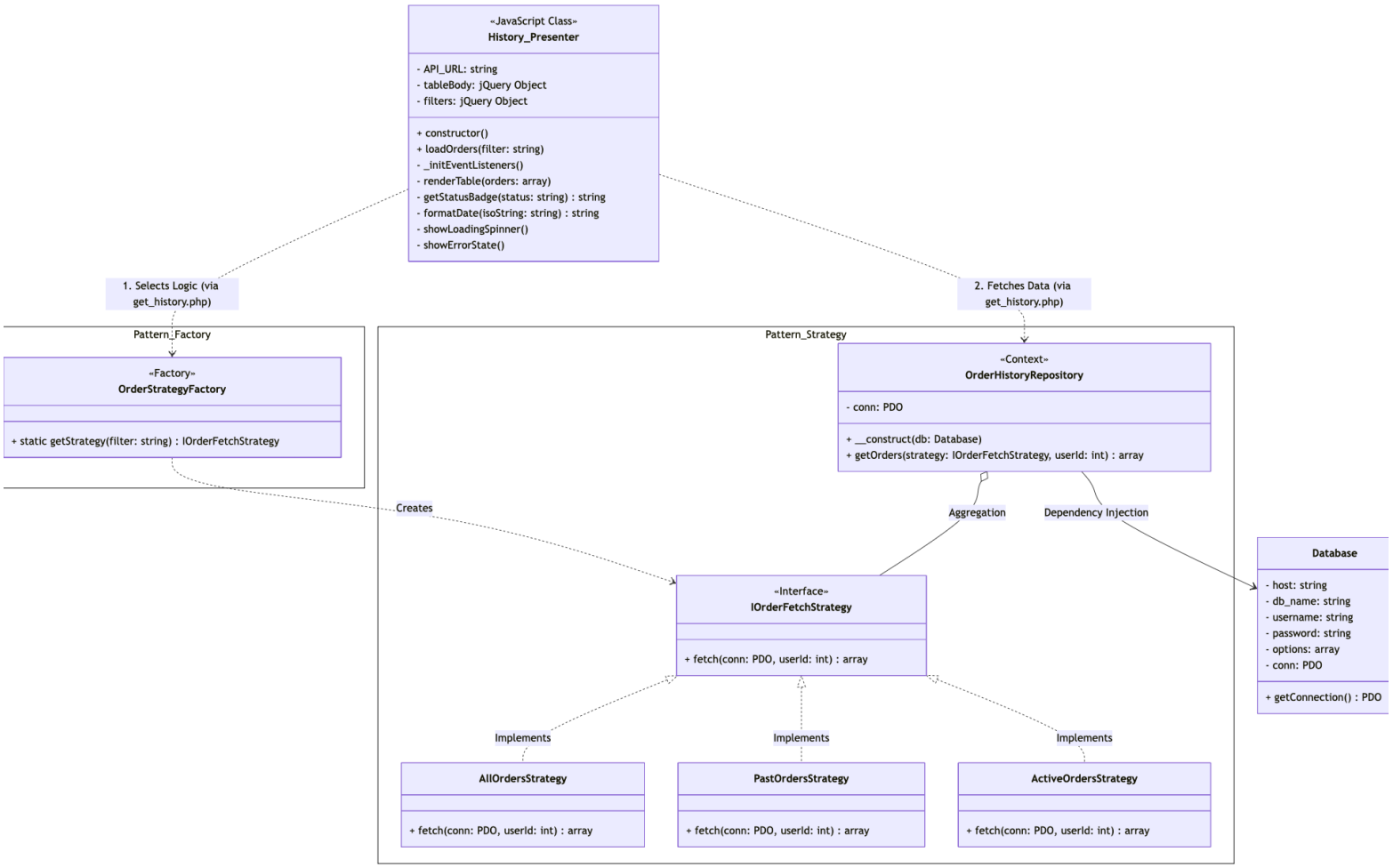
SVOLTO DA COLUCCI PASQUALE, MATR. 358141

Diagramma delle Classi: Modulo Storico Ordini



Descrizione Architetturale delle Classi: Modulo Storico Ordini

Il diagramma delle classi modella la struttura a oggetti del modulo "Storico Ordini", offrendo una vista di dettaglio pronta per l'implementazione. L'architettura è progettata per garantire una rigorosa *Separation of Concerns* e utilizza pattern per assicurare manutenibilità ed estendibilità.

Il sistema è suddiviso in due macro-livelli logici: **Frontend (Client-Side)** e **Backend Model (Server-Side)**, collegati da relazioni di dipendenza logica.

1. Frontend Layer: Il Presenter

La logica client-side è interamente incapsulata nella classe JavaScript **`History_Presenter`**, che implementa il ruolo di *Presenter* nel pattern architetturale MVP (Model-View-Presenter).

- **Incapsulamento dello Stato:** La classe protegge lo stato interno definendo attributi privati (`-`) per i riferimenti al DOM (`tableBody`, `filters`) e per la configurazione (`API_URL`). Questo impedisce la manipolazione diretta dell'interfaccia da parte di agenti esterni.
- **Logica di Presentazione:** L'interfaccia pubblica è ridotta all'essenziale (`+loadOrders`), mentre la complessità algoritmica di rendering e formattazione è delegata a metodi privati specifici (`-renderTable`, `-getStatusBadge`, `-formatDate`), garantendo il principio di *Single Responsibility*.

2. Interazione Frontend-Backend

Il diagramma evidenzia il **flusso logico di dipendenza** (frecce tratteggiate `..>`) che lega il Presenter ai componenti del Backend. Sebbene la comunicazione fisica avvenga tramite chiamate HTTP asincrone (gestite dallo script procedurale API), logicamente:

1. Il Presenter **dipende dalla Factory** (`OrderStrategyFactory`) poiché il filtro inviato dal client determina quale strategia verrà istanziata.
2. Il Presenter **dipende dal Repository** (`OrderHistoryRepository`) in quanto consumatore finale della struttura dati JSON prodotta.

3. Backend Model Layer: Business Logic

Il cuore del sistema è modellato per supportare query dinamiche senza l'uso di complessi blocchi condizionali, grazie all'uso combinato di due pattern:

- **Pattern Factory Method (`OrderStrategyFactory`):** Una classe di servizio (Factory statica) responsabile della logica decisionale. Il metodo `+getStrategy(filter)` centralizza la creazione delle istanze, disaccoppiando il codice che usa le strategie da quello che le crea.
- **Pattern Strategy (`IOrderFetchStrategy`):** Definisce un comportamento polimorfico per il recupero dati.
    - **Interfaccia:** `IOrderFetchStrategy` impone il contratto `+fetch(conn, userId)`.
    - **Strategie Concrete:** Le classi `AllOrdersStrategy`, `PastOrdersStrategy` e `ActiveOrdersStrategy` implementano l'interfaccia, incapsulando ciascuna la query SQL specifica per quel tipo di visualizzazione. Essendo componenti *stateless* (senza stato), ricevono la connessione al database come dipendenza del metodo.
- **Pattern Context (`OrderHistoryRepository`):** Agisce da contesto di esecuzione. Utilizza la **Dependency Injection** per ricevere l'istanza del database nel costruttore (`+__construct`) e mantiene la connessione come attributo privato (`-conn`). Il metodo `+getOrders` orchestra l'operazione: riceve una strategia generica ed esegue il fetch, rimanendo agnostico rispetto al tipo specifico di ordine richiesto.

4. Data Access Layer

La classe **`Database`** astrae la connessione fisica al DBMS. Include attributi privati per le credenziali e, specificamente, un array `-options` per configurare in modo robusto il driver PDO (gestione eccezioni, fetch mode associativo), esponendo la connessione tramite il metodo `+getConnection()`.