Il mio progetto ha una struttura modulare con 3 domini principali:

* **ASTA** per la gestione delle aste e delle offerte.
* **ITEM** per gli oggetti.
* **LOGIN** per gli utenti.

La struttura del progetto segue il pattern **package by feature**, dove ogni pacchetto contiene tutto ciò che serve per quella determinata feature. Questa separazione è pensata per permettere a ciascun modulo di evolvere indipendentemente.

**LOGIN**

**Entity e DTO**

La gestione del login è strutturata per consentire registrazione e login utente, oltre alla scelta del ruolo. L’entity **Utente** è composta da:

* id univoco
* username
* email
* password
* ruolo
* dataCreazione

Questi campi sono gestiti tramite DTO specifici per la registrazione e il login, sia lato richiesta che lato risposta.

**Perché usare i DTO?** Il DTO separa i dati di trasferimento dalla logica di business, migliorando la manutenibilità e la sicurezza.

Un caso particolare è il **GestoreRegisterRequest**, che estende RegisterRequest aggiungendo campi relativi al codice gestore. Questo approccio consente un downcast esplicito per gestire un'istanza di GestoreRegisterRequest quando necessario.

**Controller**

Ho unificato la logica DAO e quella di business nel **Service Layer** per ragioni pragmatiche, considerando la scala del progetto. Utilizzo una struttura di risposta tramite ResponseEntity per gestire errori e fornire risposte strutturate alle richieste. Qui, i controller si occupano di gestire le richieste e trattare eventuali errori.

**Service**

Nel **Service**, implemento sia la logica di business che la ricezione dei dati dai mapper, evitando di usare un DAO esplicito per quel compito. Utilizzo:

* **Iniezione di dipendenze manuale** tramite costruttore con attributi final. Questo garantisce l'immutabilità dopo l'inizializzazione e previene modifiche accidentali agli attributi.
* Non uso @Autowired per migliorare il debugging e mantenere un controllo più chiaro sulle dipendenze.

Il metodo registra è **transazionale**, il che significa che l'intero processo deve essere completato con successo; in caso contrario, il database effettua un rollback. Questo metodo:

1. Accetta come parametro il DTO di registrazione.
2. Valida i dati.
3. Crea un'istanza di Utente, impostandone gli attributi tramite i getter del DTO con una conversione manuale.

Per assegnare il ruolo di gestore, utilizzo un **downcast sicuro**. La gerarchia è ben definita: un GestoreRegisterRequest estende RegisterRequest, e la validazione avviene tramite instanceof per verificare il tipo di richiesta.

Ad ogni login, genero una sessione associata all'utente tramite una LoginResponse. La validazione delle sessioni avviene attraverso una classe di sicurezza che:

* Recupera l'utente corrente tramite getCurrentUser.
* Verifica l'header contenente Authorization e sessionId.
* Se validato, genera un token di autenticazione.

**Sistema di Aste**

**Gestione vincitore**

Il sistema sfrutta la catena **Controller → Service → Mapper** per accedere al database. Attraverso una **doppia left join**, ottengo l'utente che ha effettuato l'ultima offerta valida quando l'asta ha lo stato "terminata" e "non attiva".

**Scheduler delle aste**

Le aste sono gestite tramite un bean specifico chiamato **AstaScheduler**, che implementa il metodo checkAsteScadute. Questo metodo termina le aste scadute e viene eseguito ogni 60 secondi grazie a uno scheduler.

Bean  
I Bean in Spring consentono di gestire

**Notifiche**

Per le notifiche in tempo reale ho utilizzato **WebSocket**, un protocollo che mantiene una connessione aperta tra client e server. Questo sistema:

* Evita il polling continuo.
* Gestisce la logica di business tramite un servizio dedicato.

Ho deciso di limitare l'uso di WebSocket alle sole notifiche, poiché:

1. Una latenza di qualche secondo è accettabile per la chiusura delle aste, ma non per le notifiche.
2. L'implementazione di WebSocket è più onerosa rispetto ad altri approcci.

CLIENT  
  
**Login e Register**  
Quando cerco un errore restituito dal server, accedo a response.data utilizzando un punto interrogativo (?.). Questo perché response.data potrebbe essere undefined. L'operatore opzionale evita potenziali errori durante l'accesso a proprietà che potrebbero non esistere.

**useEffect**  
Nella mia applicazione, useEffect gestisce gli effetti collaterali, come il caricamento iniziale dei dati e la gestione del ciclo di vita del componente. Funzioni come handleSubmit sono invece event handlers, progettate per gestire eventi specifici e non rientrano nell'ambito di useEffect.

**handleChange**  
La funzione handleChange aggiorna lo stato del form. La proprietà name corrisponde alla chiave dello stato, mentre value rappresenta il nuovo valore inserito dall'utente.

**Chiamate con Axios**  
Con Axios, eseguiamo chiamate asincrone che utilizzano le promise. Ad esempio, inviamo i dati di registrazione al server:

javascript

CopiaModifica

const [datiRegistrazione, setDatiRegistrazione] = useState({

username: "",

email: "",

password: "",

});

Inviamo questi dati (username, email e password) come parte della richiesta.

**handleSubmit**  
La funzione handleSubmit previene il comportamento predefinito dell'evento, evitando il refresh della pagina al momento dell'invio del form.

**HomePage**

* Utilizzo l'hook useState per mantenere lo stato locale dei dati.
* Lo useEffect gestisce il ciclo di vita del componente, come il caricamento iniziale dei dati e la verifica della sessione dell'utente.

**SessionId**  
Per accedere alla homepage, è necessario un sessionId, ottenuto durante la fase di login, salvato nello storage locale e verificato tramite una richiesta HTTP asincrona. Se la sessione è valida, lo stato dell'utente viene aggiornato con l'ID.

**Fetch delle aste**  
Per ottenere le aste attive:

1. Effettuo una chiamata GET al server utilizzando il sessionId per maggiore sicurezza.
2. Imposto lo stato di asteAttive con la risposta ricevuta.
3. Se esistono aste (lunghezza > 0), creo un oggetto itemMap che cicla attraverso le aste per estrarre i dati direttamente dal database.

**Logout**  
Il logout rimuove il sessionId dallo storage locale e reindirizza l'utente alla pagina di login.

**AstaDettaglio**

* Uso useState per gestire lo stato locale del componente.
* Lo useEffect è responsabile del ciclo di vita e degli effetti collaterali, come il caricamento dei dati specifici dell'asta.

**Fetch dei dati**  
Con fetchData, eseguo una richiesta asincrona per ottenere i dettagli dell'asta:

javascript

CopiaModifica

const astaResponse = await axios.get(`http://localhost:8080/api/aste/${id}`, {

headers: { Authorization: sessionId },

});

setAsta(astaResponse.data);

La funzione è asincrona per migliorare l'esperienza utente: mentre i dati vengono caricati, l'interfaccia non si blocca.

**Sequenzialità**  
Le chiamate in fetchData sono sequenziali perché ogni chiamata potrebbe dipendere dalla precedente. Se una chiamata fallisce, le successive non vengono eseguite. Questo approccio garantisce robustezza ma introduce una dipendenza tra le chiamate.  
  
**Uso del Null**  
Il post si aspetta sempre un corpo, lo ometto mettendo null, i parametri invece vengono inseriti nell’url

**Parallelizzazione**  
Per alcune operazioni, è possibile utilizzare Promise.all per eseguire chiamate indipendenti in parallelo, ottimizzando i tempi:

javascript

CopiaModifica

const [astaResponse, offerteResponse] = await Promise.all([

axios.get(`http://localhost:8080/api/aste/${id}`, {

headers: { Authorization: sessionId },

}),

axios.get(`http://localhost:8080/api/aste/${id}/offerte`, {

params: { userId: user.id },

headers: { Authorization: sessionId },

}),

]);

Questo consente di aggiornare contemporaneamente lo stato di asta e offerte, dato che le due chiamate non dipendono l'una dall'altra.

**Dipendenze nello useEffect**  
Nelle dipendenze dello useEffect, l'id è cruciale: rappresenta l'identificativo dell'asta. Se l'utente naviga verso un'altra asta (modificando l'id), tutti i dati vengono ricaricati per garantire coerenza.

Ecco una versione migliorata e più fluida della tua spiegazione:

**AstePartecipate:**

Il componente è una funzione che utilizza **hook React** per gestire lo stato e il ciclo di vita del componente.

1. **Gestione dello stato (useState)**
   * Utilizziamo useState per:
     + Memorizzare i dati dell'utente (user).
     + Salvare le aste a cui l'utente ha partecipato (astePartecipate).
     + Tracciare lo stato di caricamento (loading).
2. **Gestione del ciclo di vita (useEffect)**
   * Lo useEffect viene utilizzato per eseguire il caricamento iniziale dei dati e per gestire gli effetti collaterali.
   * All'inizio, recuperiamo il sessionId dallo storage locale per verificare se l'utente è autenticato.
     + Se il sessionId non è presente, l'utente viene reindirizzato alla pagina di login con navigate("/login").
   * In caso contrario, viene chiamata la funzione asincrona fetchData per caricare i dati necessari.
3. **Caricamento dei dati con fetchData**
   * La funzione fetchData effettua due richieste API **sequenziali**, perché la seconda richiesta dipende dal risultato della prima:
     + La prima richiesta recupera i dati dell'utente autenticato (/api/utenti/me).
       - La risposta viene salvata nello stato user utilizzando setUser.
     + La seconda richiesta utilizza l'ID dell'utente ottenuto dalla prima risposta per ottenere le aste a cui ha partecipato (/api/aste/partecipate/${userResponse.data.id}).
       - Il risultato viene salvato nello stato astePartecipate con setAstePartecipate.
   * La sequenzialità è necessaria poiché non possiamo ottenere le aste partecipate senza conoscere prima l'ID dell'utente.
4. **Ricaricamento dei dati**
   * Lo useEffect ha come dipendenza navigate. Ogni volta che questa funzione cambia (ad esempio, in seguito a una navigazione), l'intero processo di caricamento dei dati viene rieseguito.
   * Questo garantisce che, se l'utente naviga tra diverse pagine, i dati siano sempre aggiornati e coerenti con il contesto.

CreateAstaForm:  
  
Un componente funzione con delle proprietà destrutturate, che dicono, mi aspetto che il componente che mi utilizzerà sfrutterà queste props destrutturate.  
  
Creo fondamentalmente una copia locale che poi gestoreItems utilizzerà, questo componente poi si occuperà di inviarli al server!  
  
questo viene definito come componente controllato, il submit di questo componente viene sfruttato dal gestoreItems, che è colui che invierà poi questi dati, qui stiamo creando una bozza locale.  
  
    nome: "",

    descrizione: "",

    prezzoBase: "",

    rilancioMinimo: "",

    imageUrl: "",

  });

Solito useEffect che gestisce gli effetti collaterali di quel componente ed il ciclo di vita dei componenti.  
FetchData sequenziali, uno dipende dall’altro dall’alto verso il basso, se qualcosa va male qui allora il resto non lo carico, proprio perché sono asincroni.  
  
Il primo fetch è relativo all’utente, quindi lo stessa e lo autorizza, il secondo fa capire che tutte quelle chiamate sono chiamabili solo dal gestore.  
  
carico gli oggetti, relativi a quell’utente ovvero a quel gestore e setto gli oggetti prendendomi dal server con itemsResponse.data e setto il caricamento come falso per evitare che ad ogni cambiamento di pagina non abbia quelle cose e che quindi il tutto venga ricaricato.  
  
 Parte più lunga invece è quella relativa agli handle, in quanto uno prende l’oggetto e come corpo gli passa uno spread operator di nuovoItem, perché? Perché copia tutte le proprietà dell’oggetto, in più il corpo vuole il prezzoBase ed il rilancio minimo, settati al nuovo oggetto.  
  
 setItems([...items, response.data]);

      setNuovoItem({

        nome: "",

        descrizione: "",

        prezzoBase: "",

        rilancioMinimo: "",

      });

      setError("");

    } catch (err) {

      setError(err.response?.data || "Errore nella creazione dell'item");

    }

  };

Questa parte è da chiarire, comunque poi per creare l’asta io fondamentalmente comeSelected item mi prendo l’item, ma credo sia quello relativo al CreateAstaForm, e poi creo l’asta con i dati che avevo creato in precedenza, è proprio qui che metto come props il formData, questo componente finalmente li usa, questo va ad aggiungere una data di inizio e fine,  
      const requestData = {

        itemId: *formData*.itemId,

        startNow: *formData*.startNow,

        dataInizio: *formData*.startNow ? null : dataInizio.toISOString(),

        dataFine: dataFine.toISOString(),

      };

Questo mi va a settare tutti i parametri che mi servono, e con un post invierò come corpo della richiesta quel corpo al server, che chiamerà i membri e salverà l’oggetto nel db  
  
      const itemsResponse = await axios.get(

        `http://localhost:8080/api/items/gestore/${user.id}`,

        { headers: { Authorization: localStorage.getItem("sessionId") } }

      );

      setItems(itemsResponse.data);

      setShowAstaForm(false);

      setSelectedItem(null);

    } catch (err) {

      // Log più dettagliato dell'errore

      console.error("Errore completo:", err);

      console.error("Dettagli risposta:", err.response?.data);

      console.error("Status code:", err.response?.status);

      setError(err.response?.data || "Errore nella creazione dell'asta");

    }

  };

Questo viene definito come aggiornamento ottimistico, perché ricaricare tutti gli oggetti con lo stato più aggiornato del server, chiude il form e resetta l’item selezionata, nel senso ho finito di creare l’asta, mostrami la lista e pulisci tutto