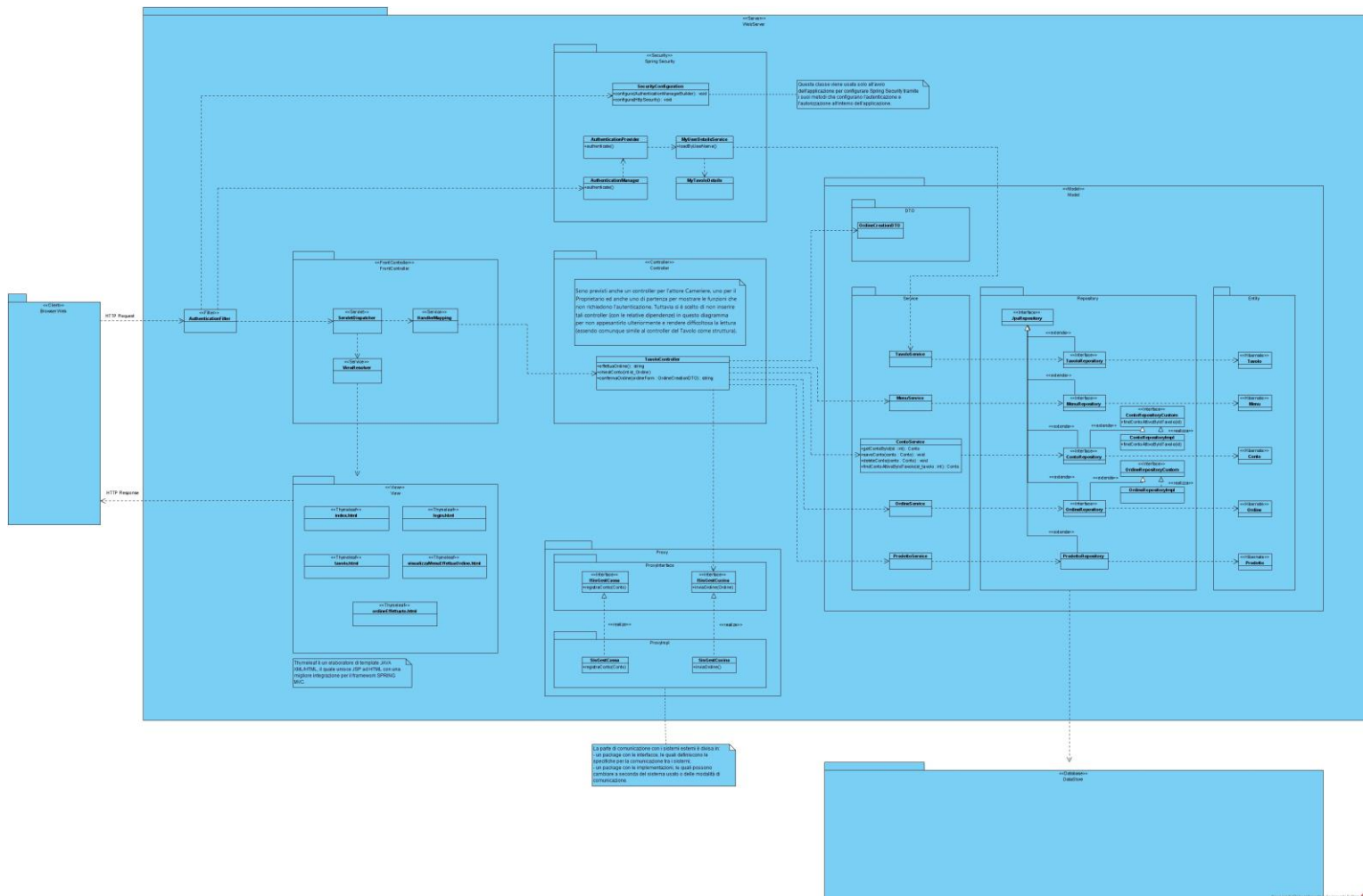
- Un package Service contenente le operazioni di CRUD e ricerca per ogni entità del modello
- Un package Repository contenente le interfacce JPA (Java Persistence API) per il collegamento al database.

Poiché Spring MVC implementa un MVC di tipo pull il Model non comunica direttamente con la View, ma gli oggetti del Model dovranno essere restituiti al Controller per poi passare al FrontController che li inoltrerà alla parte di presentazione (View).

Ogni view verrà identificata con un nome specifico grazie alla configurazione automatica del modulo ViewResolver. La vista viene infine inviata al Cliente tramite una servlet che gestisce le HTTPResponse.

## Architettura 2Service (Vista Cliente-Autenticato)

L'applicazione 2Service presenta un diagramma architetturale in parte simile a quello descritto nella sezione precedente, poiché sviluppato con il Framework Spring MVC, ma con delle piccole variazioni ed aggiunte.



Si noti che per facilitare la leggibilità si è voluto rappresentare solo le classi relative al caso d'uso "Effettua Ordine", che comunque permette di sviscerare tutte le parti dell'architettura.

Al fine di migliorare la presentazione delle viste offerte dalla view si è optato per l'uso di **Thymeleaf**. Esso è un elaboratore di template JAVA XML/HTML, il quale unisce JSP ad HTML con una migliore integrazione per il framework Spring MVC.

All'interno del server il Model presenta dei package aggiuntivi che sono:

- Un package **Entity** contenente le entità del dominio di interesse, tali entità sono annotate con annotazioni *Hibernate*;
- Un package **DTO** (Data Transfer Object) contenente le classi necessarie per un trasferimento dati tra Client e Server meno oneroso.

Esternamente alla struttura Client-Server troviamo il componente logico database realizzato tramite MySQL Server. Si è scelto di porre il database come componente esterno al sistema per garantirne maggiore modularità e scalabilità.

Infatti, realizzando l'architettura in maniera modulare, in futuro è possibile decidere di distribuirla anche diversamente (es. distribuire il database su un nodo diverso da dove si trova il server oppure collegare più server su cui il sistema si appoggia).

Questa opzione sarebbe impossibile se avessimo considerato un database interno all'applicazione. Dunque, si può vedere l'architettura di 2Service come un Client-Server a 3 livelli dove il client richiede i servizi al server il quale a sua volta richiede i dati al database per poterli poi restituire al client

Per garantire la comunicazione con i sistemi esterni si è utilizzato un package proxy che presenta all'interno un'ulteriore suddivisione in package relativi all'interfacciamento e package relativi all'implementazione. Nel dettaglio i sistemi che usufruiranno di questo package sono:

- Sistema di gestione cassa
- Sistema di gestione cucina

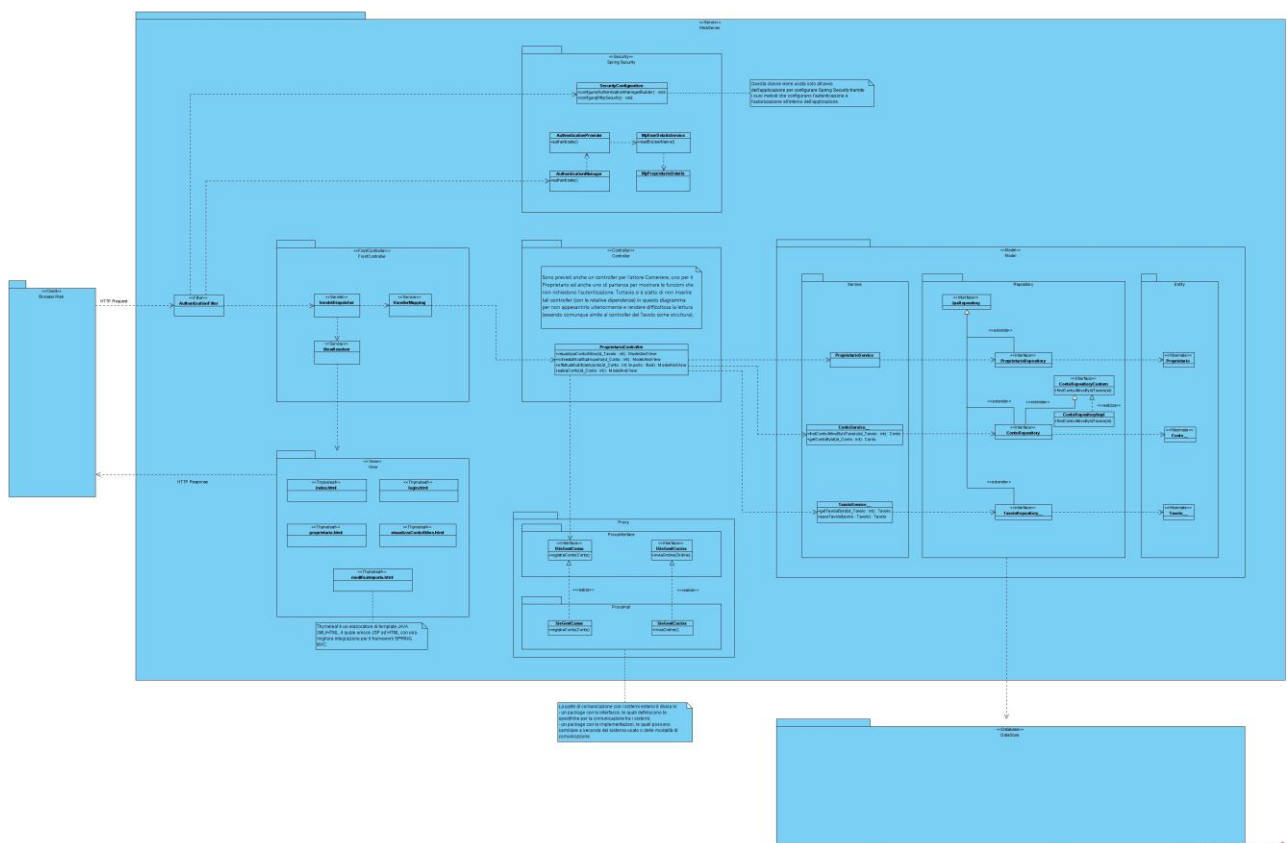
Questa scelta è stata fatta per favori la riusabilità del sistema, fornendo la possibilità di cambiare la specifica implementazione del servizio esterno senza modificare le interfacce.

Al fine di gestire la sicurezza all'interno dell'applicazione e garantire autenticazione ed autorizzazione si è utilizzato il framework Spring Security. Per cui, all'interno del diagramma di architettura vi è un Authentication Filter che intercetta tutte le richieste HTTP e valuta se può essere soddisfatta dal particolare utente. Nel caso in cui il cliente non sia autenticato, rimanda ad un form di login, mentre se non è autorizzato il cliente visualizzerà un errore [aggiungere screen].

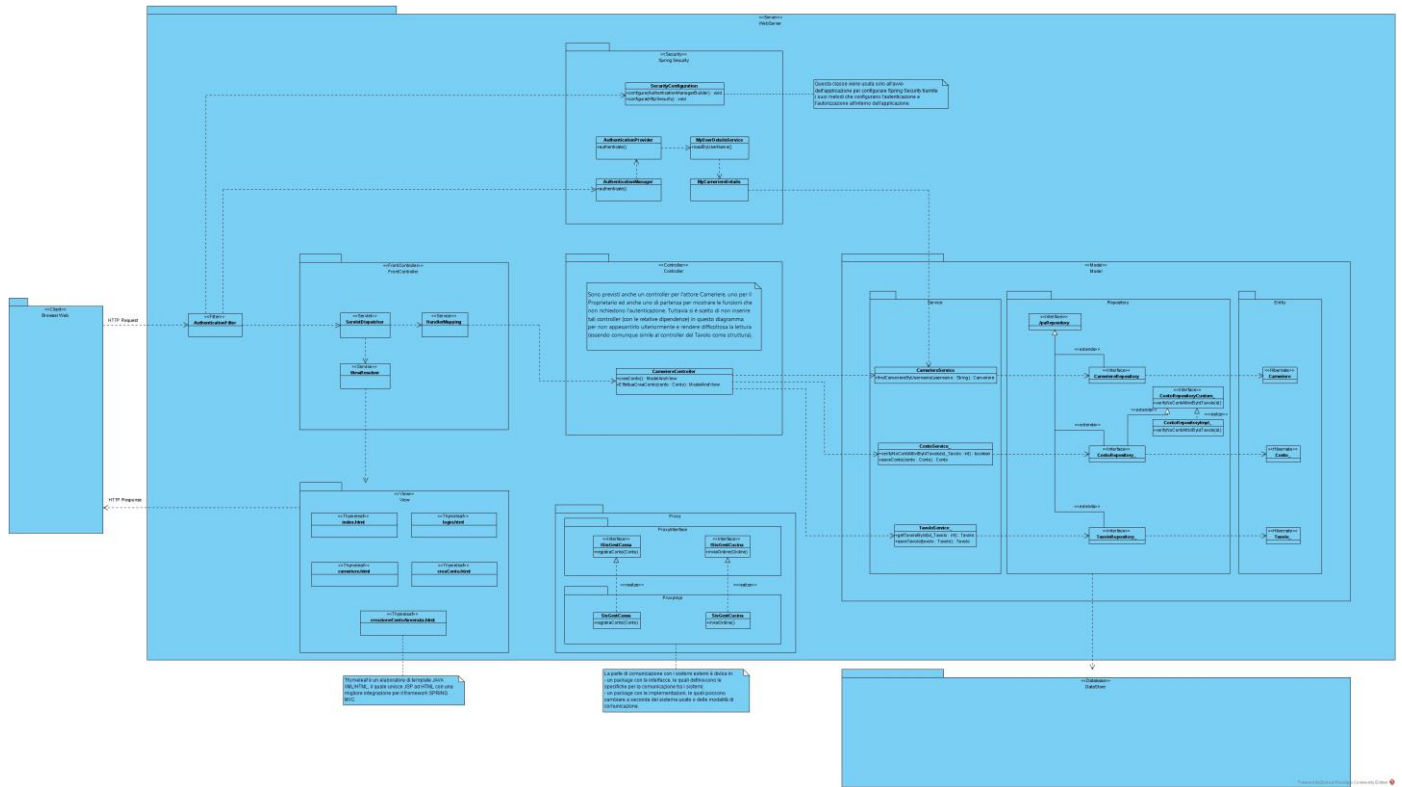
Oltre all'authentication filter, nell'architettura vi è un package di security che permette di configurare il framework all'avvio dell'applicazione in modo da gestire le diverse tipologie di utente e, durante l'esecuzione, gli accessi al sistema.

## Ulteriori viste architetturali

### Architettura 2Service (Vista Proprietario)

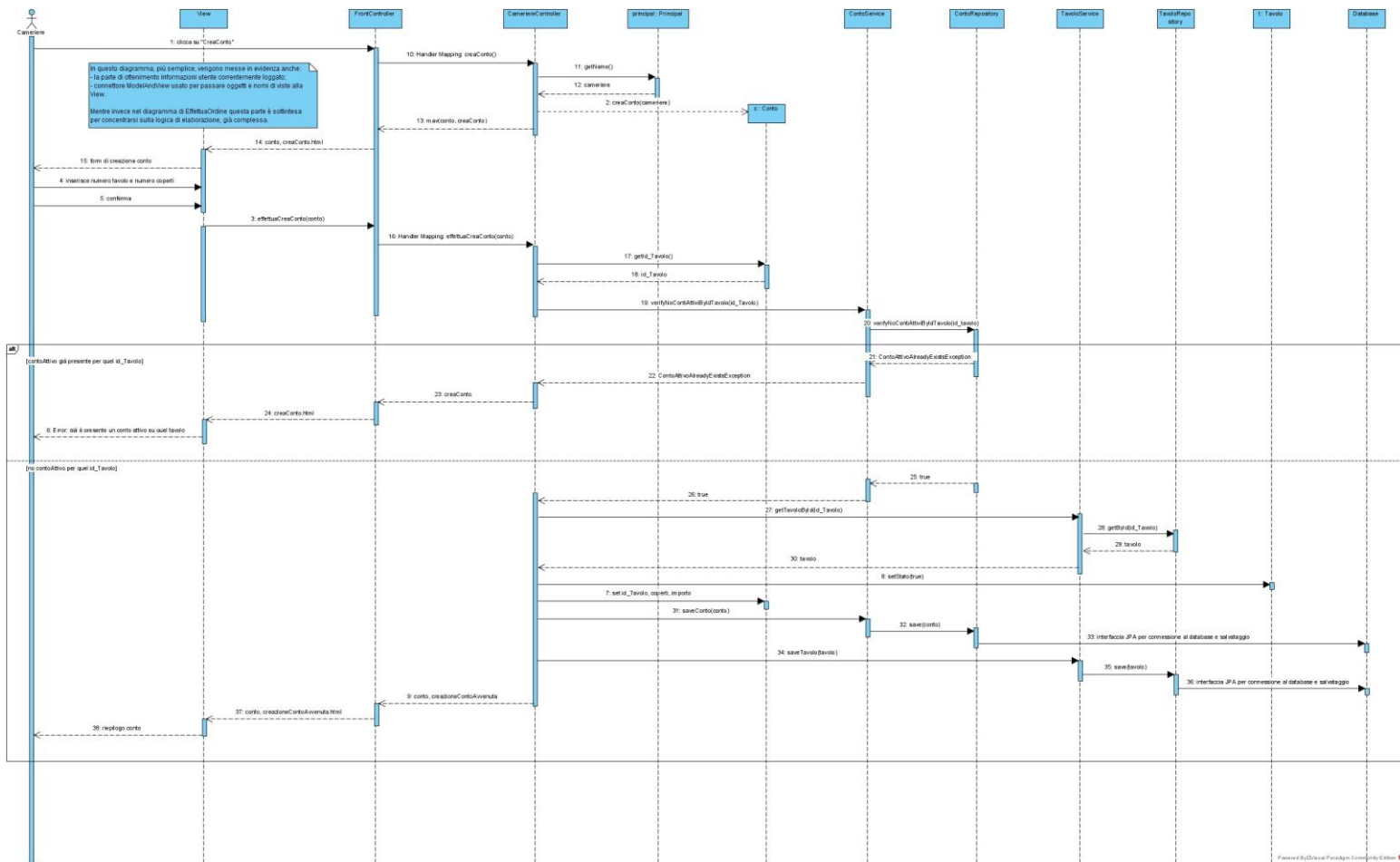


## Architettura 2Service (Vista Cameriere)

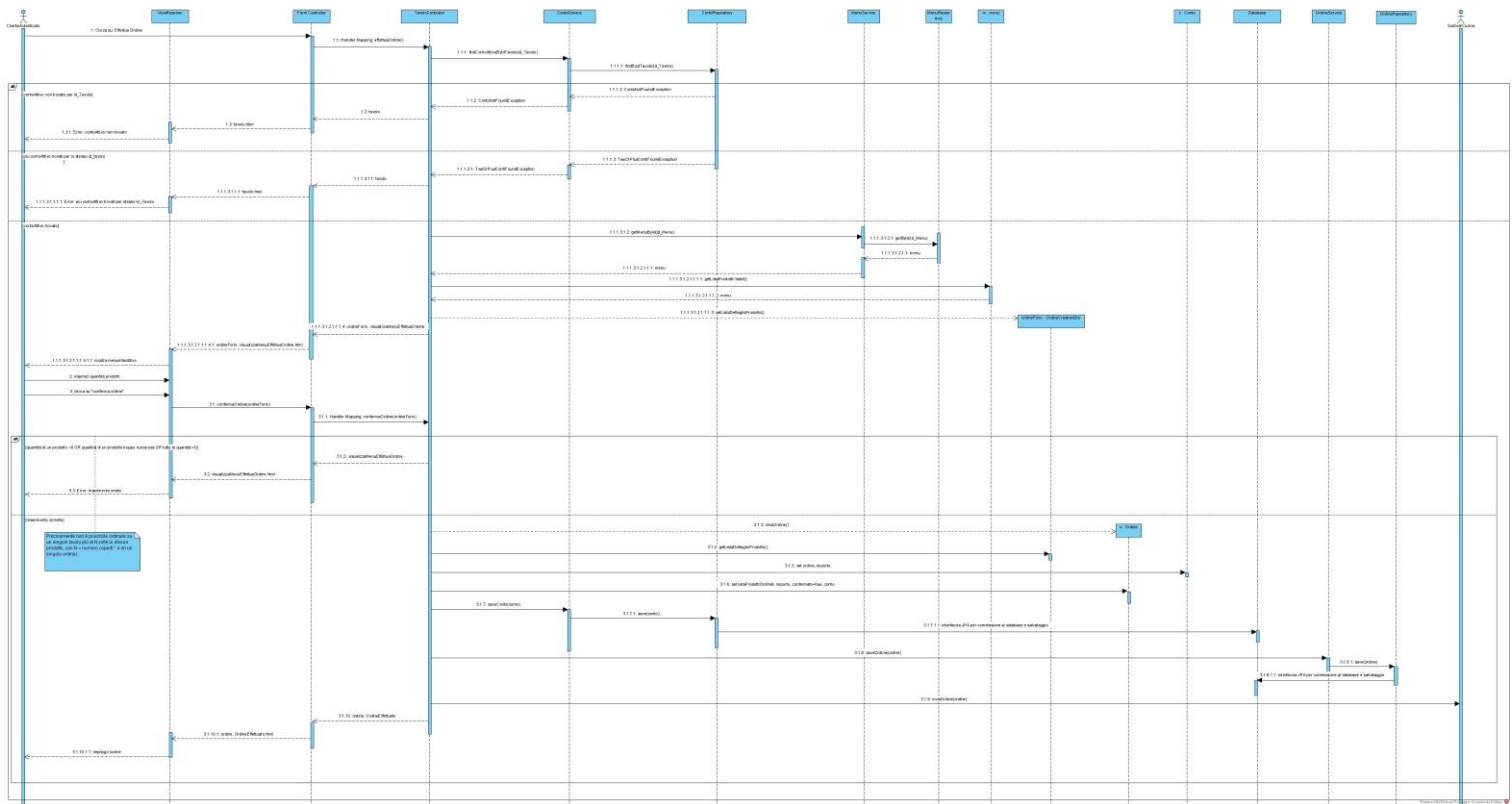


# Sequence Diagram raffinati

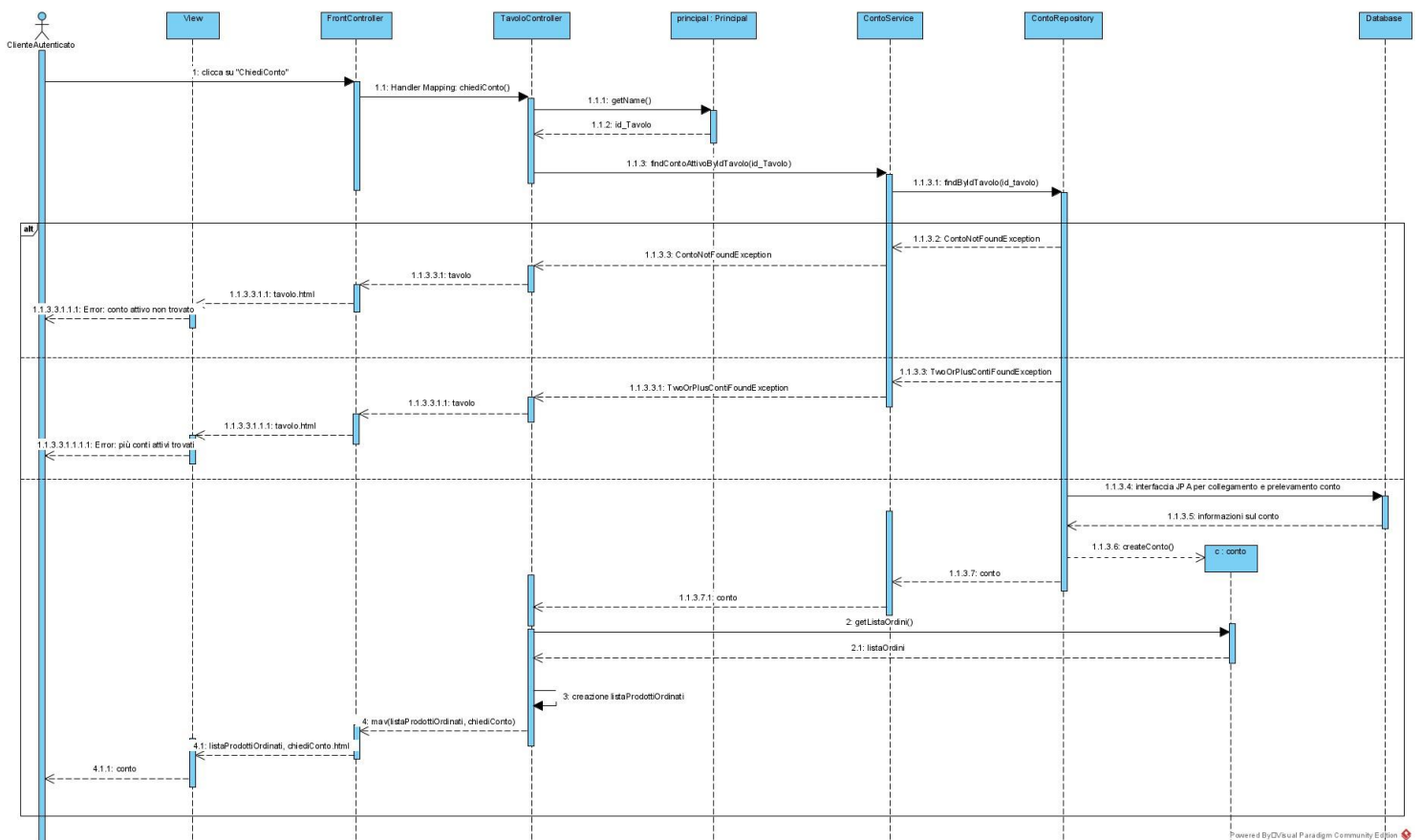
## SD raffinato Crea Conto



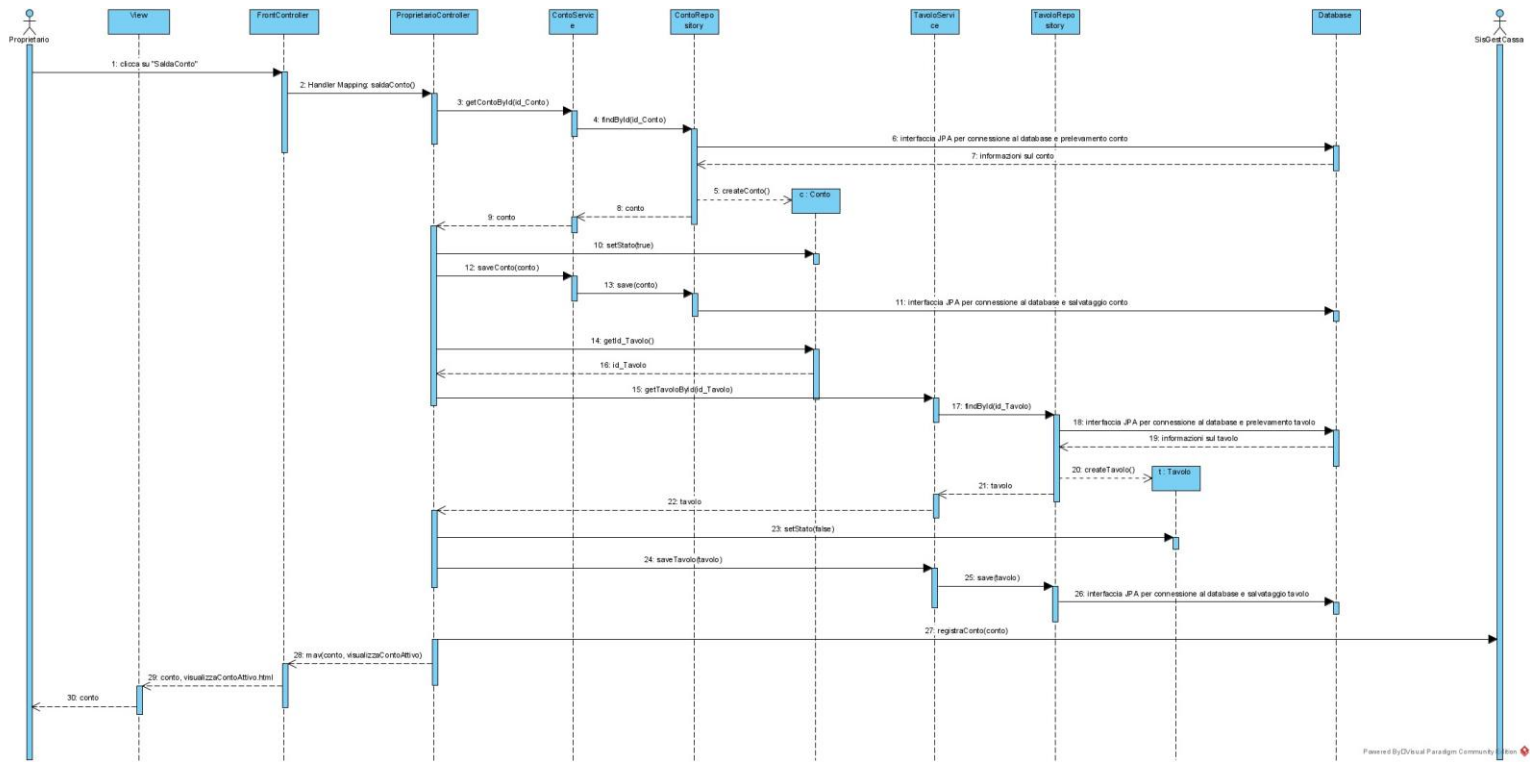
## SD raffinato Effettua Ordine



## SD raffinato Chiedi Conto



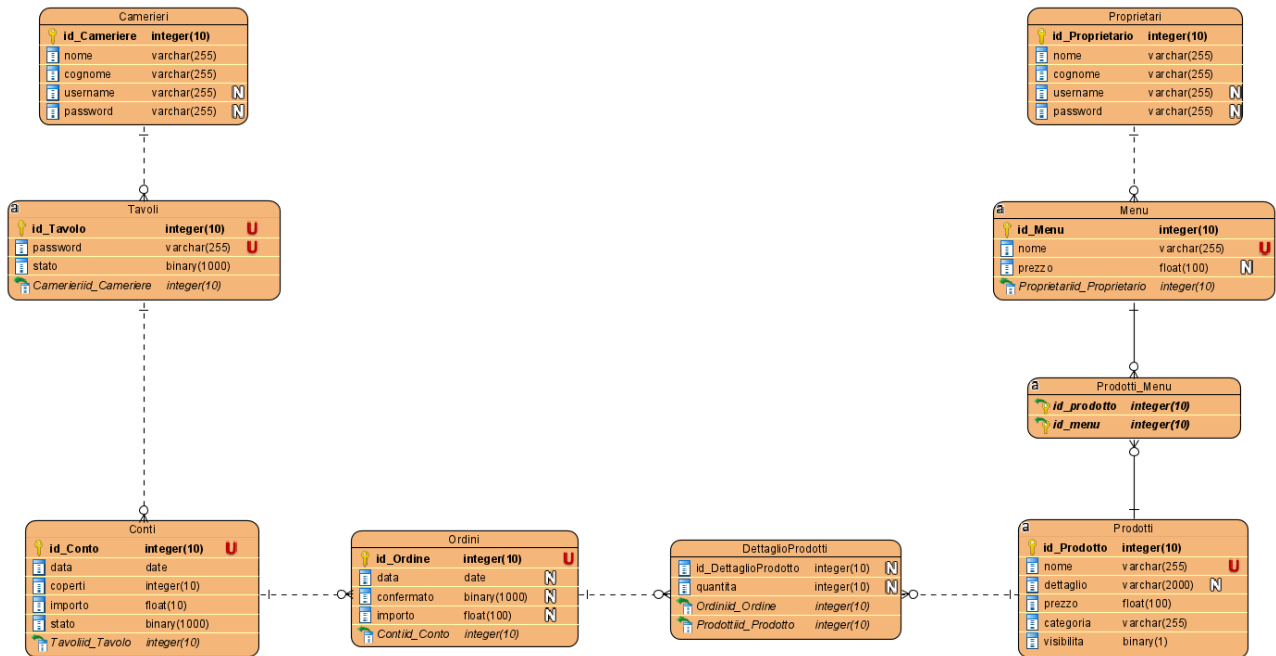
## SD raffinato Salda Conto Attivo



## Schema del database: Diagramma ER

Lo schema del database è generato automaticamente tramite le annotazioni *Hibernate* poste nelle classi *entity* del sistema.

Tuttavia, la precisione della generazione dello schema non è garantita in automatico, ma dipende dall'inserimento di annotazioni all'interno del codice. Quindi si è comunque scelto di progettare lo schema della base dati indipendentemente da *Hibernate* per poi verificare che il risultato della generazione automatica soddisfasse i requisiti.





## Implementazioni future

Soddisfatti i requisiti principali dell'applicazione, sono poi stati selezionati una serie di requisiti sia funzionali che non funzionali per future iterazioni.

In particolare, sono state previste per la prossima iterazione:

- Sviluppo del caso d'uso che permette al cameriere di modificare l'ordine effettuato dal cliente.
- Permettere al proprietario la gestione del menù, dei tavoli e del personale di sala direttamente dalla sua user interface.
- Correzione di errori e bug riscontrati in fase di testing

Sono poi previste le seguenti funzionalità aggiuntive:

- Possibilità di scaricare sul proprio dispositivo il menù di sola visione in formato PDF.
- Aggiunta nella homepage di tasti di reindirizzamento alle pagine del locale sui principali social network (es. Facebook, Instagram, TripAdvisor).

Infine, si è previsto di aggiungere all'interno del database:

- Un trigger che elimini automaticamente i conti saldati dopo 48h.

Inoltre, il team di sviluppo ha pensato di cominciare l'analisi e progettazione di un nuovo caso d'uso che fornisca al cliente autenticato di prenotare un tavolo direttamente da casa, tramite la compilazione di un form come quello sottostante.


Per favore, controlla con attenzione che tutti i campi richiesti siano inseriti correttamente. Le prenotazioni non compilate correttamente non saranno registrate. Utilizzando questo form, ci autorizzi al trattamento dei tuoi dati personali.


È possibile prenotare un tavolo tutti i giorni, dal martedì alla domenica, nei seguenti orari: - Pranzo, nelle fasce orarie 12-13, 13-14 e 14-15. - Cena, nelle fasce orarie 19-20, 20-21, 21-22, 22-23 e 23-24. Non saranno consentite prenotazioni in orari differenti da quelli indicati.


Per garantire a tutta la clientela una piena efficienza e godere della stessa esperienza di sempre, la permanenza ai tavoli sarà di circa 50 minuti. Ti invitiamo a non attardarti, ma ad anticiparti sempre. Inoltre ti invitiamo, in caso di lunga fila all'esterno della struttura, ad attendere il turno nella propria autovettura, ove possibile.


IMPORTANTE: Siete pregati di controllare la casella mail indicata, anche nello SPAM e nella posta indesiderata, e cliccare sul link di conferma, altrimenti la prenotazione non risulterà valida.


Ti ringraziamo per averci scelto.

 Nome e Cognome

 Telefono

 Numero di ospiti

 Email

 Data

Seleziona orario

CONFERMA LA PRENOTAZIONE

16