



DEVELOPER GUIDE

Componenti: Iannoli Andrea, Mazzini Alessandro, Morelli Marco, Rontini Matteo
Progetto SWENG A.A. 2022/23

Introduzione per come lanciare test e applicazione

Per lanciare i test e l'applicazione in maniera corretta bisogna seguire i seguenti passaggi:

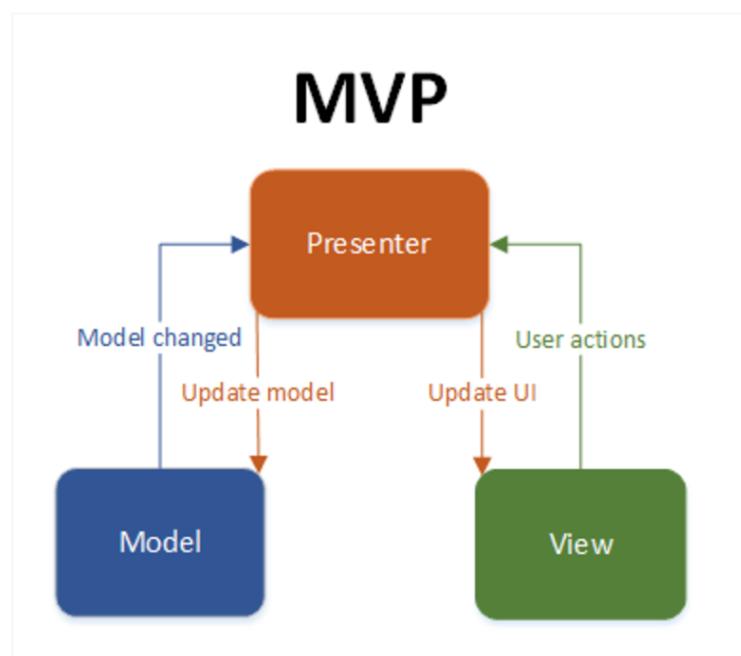
- Avere installato Java 11* all'interno del proprio dispositivo
- Clonare la seguente repository da GitHub:
<https://github.com/Andrealannoli/CardsMule.git>
- Avere installato Eclipse nel proprio dispositivo
- Aprire il progetto all'interno di Eclipse e fare un clean del progetto
- Lanciare il codice con run As e selezionare GWT Development Mode with Jetty
- A questo punto si aprirà "Jetty", una finestra di controllo dell'ambiente di esecuzione di GWT, che farà.
avviare l'applicazione. Una volta fatto questo,
è possibile aprire la pagina dell'applicazione all'indirizzo "127.0.0.1:8888" ,
cliccando sul link che viene mostrato.

*La restrizione della versione di Java per l'esecuzione del progetto è stata imposta dopo che sono stati riscontrati dei problemi con versioni superiori a Java 11 (es. Java 18). Nel dettaglio Java 11 permette l'esecuzione pur lanciando dei warning mentre Java 8 permette l'esecuzione senza alcun problema.

PATTERN ARCHITETTURALE

Inizialmente, date anche le nostre pregresse conoscenze accademiche volevamo adottare il pattern MVC (Model-View-Control). Ricercando bene sul web abbiamo però scoperto che la tendenza per quanto riguarda i pattern architetturali nello sviluppo software sta cambiando, al giorno d'oggi si tende infatti a scegliere pattern che permettono una maggiore divisione del codice e che permettono di conseguenza eseguire con più facilità testing e manutenzione, alcuni di questi pattern sono per esempio, MVVM o MVP.

Confrontando i vari pattern e per i motivi già elencati in precedenza abbiamo deciso di utilizzare il pattern MVP. Avremo quindi dei model, delle view, e dei presenter che nel nostro caso, come “deformazione” derivante dallo sviluppo nativo in android, abbiamo chiamato “activity”.



STRUTTURA DEL CODICE

Il codice per la realizzazione dell'applicazione è strutturato in 3 parti che sono le seguenti:

- CLIENT
- SERVER
- SHARED

In egual modo anche i test sono suddivisi nelle stesse categorie.

CLIENT

Il client rappresenta l' interfaccia utente della nostra applicazione.

La parte client da noi sviluppata è divisa in diversi package, i quali sono:

- Activity
- Places
- Authentication
- View
- Widgets
- Routes
- Handlers

ACTIVITY

Le Activity nel nostro progetto rappresentano i presenter del modello MVP, ogni activity contiene sia la parte logica che un' insieme di cicli di vita.

Le activity sono responsabili di gestire lo stato del client, ci consentono di creare applicazioni web interattive in cui le diverse schermate dell'applicazione possono essere caricate dinamicamente senza dover ricaricare l'intera pagina.

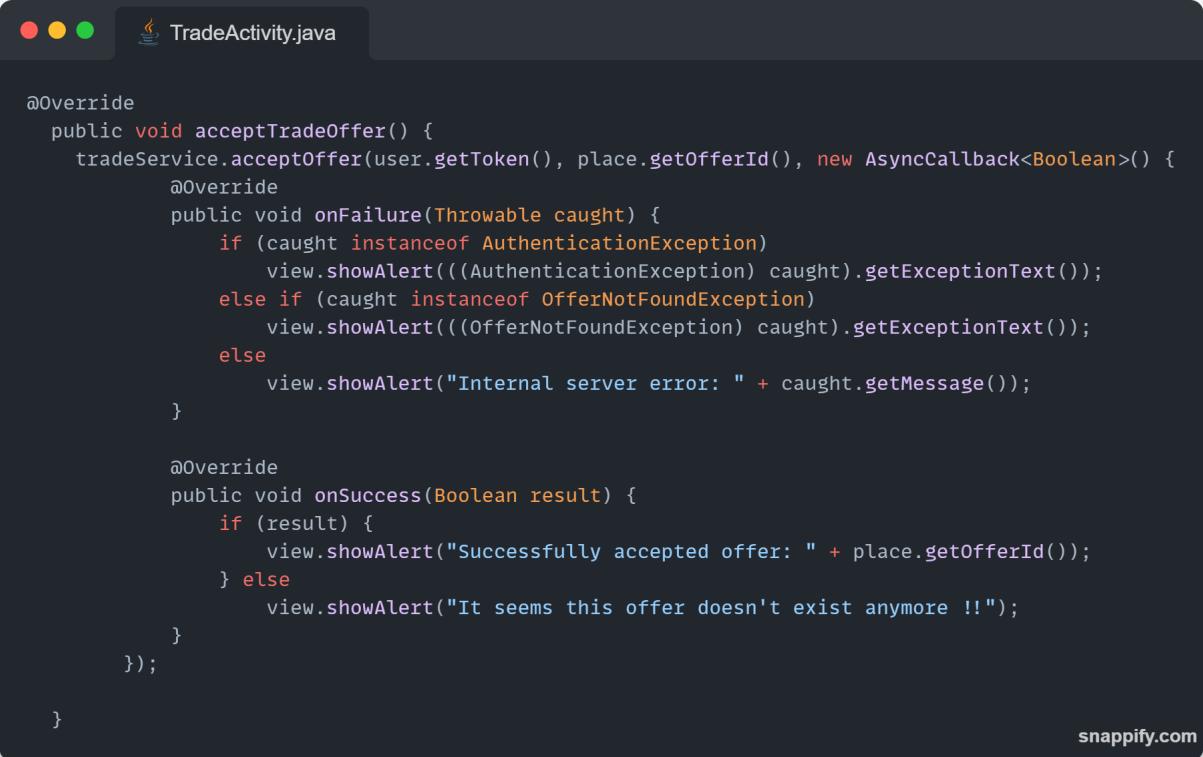
All' interno dell'applicazione abbiamo una serie di activity. Ogni activity può essere progettata per gestire una specifica funzionalità o aspetto dell'applicazione (esempio `HomeActivity` per la gestione della homepage oppure `TradeActivity` per la gestione della parte di trade).

Le activity possono anche comunicare con il server per recuperare o aggiornare dati. GWT fornisce un modo efficiente per la comunicazione asincrona con il server, consentendo alle activity di ottenere dati senza interruzioni nell' interfaccia utente,

nelle activity vengono fatte delle chiamate alle classi asincrone della sezione shared che contengono i metodi che le classi del server implementano .

Le activity hanno un ciclo di vita, che include metodi come `start()`, `stop()`, ecc. Questi metodi consentono di inizializzare, pulire le risorse e di gestire gli eventi all'interno dell'activity.

Un esempio di classe activity che abbiamo messo nel nostro codice è il seguente:



The screenshot shows a code editor window with a dark theme. The title bar says "TradeActivity.java". The code is written in Java and handles accepting a trade offer. It uses an `AsyncCallback<Boolean>` to handle the response from the server. The code includes methods for handling success and failure cases, as well as displaying alerts to the user.

```
@Override
public void acceptTradeOffer() {
    tradeService.acceptOffer(user.getToken(), place.getOfferId(), new AsyncCallback<Boolean>() {
        @Override
        public void onFailure(Throwable caught) {
            if (caught instanceof AuthenticationException)
                view.showAlert(((AuthenticationException) caught).getExceptionText());
            else if (caught instanceof OfferNotFoundException)
                view.showAlert(((OfferNotFoundException) caught).getExceptionText());
            else
                view.showAlert("Internal server error: " + caught.getMessage());
        }

        @Override
        public void onSuccess(Boolean result) {
            if (result)
                view.showAlert("Successfully accepted offer: " + place.getOfferId());
            else
                view.showAlert("It seems this offer doesn't exist anymore !!");
        }
    });
}
```

snappyf.com

Questo metodo implementa l'azione dell'utente che accetta un'offerta di scambio, successivamente alla conferma di un' alert avvenuta tramite bottone.

`tradeService.acceptOffer(...)`: è una chiamata di metodo che coinvolge un servizio chiamato `tradeService`, quest'ultimo è una variabile di istanza della classe `TradeCardsServiceAsync` implementata nella sezione shared, nella quale sono presenti i metodi implementati dal server tra cui `acceptOffer()`.

`user.getToken()`: questo codice recupera un token di autenticazione dall'oggetto `user`. Questo token è utilizzato per autenticare l'utente presso il server.

`place.getOfferId()`: rappresenta un metodo che restituisce l'ID dell'offerta che l'utente desidera accettare.

`new AsyncCallback<Boolean>() { ... }:` l' `AsyncCallback` viene utilizzata per gestire le risposte asincrone dal server dopo aver eseguito la richiesta. L'interfaccia prevede due metodi da implementare: `onFailure(Throwable caught)` per gestire le

situazioni in cui la richiesta ha avuto esito negativo e `onSuccess(Boolean result)` per gestire le situazioni in cui la richiesta ha avuto esito positivo.

Nel metodo `onFailure(Throwable caught)`, il codice gestisce vari tipi di eccezioni che possono essere sollevate durante la richiesta al server:

- Se `caught` è un'istanza di `AuthenticationException`, mostra un messaggio di avviso specifico per i problemi di autenticazione.
- Se `caught` è un'istanza di `OfferNotFoundException`, mostra un messaggio di avviso specifico per il caso in cui l'offerta non sia stata trovata.
- In tutti gli altri casi di eccezioni, mostra un messaggio di avviso generico che indica un errore interno del server.

Nel metodo `onSuccess(Boolean result)`, il codice verifica se la richiesta è stata elaborata con successo. Se `result` è `true`, mostra un messaggio di conferma che l'offerta è stata accettata con successo insieme all'ID dell'offerta accettata. Se `result` è `false`, mostra un messaggio che indica che sembra che l'offerta non esista più.

PLACES

Un `Place` rappresenta uno stato o una posizione particolare all'interno dell'applicazione. Un `Place` può essere convertito in e da un URL history token, GWT gestisce la traduzione tra URL e Place associati. Ogni Place è associato a un presenter e a una vista. Quando si accede a una Place, il presenter corrispondente viene istanziato e la vista associata viene visualizzata.

Il `PlaceController` è un componente cruciale per l'Activity e i Places. È responsabile nel coordinare la navigazione tra diversi punti dell'applicazione. Lo fa aggiornando l'URL del browser e lo stack della cronologia del browser, in modo che l'utente possa utilizzare i pulsanti indietro e Avanti per navigare nell'applicazione e possa spostarsi nei diversi Place senza dover ricaricare completamente l'applicazione.

Il `PlaceController` funge anche da ponte tra i Place e le Activity. Quando un utente naviga in un nuovo place, `PlaceController` chiamerà l'attività appropriata per gestire la transizione al nuovo place e fargli visualizzare la corrispondente schermata.

Un Place tipo è il seguente:

Il costruttore `GameCardDetailsPlace(int idCard, CardsmuleGame game)` accetta due parametri: l'ID della carta di gioco (`idCard`) e un oggetto `CardsmuleGame (game)`. Questi parametri vengono utilizzati per inizializzare i campi di istanza della classe.

`getIdCard()`: questo metodo restituisce l'ID della carta di gioco.

`getGame()`: questo metodo restituisce l'oggetto CardsmuleGame associato a questa "posizione" o "Place".



```
public class GameCardDetailsPlace extends Place{
    private final int idCard;
    private final CardsmuleGame game;

    public GameCardDetailsPlace(int idCard, CardsmuleGame game) {
        this.idCard = idCard;
        this.game = game;
    }

    public int getIdCard() {
        return idCard;
    }

    public CardsmuleGame getGame() {
        return game;
    }
}
```

snappify.com

AUTHENTICATION

Per gestire l'autenticazione lato client vengono utilizzate varie classi. La fase di authentication è distribuita in tre view con i rispettivi presenter. Abbiamo una pagina che funge da dispatcher, `PreAuthentication`. Come qualsiasi pagina dell'applicazione è implementata attraverso l'interfaccia della view e l'implementazione di quest'ultima, rispettivamente `PreAuthenticationView` e `PreAuthenticationViewImpl`, scelta dovuta dall'uso di PTI (Program To Interface) . Infine avremo il presenter, `PreAuthenticationActivity`. Questa prima pagina ha il solo scopo di reindirizzare lo user alle due pagine principali dell'authentication, Registration e Login.

Nella fase di Authentication lato client gioca un ruolo fondamentale la Classe `User.java`:



```
public class User {
    String token;
    String username;
    String email;

    public void setCredentials(String token, String username, String email) {
        this.token = token;
        this.username = username;
        this.email = email;
    }

    public String getToken() {
        return token;
    }

    public void resetToken() {
        this.token = null;
    }

    public String getUsername() {
        return username;
    }

    public String getEmail() {
        return email;
    }

    public boolean isLoggedIn() {
        return token != null;
    }
}
```

snappify.com

Questa classe viene utilizzata per conservare, dopo il login o la registrazione, le principali informazioni dell'utente loggato e per passare le informazioni dell'utente da una pagina all'altra, in particolare con l'ausilio della classe `ClientSession`.

In particolare è doveroso soffermarsi sul fulcro di questa classe, ovvero il token, il quale viene creato lato server e mandato in un `AsyncCallback` tipizzato per contenere un `CredentialsPayload`. Viene poi richiamato il metodo `setCredentials(String token, String username, String email)` di `User` per settare i campi dell'oggetto `User` e loggare di fatto l'utente.

```
...
@Override
public void signIn(String username, String password) {
    if(username.isEmpty() || password.length() < 8) {
        view.displayAlert("Invalid credentials");
    } else {
        AsyncCallback<CredentialsPayload> asyncCallback = new AsyncCallback<CredentialsPayload>() {
            @Override
            public void onFailure(Throwable caught) {
                if (caught instanceof AuthenticationException) {
                    view.displayAlert(((AuthenticationException) caught).getExceptionText());
                } else {
                    view.displayAlert("Unexpected error occurred");
                }
            }

            @Override
            public void onSuccess(CredentialsPayload result) {
                view.setAuthToken(result.getToken());
                user.setCredentials(result.getToken(), result.getUsername(), result.getEmail());
                goTo(new HomePlace());
            }
        };
        authenticationService.signIn(username, password, asyncCallback);
    }
}
...

```

snappify.com

Per controllare se l'utente sia loggato o meno ci si affiderà alla presenza o meno del token all'interno dell'oggetto di tipo User. In particolare viene usato il metodo `isLoggedIn()` sull'oggetto di tipo User. Un esempio d'uso di questo metodo si trova nelle classi di routing ed in particolare in `AppPlaceHistoryMapper`:

```
...
} else if (token.equals(loginLink) && !user.isLoggedIn()) {
    return new LoginPlace();
} else if (token.equals(registrationLink) && !user.isLoggedIn()) {
    return new RegistrationPlace();
} else if (token.equals(homeLink) && user.isLoggedIn() ) {
    return new HomePlace();
}
...

```

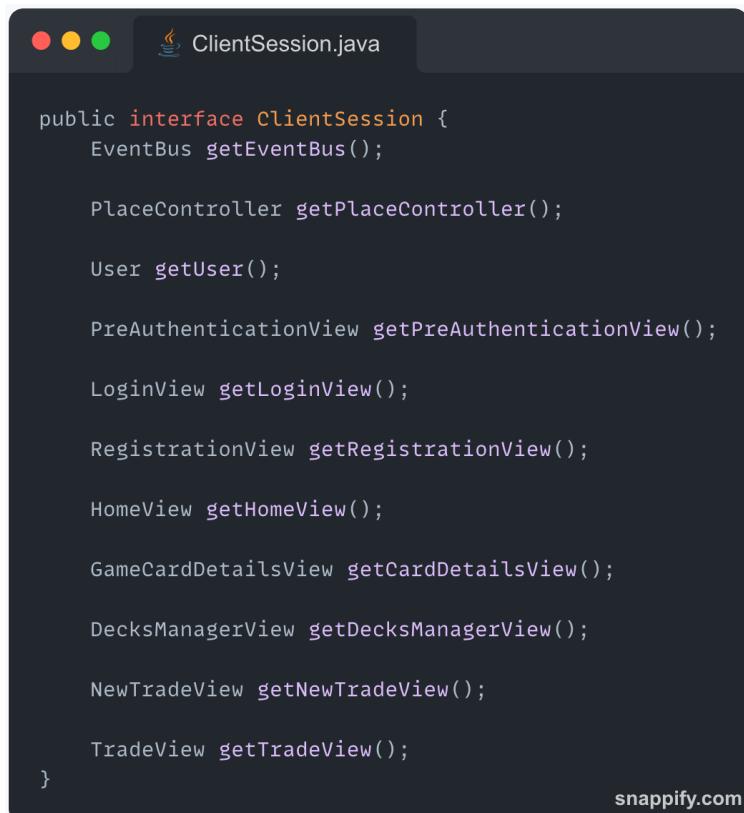
snappify.com



N.B. La classe User e Account per quanto possano sembrare uguali, svolgono due ruoli diversi. User non è serializzabile e di conseguenza si distingue da quella di Account che viene poi messa in storage e mantenuta da un'esecuzione all'altra dell'applicazione. La classe User viene infatti utilizzata esclusivamente dal client per passare da pagina in pagina le informazioni riguardanti lo user loggato, contiene informazioni riguardante la sessione di login, come il token. La classe Account viene utilizzata prevalentemente lato server per mettere in storage e conservare informazioni più delicate come la password.

CLIENTSESSION

La classe `ClientSession` e `ClientSessionImpl`, sono rispettivamente l'interfaccia e la classe che implementa l'interfaccia dell'oggetto che viene utilizzato per contenere tutti i dati utili per la sessione del client. Al suo interno possiamo trovare l'oggetto di tipo User e l'istanza singolare delle varie view dell'applicazione. ClientSession è quindi l'oggetto principale nella persistenza dei dati, riguardanti la sessione del client, tra le pagine. Il suo utilizzo è quindi per lo più relegato alle classi che svolgono funzioni di routing. Inoltre contiene un'istanza della classe `PlaceController`, già spiegata in precedenza.



```
ClientSession.java

public interface ClientSession {
    EventBus getEventBus();

    PlaceController getPlaceController();

    User getUser();

    PreAuthenticationView getPreAuthenticationView();

    LoginView getLoginView();

    RegistrationView getRegistrationView();

    HomeView getHomeView();

    GameCardDetailsView getCardDetailsView();

    DecksManagerView getDecksManagerView();

    NewTradeView getNewTradeView();

    TradeView getTradeView();
}
```

snappify.com

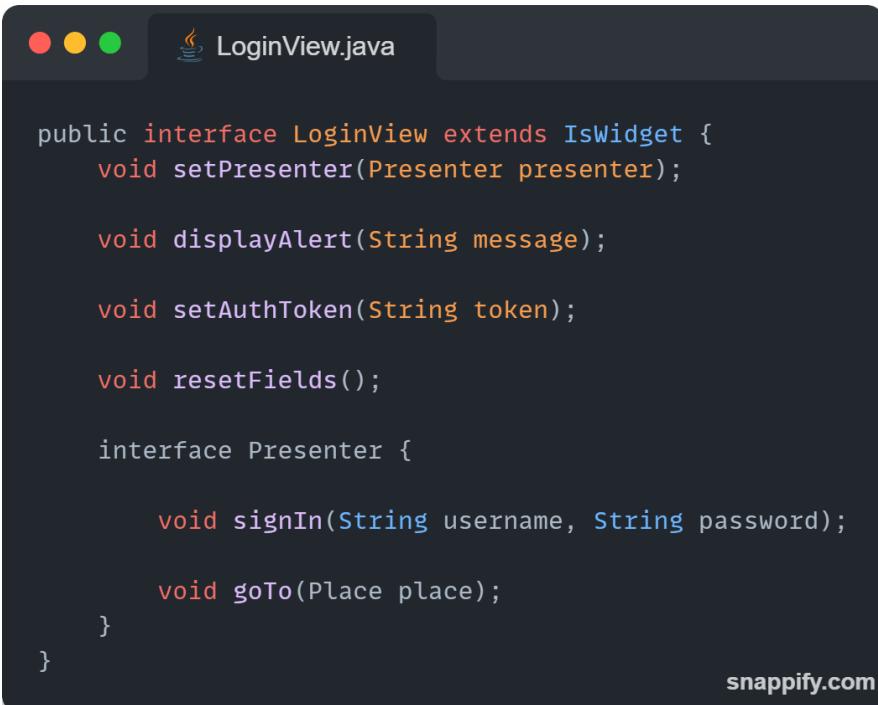
VIEW

Le view sono responsabili della rappresentazione grafica dell'interfaccia utente dell'applicazione. Questo significa che contengono i widget GWT, i layout e gli elementi HTML necessari per mostrare informazioni all'utente e consentire interazioni utente. Le view definiscono come i dati del modello (Model) vengono visualizzati e resi accessibili all'utente. La logica dell'applicazione viene separata dalla view e gestita dalla componente Presenter. Le view comunicano con il Presenter attraverso un'interfaccia definita (spesso chiamata `View`). Questa interfaccia definisce i metodi che il Presenter può chiamare per interagire con la view. Le view sono responsabili di catturare eventi utente come clic sui bottoni, inserimento di testo, selezioni e altro. Quando avvengono questi eventi, la view può generare eventi o chiamare metodi del Presenter per gestire le azioni associate a questi eventi.

Ogni view è divisa in 3 parti nel nostro progetto e sono le seguenti: Una classe, un'interfaccia e un file .xml.

Nell' interfaccia sono presenti tutti i metodi implementati nella classe, tutte le classi sono nominate nella seguente maniera: nomeclasseImpl.java

Un esempio di interfaccia è il seguente:



```
>LoginView.java

public interface LoginView extends IsWidget {
    void setPresenter(Presenter presenter);

    void displayAlert(String message);

    void setAuthToken(String token);

    void resetFields();

    interface Presenter {

        void signIn(String username, String password);

        void goTo(Place place);
    }
}
```

snappify.com

`LoginView` è un'interfaccia che estende `IsWidget` . `IsWidget` è un'interfaccia di base in GWT utilizzata per rappresentare widget che possono essere visualizzati nella UI.

L'interfaccia definisce vari metodi che devono essere implementati da una classe che funge da vista di login nell'applicazione. Questi metodi includono:

- `setPresenter(Presenter presenter)`: questo metodo consente di impostare il presenter associato a questa vista. La vista comunica con il presenter per notificare eventi dell'interfaccia utente o per mostrare dati all'utente.
- `displayAlert(String message)`: questo metodo è utilizzato per mostrare un messaggio di avviso o una notifica all'utente.
- `setAuthToken(String token)`: questo metodo è utilizzato per impostare il token di autenticazione dell'utente nella vista. Questo token può essere ottenuto dopo un'operazione di accesso con successo e deve essere utilizzato per le successive richieste al server.
- `resetFields()`: questo metodo consente di reimpostare i campi del modulo di accesso alla loro condizione predefinita o vuota.

All'interno dell'interfaccia, è definita anche un'interfaccia interna chiamata Presenter. Questa interfaccia rappresenta il presenter associato a questa vista. Il metodo `signIn` è utilizzato per gestire l'operazione di accesso, mentre `goTo` è utilizzato per navigare verso una nuova Place all'interno dell'applicazione.

Le classi del View rappresentano l'implementazione concreta di una vista:

La seguente classe estende la classe `Composite`. `Composite` è una classe di base di GWT utilizzata per creare widget composti, ovvero widget che possono contenere altri widget.

Si utilizza l'annotazione `@UiField` per associare i campi di istanza alle corrispondenti parti dell'interfaccia utente definite in un file XML di definizione delle interfacce utente (UI Binder).

La classe contiene un'interfaccia interna chiamata `LoginViewImplUIBinder`, che è un'interfaccia di legame per il file UI Binder associato a questa vista.

Nel costruttore `LoginViewImpl`, si inizializza il widget chiamando `uiBinder.createAndBindUi(this)`, che collega l'implementazione della vista ai suoi elementi dell'interfaccia utente definiti nel file XML.

Il pulsante "Login" è associato al metodo `presenter.signIn(username.getText(), password.getText())`, che invia le credenziali di accesso al presenter quando il pulsante viene premuto.

`setAuthToken(String token)`: imposta un cookie nel browser dell'utente con un token di autenticazione. Il token ha una durata di 7 giorni (calcolata in millisecondi) prima della scadenza.

`displayAlert(String message)`: Mostra una finestra di avviso nel browser con il messaggio specificato.

`setPresenter(Presenter presenter)`: imposta il presenter associato a questa vista.

`resetFields()`: ripristina i campi di testo alla loro condizione predefinita (vuoti).

La classe contiene un metodo `onBackClick()`, che viene chiamato quando l'utente preme il pulsante "Back". Questo metodo usa il presenter per navigare ad un Place chiamata `PreAuthenticationPlace`.

```
public class LoginViewImpl extends Composite implements LoginView {
    private static final LoginViewImplUIBinder uiBinder = GWT.create(LoginViewImplUIBinder.class);
    Presenter presenter;
    @UiField
    TextBox username;
    @UiField
    PasswordTextBox password;
    @UiField
    Button btnLogin;
    @UiField
    Button btnBack;

    @UiTemplate("LoginViewImpl.ui.xml")
    interface LoginViewImplUIBinder extends UiBinder<Widget, LoginViewImpl> {
    }

    public LoginViewImpl() {
        initWidget(uiBinder.createAndBindUi(this));
        username.getElement().setAttribute("placeholder", "Username");
        password.getElement().setAttribute("placeholder", "Password");
        btnBack.addClickHandler(e → onBackClick());
        btnLogin.addClickHandler(e → presenter.signIn(username.getText(), password.getText()));
    }

    @Override
    public void setAuthToken(String token) {
        final long DURATION = 1000 * 60 * 60 * 24 * 7;
        Date expires = new Date(System.currentTimeMillis() + DURATION);
        Cookies.setCookie("token", token, expires, null, "/", false);
    }

    @Override
    public void displayAlert(String message) {
        Window.alert(message);
    }

    @Override
    public void setPresenter(Presenter presenter) {
        this.presenter = presenter;
    }

    public void onBackClick() {
        this.presenter.goTo(new PreAuthenticationPlace());
    }

    @Override
    public void resetFields() {
        username.setText("");
        password.setText("");
    }
}
```

Infine il file xml implementa la parte di styling della pagina che viene visualizzata dall'utente.

WIDGETS

I widgets sono componenti dell'interfaccia utente che vengono utilizzati per costruire l'aspetto e il comportamento grafico delle pagine web, possono essere aggiunti a un'applicazione web per creare interfacce utente interattive.

I widgets nella nostra applicazione sono divisi in due parti: una classe e un file .xml
La classe serve per creare il widget vero e proprio mentre il file .xml serve per fare lo styling del widget.

Ecco un esempio di widget:



```
public class AddCardToCollectionWidget extends Composite {
    private static final AddCardToCollectionUiBinder uiBinder = GWT.create(AddCardToCollectionUiBinder.class);
    @UiField
    Button addToCollectionButton;
    @UiBindId
    ListBox collectionListBox;

    public AddCardToCollectionWidget(HandleAddCardToCollection parent) {
        initWidget(uiBinder.createAndBindUi(this));
        addToCollectionButton.addClickHandler(clickEvent -> parent.onClickAddToDeck());
    }

    public String getDeckName() {
        return collectionListBox.getSelectedValue();
    }

    interface AddCardToCollectionUiBinder extends UiBinder<Widget, AddCardToCollectionWidget> {
    }
}
```

snappyf.com

`UiBinder: AddCardToCollectionUiBinder:` è un'interfaccia generata da GWT utilizzata per definire la struttura del widget in un file XML.

`addToCollectionButton:` questo è un pulsante che gli utenti possono fare clic per aggiungere una carta alla raccolta.

`collectionListBox:` questo è un elenco a discesa che consente agli utenti di selezionare una raccolta in cui aggiungere la carta.

`AddCardToCollectionWidget` è il widget che viene inizializzato e il metodo.

`createAndBindUi(this)` viene utilizzato per collegare il widget ai dettagli definiti nell'interfaccia.

Si aggiunge un `clickHandler` al pulsante `addToCollectionButton`. Quando l'utente fa clic su questo pulsante, verrà chiamato il metodo `onClickAddToDeck()` nel

componente genitore parent. Questo significa che quando l'utente fa clic sul bottone, verrà gestito un evento nel componente genitore.

```
getNameDeck() restituisce il valore selezionato nell'elenco a discesa collectionListBox.
```

ROUTES

La classe `ClientSession` viene utilizzata all'interno dell'applicazione per la classe `AppActivityMapper`.

Istanziata all'avvio dell'applicazione, crea istanze di tutti gli oggetti disponibili nella navigazione consentendo ad `AppPlaceHistoryMapper` di controllare la navigazione proprio come un Routes Manager: passando i riferimenti corretti delle views desiderate alle rispettive Activity in base al Place definito nell'URI.

`AppPlaceHistoryMapper` consente di richiamare i Places corretti in base all'URI utilizzato per navigare l'applicazione.

Dall'indirizzo attuale viene analizzato ed estratto un token, quindi confrontato con le possibili costanti definite nella classe Route Constants: a seconda del token trovato, restituisce la views desiderata creando il relativo Place oppure reindirizza alla views predefinita (Home).

Dopo aver creato il Place, questo viene passato all'`AppActivityMapper`: questa classe gestisce il Place ricevuto e in base al risultato del confronto, restituisce l'activity corrispondente.

Qui entra in gioco la funzionalità di `ClientSession`, poiché viene utilizzata per passare i componenti all'attività selezionata, come `PlaceController`, `User` e `Views`. Le dipendenze vengono sempre iniettate nell'Activity, questo consente l'uso di `User`, `PlaceController` e `Views` come Singleton.

L'uso dei Singleton è stato scelto perché si eseguono ripetute operazioni di istanziazione in GWT durante l'utilizzo dell'applicazione e sarebbe molto dispendioso per l'utente, in quanto le operazioni sul DOM sono piuttosto costose considerando che il framework GWT si interfaccia direttamente con il DOM stesso.

Le classi implementate in questa parte sono le due già citate `AppPlaceHistoryMapper` e `AppActivityMapper`.

`AppPlaceHistory` svolge un ruolo importante nel routing e nella navigazione all'interno dell'applicazione. Il suo scopo è di mappare le stringhe di token alle posizioni (Place) dell'applicazione e viceversa consentendo agli utenti di navigare tra diverse pagine e visualizzare il contenuto corretto in base al token dell'URL.

`PlaceHistoryMapper`: questa interfaccia è utilizzata per mappare le stringhe di token alle posizioni dell'applicazione.

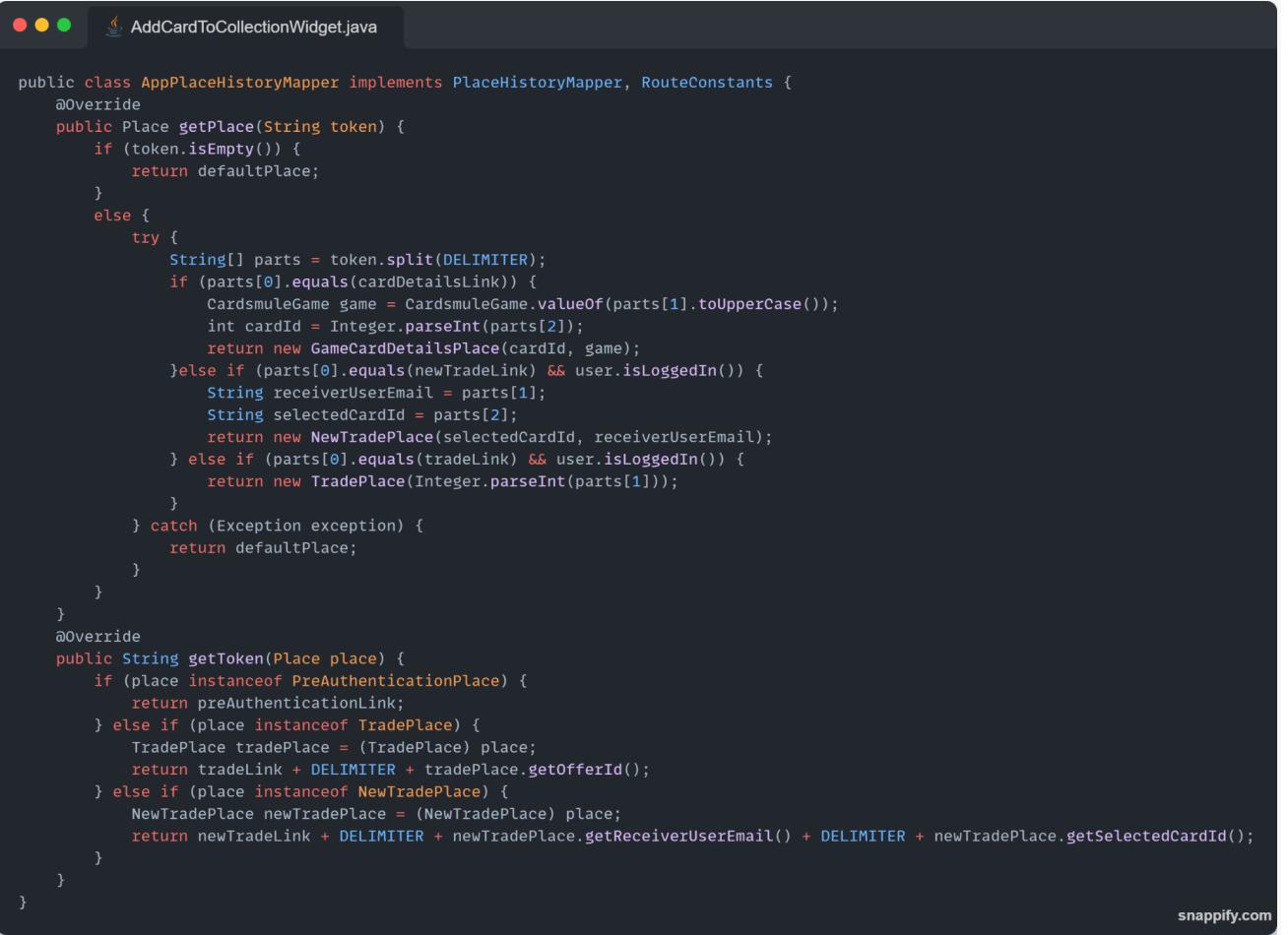
`RouteConstants`: questa interfaccia contiene costanti utilizzate per definire i token di routing all'interno dell'applicazione.

`getPlace(String token)`: questo metodo converte una stringa di token in una posizione dell'applicazione. Esegue le seguenti azioni:

- Verifica se il token è vuoto. Se lo è, restituisce la posizione predefinita, che in genere è associata alla pagina di autenticazione pre-applicazione.
- Controlla se il token corrisponde a una serie di token noti, come quelli per il login, la registrazione, la visualizzazione delle raccolte e altre pagine specifiche dell'applicazione.
- Se il token non corrisponde a nessuna delle situazioni sopra menzionate, cerca di analizzare il token come un token personalizzato, ad esempio uno associato ai dettagli di una carta di gioco o a una proposta di scambio.

`getToken(Place place)`: questo metodo fa il contrario di `getPlace`. Converte una posizione dell'applicazione in una stringa di token che può essere utilizzata per rappresentare l'URL corrente. Esegue le seguenti azioni:

- Controlla il tipo di posizione (place) e restituisce una stringa di token corrispondente.
- Se la posizione non corrisponde a nessuna delle situazioni sopra menzionate, restituirà una stringa vuota.



The screenshot shows a Java code editor window with the title "AddCardToCollectionWidget.java". The code is a class named "AppPlaceHistoryMapper" that implements "PlaceHistoryMapper" and "RouteConstants". It contains two overridden methods: "getPlace(String token)" and "getToken(Place place)". The "getPlace" method uses a try-catch block to split the token by a delimiter and then checks if it's a card details link or a new trade link. If it's a card details link, it creates a "GameCardDetailsPlace" object. If it's a new trade link and the user is logged in, it creates a "NewTradePlace" object. Otherwise, it creates a "TradePlace" object. The "getToken" method returns a string based on the type of place, including a pre-authentication link for PreAuthenticationPlace, a trade link for TradePlace, and a new trade link for NewTradePlace.

```
public class AppPlaceHistoryMapper implements PlaceHistoryMapper, RouteConstants {
    @Override
    public Place getPlace(String token) {
        if (token.isEmpty()) {
            return defaultPlace;
        }
        else {
            try {
                String[] parts = token.split(DELIMITER);
                if (parts[0].equals(cardDetailsLink)) {
                    CardsmuleGame game = CardsmuleGame.valueOf(parts[1].toUpperCase());
                    int cardId = Integer.parseInt(parts[2]);
                    return new GameCardDetailsPlace(cardId, game);
                } else if (parts[0].equals(newTradeLink) && user.isLoggedIn()) {
                    String receiverUserEmail = parts[1];
                    String selectedCardId = parts[2];
                    return new NewTradePlace(selectedCardId, receiverUserEmail);
                } else if (parts[0].equals(tradeLink) && user.isLoggedIn()) {
                    return new TradePlace(Integer.parseInt(parts[1]));
                }
            } catch (Exception exception) {
                return defaultPlace;
            }
        }
    }
    @Override
    public String getToken(Place place) {
        if (place instanceof PreAuthenticationPlace) {
            return preAuthenticationLink;
        } else if (place instanceof TradePlace) {
            TradePlace tradePlace = (TradePlace) place;
            return tradeLink + DELIMITER + tradePlace.getOfferId();
        } else if (place instanceof NewTradePlace) {
            NewTradePlace newTradePlace = (NewTradePlace) place;
            return newTradeLink + DELIMITER + newTradePlace.getReceiverUserEmail() + DELIMITER + newTradePlace.getSelectedCardId();
        }
    }
}
```

snappyf.com

`AppActivityMapper` è utilizzata per mappare le posizioni (`Place`) dell'applicazione alle attività (`Activity`) associate.

Quando un utente naviga tra le diverse pagine dell'applicazione, questa classe si assicura che l'attività corretta venga creata per gestire quella pagina specifica.

`ClientSession`: rappresenta una sessione del client e viene utilizzata per fornire le dipendenze necessarie per la creazione di attività.

`getActivity(Place place)`: Questo metodo è chiamato per ottenere l'attività associata a una determinata posizione (`place`) all'interno dell'applicazione. Esegue le seguenti azioni:

- Controlla il tipo di posizione (`place`) fornita come parametro.
- In base al tipo di posizione, crea un'istanza dell'attività associata. Ogni attività è associata a una vista specifica e gestisce la logica di quella vista.
- Passa le dipendenze necessarie (come le viste e i servizi) all'attività durante la sua creazione.
- Restituisce l'istanza dell'attività creata

```
● ● ● AddCardToCollectionWidget.java

public class AppActivityMapper implements ActivityMapper {
    private final ClientSession clientSession;

    public AppActivityMapper(ClientSession clientSession) {
        this.clientSession = clientSession;
    }

    @Override
    public Activity getActivity(Place place) {
        if (place instanceof PreAuthenticationPlace)
            return new PreAuthenticationActivity(clientSession.getPreAuthenticationView(), clientSession.getUser(), clientSession.getPlaceController());
        else if (place instanceof LoginPlace)
            return new LoginActivity(clientSession.getLoginView(), clientSession.getUser(), clientSession.getPlaceController(),
                GWT.create(AuthenticationService.class));
        else if (place instanceof GameCardDetailsPlace)
            return new GameCardDetailsActivity(clientSession.getCardDetailsView(), (GameCardDetailsPlace) place, GWT.create(CardService.class),
                GWT.create(CollectionService.class), GWT.create(AuthenticationService.class), clientSession.getUser(), clientSession.getPlaceController());
        else if (place instanceof NewTradePlace)
            return new NewTradeActivity((NewTradePlace) place, clientSession.getNewTradeView(), GWT.create(CollectionService.class),
                GWT.create(TradeCardsService.class),
                clientSession.getUser(), clientSession.getPlaceController());
        else if (place instanceof TradesPlace)
            return new TradesActivity(clientSession.getTradeView(), GWT.create(TradeCardsService.class), clientSession.getUser(),
                clientSession.getPlaceController());
        else if (place instanceof TradePlace)
            return new TradeActivity((TradePlace) place, clientSession.getNewTradeView(), GWT.create(TradeCardsService.class),
                clientSession.getUser(), clientSession.getPlaceController());
        return null;
    }
}
```

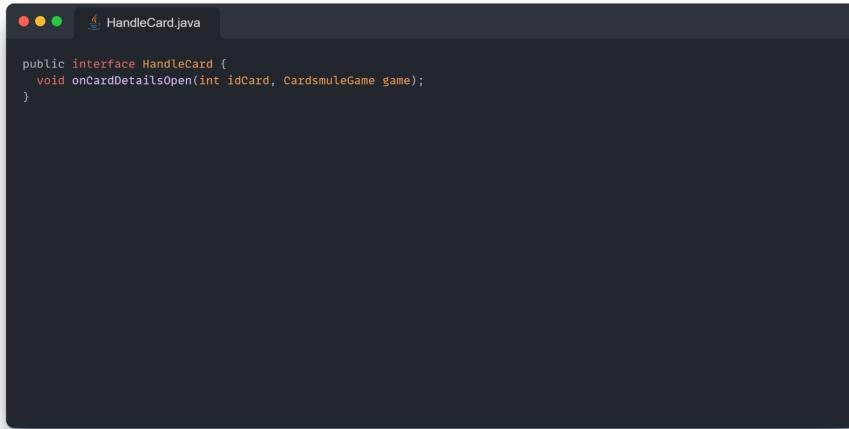
snappyify.com



N.B. Gli ultimi due file Java sono stati modificati per farli rendere meglio in foto, quindi certi metodi e condizioni sono state tagliate

HANDLERS

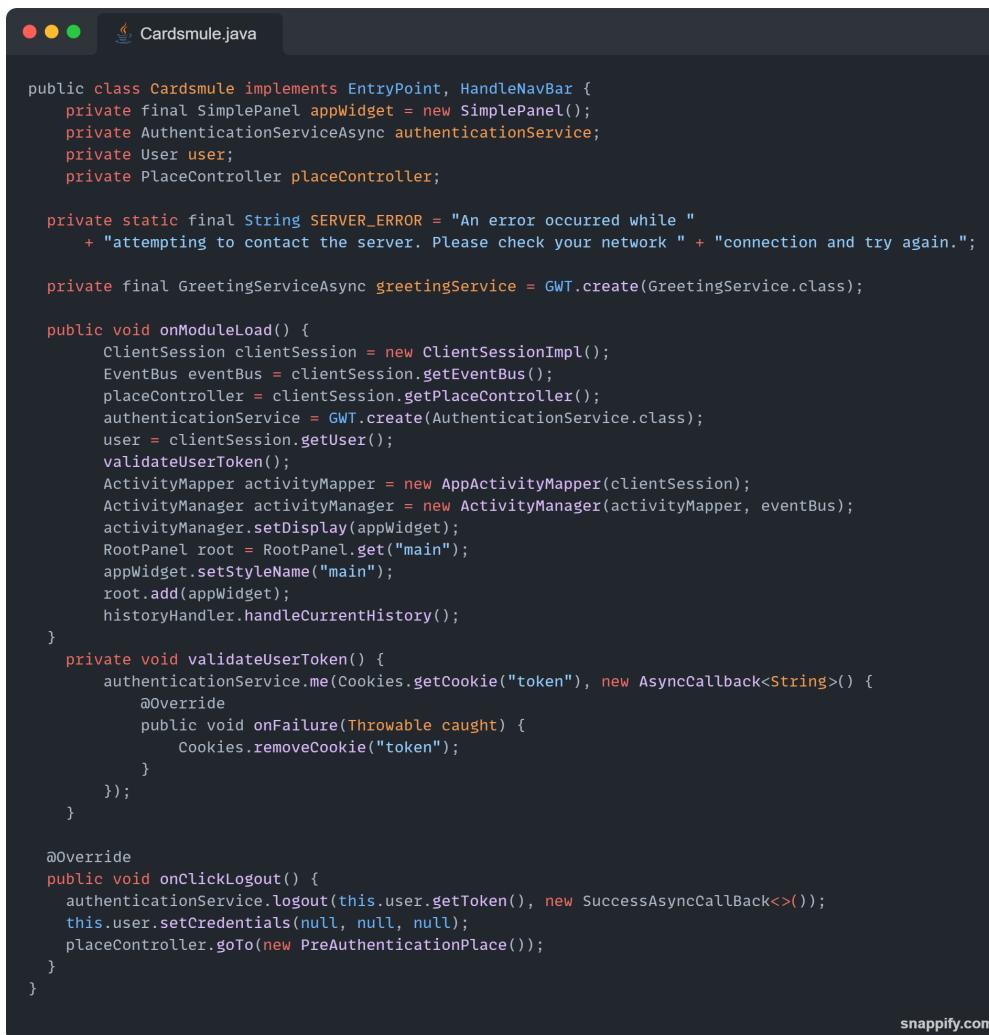
Gli handlers si riferiscono generalmente a oggetti o funzionalità utilizzati per gestire gli eventi nell'interfaccia utente. GWT fornisce un sistema di gestione degli eventi per catturare e gestire eventi generati dagli elementi dell'interfaccia utente, come clic del mouse, pressioni di tasti, movimenti del mouse e così via. Gli handlers sono spesso utilizzati per collegare il codice che deve essere eseguito in risposta a eventi specifici.



```
public interface HandleCard {
    void onCardDetailsOpen(int idCard, CardsmuleGame game);
}
```

L'interfaccia HandleCard definisce un contratto per gestire eventi legati alle carte da gioco

Cardsmule.java



```
public class Cardsmule implements EntryPoint, HandleNavBar {
    private final SimplePanel appWidget = new SimplePanel();
    private AuthenticationServiceAsync authenticationService;
    private User user;
    private PlaceController placeController;

    private static final String SERVER_ERROR = "An error occurred while "
        + "attempting to contact the server. Please check your network " + "connection and try again.";

    private final GreetingServiceAsync greetingService = GWT.create(GreetingService.class);

    public void onModuleLoad() {
        ClientSession clientSession = new ClientSessionImpl();
        EventBus eventBus = clientSession.getEventBus();
        placeController = clientSession.getPlaceController();
        authenticationService = GWT.create(AuthenticationService.class);
        user = clientSession.getUser();
        validateUserToken();
        ActivityMapper activityMapper = new AppActivityMapper(clientSession);
        ActivityManager activityManager = new ActivityManager(activityMapper, eventBus);
        activityManager.setDisplay(appWidget);
        RootPanel root = RootPanel.get("main");
        appWidget.setStyleName("main");
        root.add(appWidget);
        historyHandler.handleCurrentHistory();
    }

    private void validateUserToken() {
        authenticationService.me(Cookies.getCookie("token"), new AsyncCallback<String>() {
            @Override
            public void onFailure(Throwable caught) {
                Cookies.removeCookie("token");
            }
        });
    }

    @Override
    public void onClickLogout() {
        authenticationService.logout(this.user.getToken(), new SuccessAsyncCallBack<>());
        this.user.setCredentials(null, null, null);
        placeController.goTo(new PreAuthenticationPlace());
    }
}
```

`Cardsmule` è il punto di ingresso (entry point) per l'applicazione GWT. Il codice rappresenta la parte di avvio dell'applicazione GWT e definisce il flusso principale di navigazione e le azioni legate agli eventi all'interno dell'applicazione.

Nel metodo `onModuleLoad`, vengono eseguite le seguenti operazioni:

- Viene istanziato un oggetto `ClientSession`, che è parte dell'infrastruttura dell'applicazione e gestisce la sessione del client.
- Viene ottenuto un riferimento all' `EventBus` da `ClientSession`, che gestisce gli eventi all'interno dell'applicazione.
- Viene creato un `ActivityMapper` specifico dell'applicazione, un `ActivityManager`, e viene impostato il widget principale per visualizzare le attività gestite.
- Viene inizializzato un gestore della cronologia dei luoghi (`PlaceHistoryHandler`) utilizzando un `PlaceHistoryMapper` personalizzato e viene registrato con l'`EventBus`. La navigazione tra le diverse schermate dell'applicazione sarà gestita da questa classe.

Il metodo `validateUserToken` viene utilizzato per verificare il token dell'utente. Se il token è valido, vengono impostate le credenziali dell'utente. In caso di errore, il token viene rimosso e l'utente perde l'accesso.



N.B. Gli ultimi due file Java sono stati modificati per farli rendere meglio in foto, quindi certi metodi e condizioni sono state tagliate

SERVER

- gsonserializer
- mapdDB
- parseJson
- services

GSONSERIALIZER

In generale, questa classe GsonSerializer è una componente chiave per la gestione della serializzazione e deserializzazione di oggetti di un tipo generico T utilizzando la libreria GSON. Questo componente svolge un ruolo fondamentale nella gestione efficiente di oggetti, consentendo la loro conversione in una rappresentazione binaria e la successiva memorizzazione.



```
public class GsonSerializer<T> implements Serializer<T> {

    private final Gson gson;
    private Type type;

    public GsonSerializer(Gson gson) {
        this.gson = gson;
    }

    public GsonSerializer(Gson gson, Type type) {
        this.gson = gson;
        this.type = type;
    }

    @Override
    public void serialize(@NotNull DataOutput2 out, @NotNull T value) throws IOException {
        if (type != null) {
            String json = gson.toJson(value, type);
            out.writeUTF(json);
        } else {
            JSONObject jsonObj = gson.toJsonTree(value).getAsJsonObject();
            jsonObj.addProperty("classType", value.getClass().getName());
            out.writeUTF(jsonObj.toString());
        }
    }

    @Override
    public T deserialize(DataInput2 in, int available) throws IOException {
        JSONObject jsonObj = gson.fromJson(in.readUTF(), JSONObject.class);
        JsonElement classType = jsonObj.get("classType");
        if (classType == null) {
            return gson.fromJson(jsonObj, type);
        }
        return gson.fromJson(jsonObj, (Type) getObjectClass(classType.getAsString()));
    }

    private Class<?> getObjectClass(String classType) {
        try {
            return Class.forName(classType);
        } catch (ClassNotFoundException e) {
            throw new JsonParseException("Unable to deserialize class of type " + classType);
        }
    }
}
```

snappyf.com

La classe presenta un doppio costruttore per offrire flessibilità durante le operazioni di serializzazione e deserializzazione. Il primo costruttore accetta un oggetto Gson, fondamentale per il processo di conversione JSON. Il secondo costruttore, oltre a un oggetto Gson, consente anche di specificare un oggetto Type, risultando particolarmente utile quando si gestiscono collezioni o oggetti generic.

Al metodo “`serialize`” viene passato come parametro di input un oggetto di tipo generico e svolge un processo di conversione, trasformando l'oggetto in un JsonObject mediante il metodo “`toJsonObject()`” della libreria GSON. Un'aggiunta chiave a questo oggetto è la proprietà “`classType`” che conserva il nome della classe dell'oggetto. La rappresentazione finale dell'oggetto JsonObject viene quindi tradotta in una stringa UTF e scritta nell'oggetto “out”.

Il metodo “`deserialize`” legge la stringa UTF da un oggetto DataInput2 e la converte nuovamente in un oggetto JsonObject utilizzando il metodo “`fromJson()`” di GSON. Successivamente, utilizza la proprietà “`classType`” dall'oggetto JsonObject per identificare la classe dell'oggetto recuperato. Infine, GSON viene nuovamente impiegato per deserializzare l'oggetto JsonObject in un oggetto della classe corretta, sfruttando l'informazione sulla classe ottenuta precedentemente.

MAPDB

MapDB è stato utilizzato per il database, sfruttando le sue proprietà transazionali. Questa scelta di implementazione offre la possibilità di avere atomicità nelle operazioni di scrittura e di evitare problemi di concorrenza sul database. Le principali caratteristiche di mapDB sono la possibilità di eseguire i rollback e il garantire l'integrità del database nel caso di arresti anomali del sistema quando il database è attivo. La possibilità di eseguire rollback la si sfrutta quando vengono rilevati problemi e di conseguenza non si vogliono conservare le modifiche.

La classe DBImplements è un'implementazione concreta dell'interfaccia MapDB e costituisce un componente chiave nel sistema di gestione del database MapDB per l'applicazione CardsMule. Questa classe fornisce funzionalità per interagire con il database, includendo la gestione di mappe in memoria e persistenti su file.

```
public class DBImplements implements MapDB, MapDBConst {

    private static DB getDB(ServletContext ctx, String dbType) {
        synchronized (ctx) { // solo un thread alla volta può eseguire quel codice
            DB db = (DB) ctx.getAttribute(dbType + "_CTX_ATTRIBUTE");
            if (db == null) {
                if (dbType.equals(DB_MEMORY)) {
                    db = DBMaker.memoryDB().make();
                } else if (dbType.equals(DB_FILE)) {
                    String path= System.getenv("DB_FILE_PATH") + "/mapDB.db";
                    File dbFile = new File(path);
                    db = DBMaker.fileDB(dbFile).transactionEnable().closeOnJvmShutdown().make();
                }
                ctx.setAttribute(dbType + "_CTX_ATTRIBUTE", db);
            }
            return db;
        }
    }

    @Override
    public <K, V> Map<K, V> getCachedMap(ServletContext ctx, String mapName, Serializer<K> keySerializer,
                                             Serializer<V> valueSerializer) {
        return getDB(ctx, DB_MEMORY).hashMap(mapName, keySerializer, valueSerializer).createOrOpen();
    }

    @Override
    public <K, V> Map<K, V> getPersistentMap(ServletContext ctx, String mapName, Serializer<K> keySerializer,
                                                Serializer<V> valueSerializer) {
        return getDB(ctx, DB_FILE).hashMap(mapName, keySerializer, valueSerializer).createOrOpen();
    }

    @Override
    public <K, V, T> T writeOperation(ServletContext ctx, String mapName, Serializer<K> keySerializer,
                                       Serializer<V> valueSerializer, Function<Map<K, V>, T> operation) {
        DB db = getDB(ctx, DB_FILE);
        try {
            T value = operation.apply(db.hashMap(mapName, keySerializer, valueSerializer).createOrOpen());
            db.commit();
            return value;
        } catch (Exception e) {
            db.rollback();
            return null;
        }
    }

    @Override
    public boolean writeOperation(ServletContext ctx, Runnable runnable) {
        DB db = getDB(ctx, DB_FILE);
        try {
            runnable.run();
            db.commit();
            return true;
        } catch (Exception e) {
            db.rollback();
            return false;
        }
    }
}
```

snappyf.com

Il metodo `getDB` è un metodo privato che restituisce un'istanza del database `MapDB` in base al tipo specificato (memoria o file). Utilizza la sincronizzazione per garantire che solo un thread alla volta possa eseguire questo codice. Se il database non è stato ancora creato e memorizzato nel contesto dell'applicazione, viene istanziato e salvato per uso futuro.

Il metodo “`getCachedMap`” restituisce una mappa in memoria associata a un determinato nome. Questo metodo è utile per ottenere una mappa che risiede completamente in memoria.

Il metodo “`getPersistentMap`” restituisce una mappa persistente su file associata a un nome specifico. Questo tipo di mappa è salvato su disco e conserva i dati anche attraverso riavvio dell'applicazione, garantendo la persistenza dei dati.

Il metodo “`writeOperation`” rappresenta un'operazione di scrittura transazionale all'interno del database. Accetta una funzione che opera sulla mappa specificata e restituisce un valore. In caso di successo, la transazione viene confermata (commit), altrimenti viene eseguito un rollback.

I metodi spiegati in precedenza sono tutti dichiarati all'interno dell'interfaccia `MapDB.java` che viene implementata nella classe `MapDBImplements.java`



A screenshot of a code editor window titled "MapDBConst.java". The code defines a public interface with various string constants for map names. The interface is as follows:

```
package com.sweng.cardsmule.server.mapDB;

public interface MapDBConst {
    String DB_NAME = "DB";
    String DB_MEMORY = "db_MEMORY";
    String DB_FILE = "db_FILE";
    String MAP_MAGIC = "db_MAGIC_MAP";
    String MAP_POKEMON = "db_POKEMON_MAP";
    String MAP_YUGIOH = "db_YUGIOH_MAP";
    String MAP_DECK = "db_DECK_MAP";
    String MAP_USER = "db_USER_MAP";
    String MAP_LOGIN = "db_LOGIN_MAP";
    String MAP_PROPOSAL = "db_PROPOSAL_MAP";
    String MAP_ACCOUNT = "db_ACCOUNT_MAP";
}
```

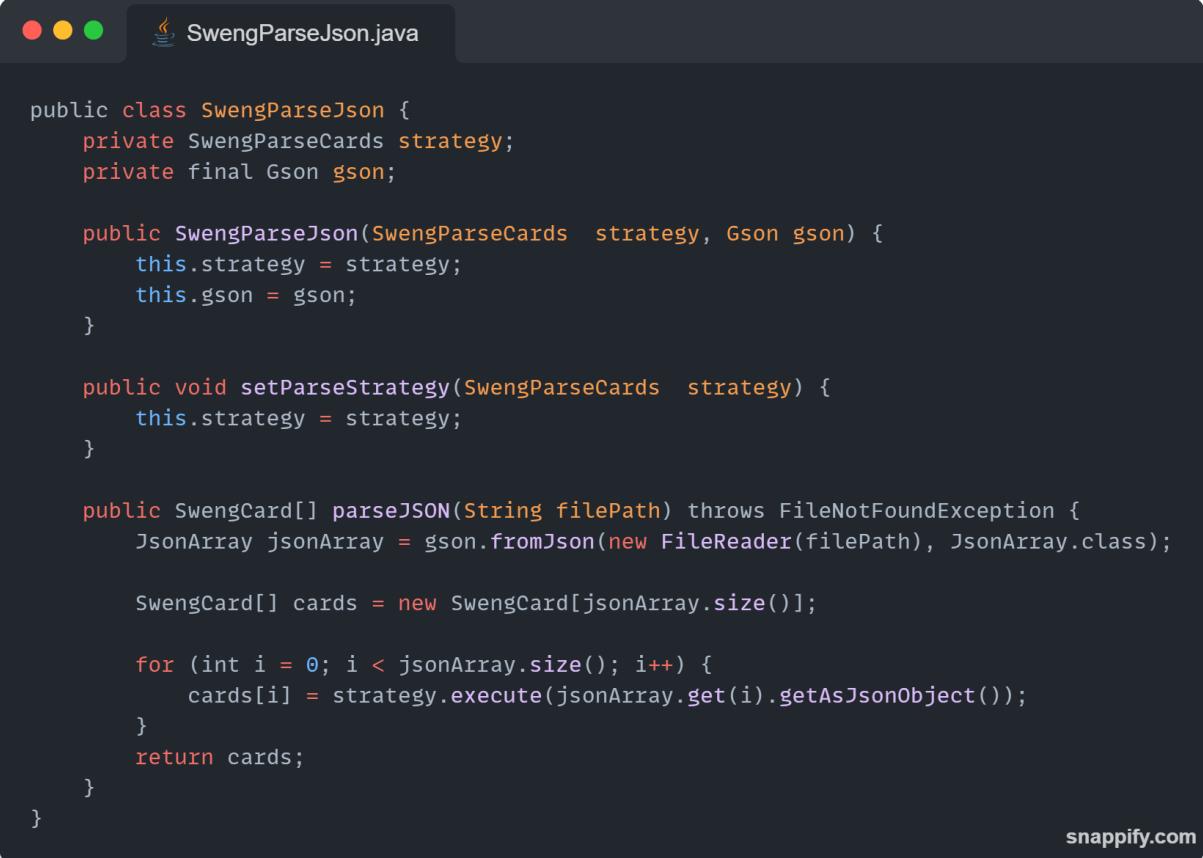
The code is displayed in a dark-themed code editor. The "snappify.com" watermark is visible at the bottom right of the code area.

Questa interfaccia implementa la dichiarazione delle mappe utilizzate per mapDB. Ogni costante è utilizzata per accedere ad una diversa mappa implementata nel database e di conseguenza utilizzata per la manipolazione di dati in base ad una chiave specifica.

JSON

La classe SwengParseJson è responsabile dell'importazione di file JSON dalle risorse e della loro conversione in una serie di oggetti Carta specifici per ogni gioco. Il costruttore della classe SwengParseJson accetta un'istanza di una classe che implementa l'interfaccia SwengParseCards, che gli consente di utilizzare un'interfaccia personalizzata execute() per ottenere un oggetto Card dall'oggetto JSON passato al parser. Richiede anche un'istanza di Gson per analizzare il file JSON.

Il metodo parseJSON() accetta un percorso di un file come argomento e utilizza GSON per analizzare il file JSON, restituendo una serie di oggetti JSON. A questo array viene passato, il metodo execute() di SwengParseCards, che restituisce un array di oggetti Card.



The screenshot shows a Java code editor window with a dark theme. The title bar says "SwengParseJson.java". The code is as follows:

```
public class SwengParseJson {
    private SwengParseCards strategy;
    private final Gson gson;

    public SwengParseJson(SwengParseCards strategy, Gson gson) {
        this.strategy = strategy;
        this.gson = gson;
    }

    public void setParseStrategy(SwengParseCards strategy) {
        this.strategy = strategy;
    }

    public SwengCard[] parseJSON(String filePath) throws FileNotFoundException {
        JSONArray jsonArray = gson.fromJson(new FileReader(filePath), JSONArray.class);

        SwengCard[] cards = new SwengCard[j jsonArray.size()];

        for (int i = 0; i < jsonArray.size(); i++) {
            cards[i] = strategy.execute(jsonArray.get(i).getAsJsonObject());
        }
        return cards;
    }
}
```

At the bottom right of the code editor, there is a watermark that says "snappify.com".

Inoltre abbiamo implementato 3 classi per i 3 differenti giochi, in ogni classe facciamo il metodo `execute()`:

Il metodo analizza l'oggetto JSON per estrarre i seguenti campi:

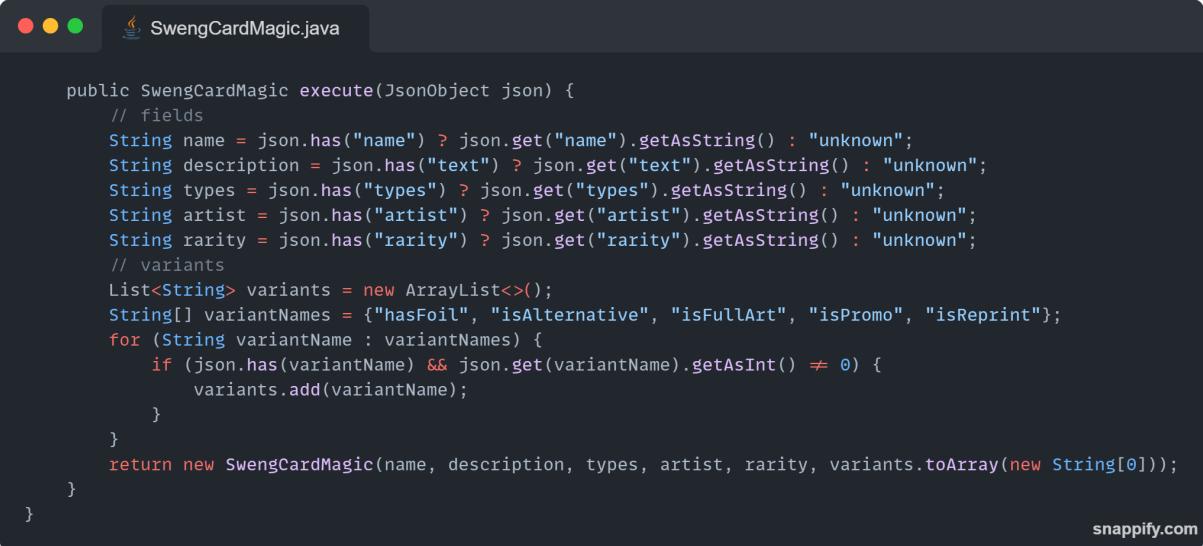
- name: Il nome della carta.
- text: La descrizione della carta.
- types: Il tipo della carta (ad esempio, creatura, incantesimo, ecc.).

- artist: Il nome dell'artista che ha illustrato la carta.
- rarity: La rarità della carta.

Viene creato un elenco (List) chiamato variants per memorizzare le varianti della carta.

In seguito viene eseguito un ciclo su questi nomi di varianti, e se un campo corrispondente è presente nell'oggetto JSON e il suo valore è diverso da zero, il nome della variante viene aggiunto all'elenco variants.

Successivamente viene creato un oggetto SwengCardMagic in questo caso utilizzando le informazioni estratte e restituito come risultato.



```

public SwengCardMagic execute(JsonObject json) {
    // fields
    String name = json.has("name") ? json.get("name").getAsString() : "unknown";
    String description = json.has("text") ? json.get("text").getAsString() : "unknown";
    String types = json.has("types") ? json.get("types").getAsString() : "unknown";
    String artist = json.has("artist") ? json.get("artist").getAsString() : "unknown";
    String rarity = json.has("rarity") ? json.get("rarity").getAsString() : "unknown";
    // variants
    List<String> variants = new ArrayList<>();
    String[] variantNames = {"hasFoil", "isAlternative", "isFullArt", "isPromo", "isReprint"};
    for (String variantName : variantNames) {
        if (json.has(variantName) && json.get(variantName).getAsInt() != 0) {
            variants.add(variantName);
        }
    }
    return new SwengCardMagic(name, description, types, artist, rarity, variants.toArray(new String[0]));
}

```

snappify.com

Al caricamento dell'applicazione, grazie alla SwengParseJson appena visto, vengono visualizzati i dati delle carte per ogni gioco caricato da risorse JSON. Questi vengono salvati in memoria subito dopo il caricamento, ma la scelta fatta in questo caso è stata quella di utilizzare un MemoryDB, cioè di salvare i dati su una memoria cache. Questo permette, durante il funzionamento dell'applicazione e l'utilizzo da parte dell'utente, il recupero dei dati in modo veloce ed efficiente, senza la necessità di contattare il database fisico eseguendo operazioni di I/O ad ogni operazione di lettura.

Le informazioni necessarie alla persistenza dei dati per il catalogo delle carte da gioco disponibili sono fornite dagli stessi file JSON presenti nelle risorse del progetto.

SwengListenerImpl

La classe ListenerImpl, è un ServletContextListener (è un'interfaccia nell'ambito della tecnologia Java Servlet che consente di ricevere notifiche quando il contesto dell'applicazione web viene inizializzato o distrutto), viene utilizzata per ascoltare l'applicazione web quando è lanciata. Il metodo contextInitialized viene chiamato all'avvio dell'applicazione web e sovrascrive quello in ServletContextListener.

Innanzitutto si inizializza il database MapDB e si creano alcune mappe per memorizzare gli oggetti Card. Si creano istanze della libreria GSON, JSONParser e le diverse strategie.



The screenshot shows a code editor window with the title "SwengListenerImpl.java". The code is written in Java and implements the ServletContextListener interface. It includes methods for initializing the context and destroying it, as well as logic for loading data from files into a MapDB database. The code uses GSON for JSON parsing and various strategies for card serialization. A watermark for "snappyify.com" is visible at the bottom right of the code editor.

```
public class SwengListenerImpl implements ServletContextListener, MapDBConst {
    private final MapDB db;
    private final String path;

    public SwengListenerImpl() {
        db = new DBImplements();
        path = "./WEB-INF/classes/Json/";
    }

    public SwengListenerImpl(MapDB db, String path) {
        this.db = db;
        this.path = path;
    }

    private void uploadDataToDB(Map<Integer, SwengCard> map, SwengCard[] cards) {
        for (SwengCard card : cards) {
            map.put(card.getId(), card);
        }
    }

    @Override
    public void contextInitialized(ServletContextEvent sce) throws RuntimeException {
        System.out.println("Context initialized.");
        System.out.println("*** Loading data from file. ***");

        Gson gson = new Gson();
        GsonSerializer<SwengCard> cardSerializer = new GsonSerializer<>(gson);

        Map<Integer, SwengCard> yuGiOhMap = db.getCachedMap(sce.getServletContext(), MAP_YUGIOH,
                Serializer.INTEGER, cardSerializer);
        Map<Integer, SwengCard> magicMap = db.getCachedMap(sce.getServletContext(), MAP_MAGIC,
                Serializer.INTEGER, cardSerializer);
        Map<Integer, SwengCard> pokemonMap = db.getCachedMap(sce.getServletContext(), MAP_POKEMON,
                Serializer.INTEGER, cardSerializer);

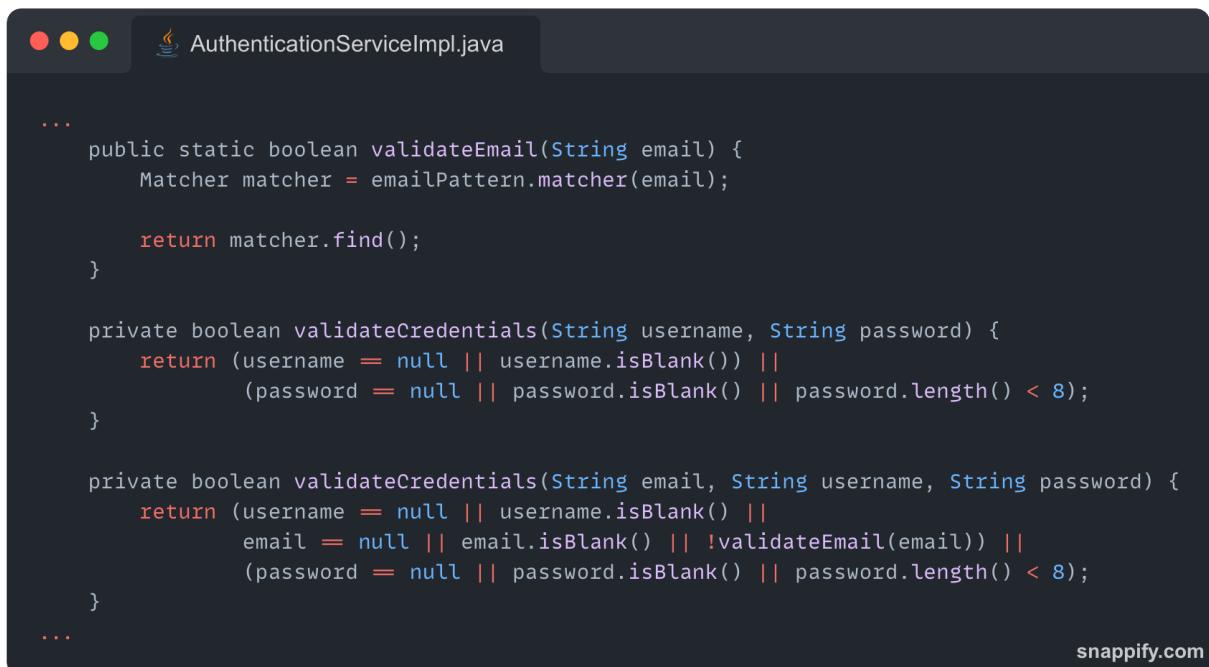
        SwengParseJson parser = new SwengParseJson(new SwengParseYiGiOhCard(), gson);
        try {
            System.out.println("Working Directory = " + System.getProperty("user.dir"));
            uploadDataToDB(yuGiOhMap, parser.parseJSON(path + "yugioh_cards.json"));
            parser.setParseStrategy(new SwengParseMagic());
            uploadDataToDB(magicMap, parser.parseJSON(path + "magic_cards.json"));
            parser.setParseStrategy(new SwengParsePokemon());
            uploadDataToDB(pokemonMap, parser.parseJSON(path + "pokemon_cards.json"));
        } catch (FileNotFoundException e) {
            System.out.println("Error");
            System.out.println(e.getMessage());
        }
        System.out.println("*** Data Loaded. ***");
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
    }
}
```

SERVICES

I services sono delle classi lato server che implementano le funzionalità dell'applicazione e si interfacciano al DB, i metodi delle classi server sono dichiarati nella sezione shared e vengono chiamati in maniera remota dal servizio RPC.

AuthenticationServiceImpl: è un servizio che implementa le operazioni necessarie per l'autenticazione dell'utente.



The screenshot shows a code editor window with a dark theme. The title bar says "AuthenticationServiceImpl.java". The code itself is as follows:

```
...
public static boolean validateEmail(String email) {
    Matcher matcher = emailPattern.matcher(email);

    return matcher.find();
}

private boolean validateCredentials(String username, String password) {
    return (username == null || username.isBlank()) ||
           (password == null || password.isBlank() || password.length() < 8);
}

private boolean validateCredentials(String email, String username, String password) {
    return (username == null || username.isBlank() ||
            email == null || email.isBlank() || !validateEmail(email)) ||
           (password == null || password.isBlank() || password.length() < 8);
}
...

```

snappyf.com

Questi tre metodi validate hanno lo scopo di validare le credenziali inserite dall'utente e mandate al server. In particolare il metodo `validateEmail()` ha lo scopo di controllare che l'email inserita dall'utente rispetti la regular expression del pattern di un classico indirizzo email, nome@dominio.it. Abbiamo un overload del metodo `validateCredentials()`, la prima dichiarazione del metodo è utilizzata nel login, dove l'utente inserisce solamente lo username e la password, la seconda dichiarazione del metodo, come suggeriscono i tre parametri della firma del metodo, viene utilizzata nella registrazione dove l'utente inserisce oltre che l'username e la password, anche l'email.

```
...  
private static String generateHash(String userEmail) {  
    try {  
        MessageDigest digest = MessageDigest.getInstance("SHA-256");  
        byte[] encodedHash = digest.digest((userEmail + SECRET).getBytes());  
        return bytesToHex(encodedHash);  
    } catch (NoSuchAlgorithmException e) {  
        throw new RuntimeException(e);  
    }  
}  
  
private static String bytesToHex(byte[] hash) {  
    StringBuilder hexString = new StringBuilder();  
    for (byte b : hash) {  
        String hex = Integer.toHexString(0xff & b);  
        if (hex.length() == 1) hexString.append('0');  
        hexString.append(hex);  
    }  
    return hexString.toString();  
}  
  
private String generateAndStoreLoginToken(Account account) {  
    String token = generateHash(account.getEmail());  
  
    db.writeOperation(getServletContext(), MAP_LOGIN, Serializer.STRING, new GsonSerializer<>(gson),  
        (Map<String, LoginSession> loginMap) -> {  
            loginMap.put(token, new LoginSession(account.getUsername(), account.getEmail(), System.currentTimeMillis()));  
            return null;  
        });  
    return token;  
}  
...
```

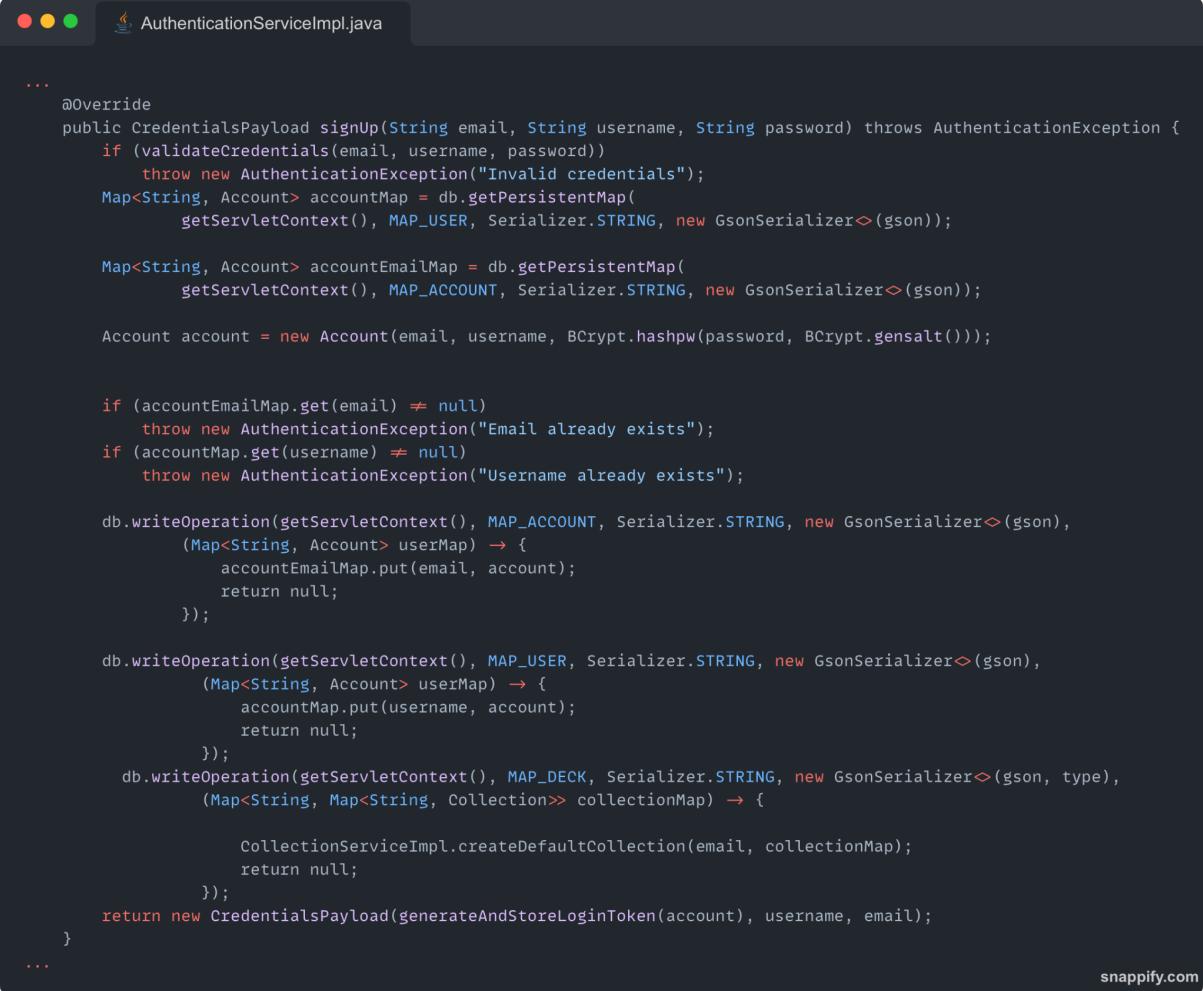
snappyf.com

I metodi per la creazione e lo storage del token per l'autenticazione sono tre. Il primo è `generateHash()` il quale si preoccupa di generare di fatto il token partendo dall'email dell'utente come input e generando l'hash di quest'ultima, per questo motivo utilizza un il metodo `bytesToHex()` il quale nome è abbastanza esplicativo per il metodo stesso. Il metodo `generateHash()` viene richiamato dal metodo `generateAndStoreLoginToken()` il quale oltre a richiamare il metodo `generateHash()` ne prende il risultato e lo mette in storage nel dizionario di mapdb contenente tutti i token come chiavi mentre tutte gli oggetti `LoginSession` come valori. Il nome del dizionario è accessibile attraverso l'enum delle costanti di mapdb, `MAP_LOGIN`.

```
...  
private static boolean checkTokenExpiration(long loginTime) {  
    final long EXPIRATION_TIME = 1000 * 60 * 60 * 24 * 7;  
    return loginTime > System.currentTimeMillis() - EXPIRATION_TIME;  
}  
  
public static String checkTokenValidity(String token, Map<String, LoginSession> loginMap) throws AuthenticationException {  
    if (token == null) {  
        throw new AuthenticationException("Invalid token");  
    }  
    LoginSession session = loginMap.get(token);  
    if (session == null) {  
        throw new AuthenticationException("Invalid token");  
    } else if (!checkTokenExpiration(session.getLoginTime())) {  
        throw new AuthenticationException("Expired token");  
    }  
    return session.getEmail();  
}  
  
@Override  
public String me(String token) throws AuthenticationException {  
    return checkTokenValidity(token, db.getPersistentMap(getServletContext(), MAP_LOGIN, Serializer.STRING, new GsonSerializer<>(gson)));  
}  
...
```

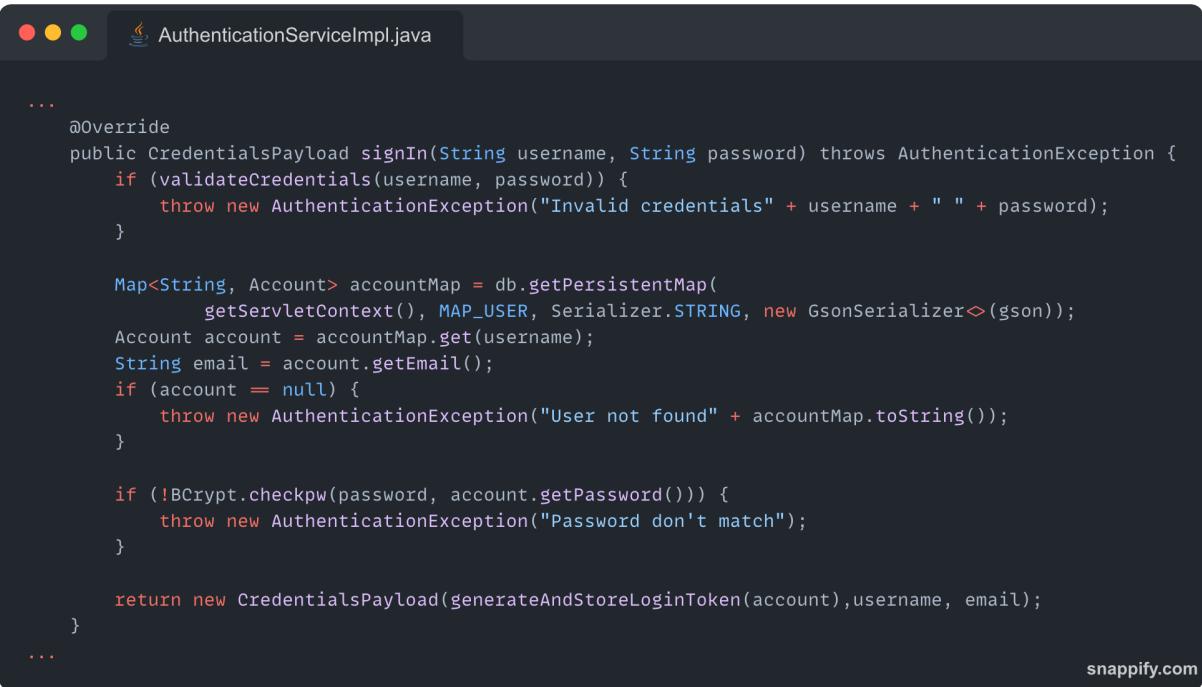
snappyf.com

I metodi che gestiscono la validazione del token sono tre. `checkTokenExpiration()` non fa altro che controllare che il `loginTime` del token non abbia superato l'expiration time. Il metodo `checkTokenExpiration()` viene poi richiamato dal metodo `checkTokenValidity()` che oltre a fare dei controlli ulteriori sul token ottiene l'oggetto `LoginSession` dal dizionario dei login di mapdb. Infine questo metodo viene chiamato da un metodo wrapper chiamato `me()`.



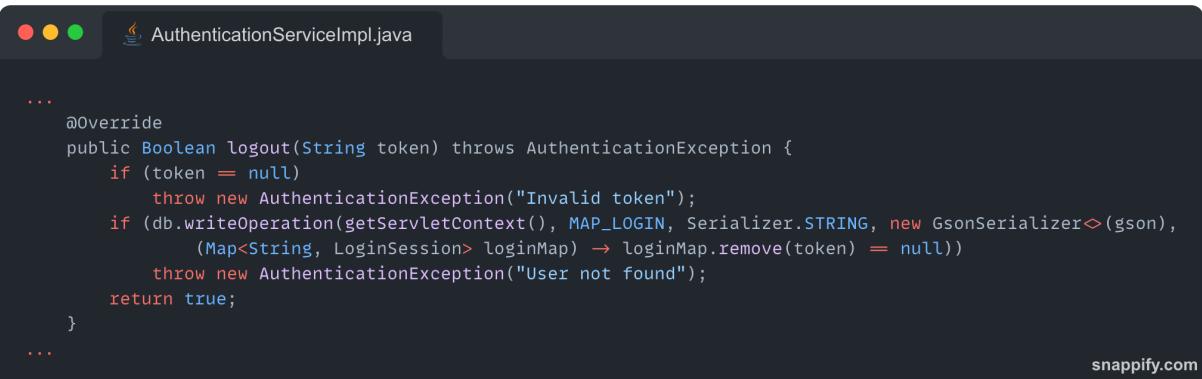
```
...  
    @Override  
    public CredentialsPayload signUp(String email, String username, String password) throws AuthenticationException {  
        if (!validateCredentials(email, username, password))  
            throw new AuthenticationException("Invalid credentials");  
        Map<String, Account> accountMap = db.getPersistentMap(  
            getServletContext(), MAP_USER, Serializer.STRING, new GsonSerializer<>(gson));  
  
        Map<String, Account> accountEmailMap = db.getPersistentMap(  
            getServletContext(), MAP_ACCOUNT, Serializer.STRING, new GsonSerializer<>(gson));  
  
        Account account = new Account(email, username, BCrypt.hashpw(password, BCrypt.gensalt()));  
  
        if (accountEmailMap.get(email) != null)  
            throw new AuthenticationException("Email already exists");  
        if (accountMap.get(username) != null)  
            throw new AuthenticationException("Username already exists");  
  
        db.writeOperation(getServletContext(), MAP_ACCOUNT, Serializer.STRING, new GsonSerializer<>(gson),  
            (Map<String, Account> userMap) -> {  
                accountEmailMap.put(email, account);  
                return null;  
            });  
  
        db.writeOperation(getServletContext(), MAP_USER, Serializer.STRING, new GsonSerializer<>(gson),  
            (Map<String, Account> userMap) -> {  
                accountMap.put(username, account);  
                return null;  
            });  
        db.writeOperation(getServletContext(), MAP_DECK, Serializer.STRING, new GsonSerializer<>(gson, type),  
            (Map<String, Map<String, Collection>> collectionMap) -> {  
  
                CollectionServiceImpl.createDefaultCollection(email, collectionMap);  
                return null;  
            });  
        return new CredentialsPayload(generateAndStoreLoginToken(account), username, email);  
    }  
...  
snappify.com
```

Il metodo `signUp()` è utilizzato per la registrazione dell'utente. Questo metodo si occupa di richiamare i metodi di validazione descritti in precedenza e lanciare, nel caso, un'eccezione. Il metodo inoltre controlla che l'email non sia già presente nel sistema, attraverso il dizionario `MAP_ACCOUNT` che ha come chiavi le email degli utenti registrati. Il metodo infine, se non lancia nessuna eccezione, scrive le informazioni del nuovo utente sui vari dizionari di mapdb. Il metodo ritorna poi un `CredentialsPayload` il quale viene poi utilizzato lato client come spiegato nell'apposita sezione della documentazione.



```
...  
    @Override  
    public CredentialsPayload signIn(String username, String password) throws AuthenticationException {  
        if (validateCredentials(username, password)) {  
            throw new AuthenticationException("Invalid credentials" + username + " " + password);  
        }  
  
        Map<String, Account> accountMap = db.getPersistentMap(  
            getServletContext(), MAP_USER, Serializer.STRING, new GsonSerializer<>(gson));  
        Account account = accountMap.get(username);  
        String email = account.getEmail();  
        if (account == null) {  
            throw new AuthenticationException("User not found" + accountMap.toString());  
        }  
  
        if (!BCrypt.checkpw(password, account.getPassword())) {  
            throw new AuthenticationException("Password don't match");  
        }  
  
        return new CredentialsPayload(generateAndStoreLoginToken(account), username, email);  
    }  
...  
snappyf.com
```

Il metodo `signIn()` il quale viene utilizzato per il login dell'utente compie azioni analoghe in parte a quelle del metodo `signUp()` se non fosse che valida un campo in meno (l'email) e non ha azioni di writing sui dizionari.



```
...  
    @Override  
    public Boolean logout(String token) throws AuthenticationException {  
        if (token == null)  
            throw new AuthenticationException("Invalid token");  
        if (db.writeOperation(getServletContext(), MAP_LOGIN, Serializer.STRING, new GsonSerializer<>(gson),  
            (Map<String, LoginSession> loginMap) → loginMap.remove(token) == null))  
            throw new AuthenticationException("User not found");  
        return true;  
    }  
...  
snappyf.com
```

Infine, il metodo `logout()` rimuove la entry che ha come chiave il token nel dizionario `MAP_LOGIN`.

CardServiceImpl: è un servizio che implementa diverse operazioni sulle carte, sul dettaglio di esse e permette di recuperare informazioni su di esse.

```
public List<SwengCard> getGameCards(CardsmuleGame game) throws InputException {
```

snappyf.com

`getGameCards (CardsmuleGame game)`: questo metodo recupera le carte dal database associato al gioco, serializza i dati e li restituisce al client. Viene preso in input un parametro che è `game`, che è un enum e deve essere uno dei valori "Magic", "Pokemon" o "Yugioh" e restituisce in caso di successo una lista di oggetti `SwengCard` che rappresentano le carte del gioco specificato.

```
public SwengCard getGameCard(CardsmuleGame game, int cardId) throws InputException {
```

snappyf.com

`getGameCard (CardsmuleGame game, int cardId)`: questo metodo cerca la carta nel database associato al gioco e la restituisce. Ci sono due parametri di input, un parametro `game` che è un enum e deve essere uno dei valori "Magic", "Pokemon" o "Yugioh" e un parametro `cardId` che rappresenta un numero intero ovvero l'identificazione della carta. Il metodo restituisce un oggetto `SwengCard` specifico identificato da `cardId`.

CollectionServiceImpl: questo servizio gestisce l'interazione con le collezioni di carte degli utenti, consentendo loro di aggiungere, rimuovere, modificare e visualizzare le carte nelle proprie collezioni e deck.

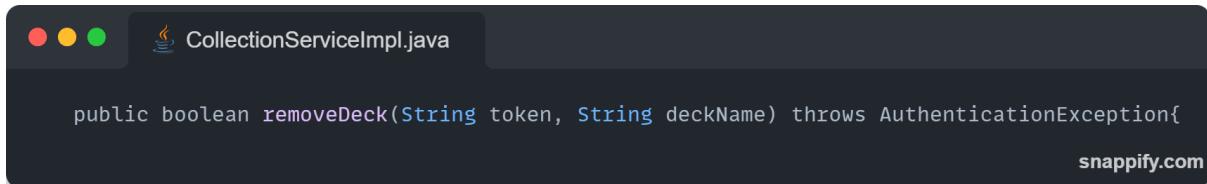
```
public boolean addCollection(String token, String collectionName) throws AuthenticationException {
```

snappyf.com

`addCollection (String token, String collectionName)` : consente agli utenti di aggiungere una nuova collezione. Questa operazione richiede un token di autenticazione per verificare l'identità dell'utente tramite l'email.

Successivamente, il metodo estrae la mappa della collection dell'utente dal DB e controlla se la collection esiste già. Nel caso esistesse restituisce false. Se invece

quella collection non esiste ancora per quell'utente specifico, la si crea e si restituisce true.



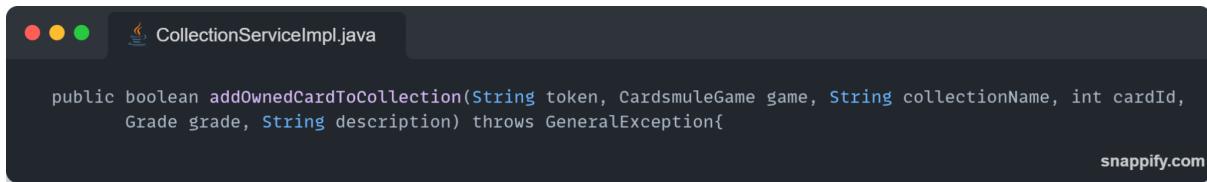
```
CollectionServiceImpl.java

public boolean removeDeck(String token, String deckName) throws AuthenticationException{
```

snappify.com

`removeDeck(String token, String deckName)`: consente agli utenti di rimuovere un mazzo di carte esistente. Questa operazione richiede un token di autenticazione per verificare l'identità dell'utente tramite l'email.

Successivamente, il metodo si occupa di verificare la consistenza del parametro di input, legge dal db la mappa dei deck dell'utente e rimuove il mazzo specificato nella richiesta. Infine, il metodo restituisce un true se la richiesta ha avuto successo, false se il mazzo non è stato trovato o non esiste una mappa dei mazzi utente, oppure un'eccezione nel caso ci fossero problemi con la rimozione del deck dal db.



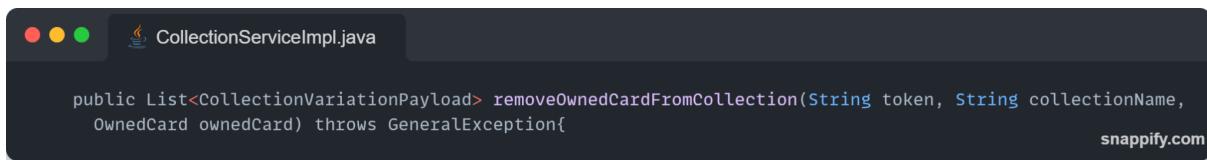
```
CollectionServiceImpl.java

public boolean addOwnedCardToCollection(String token, CardsmuleGame game, String collectionName, int cardId, Grade grade, String description) throws GeneralException{
```

snappify.com

`addOwnedCardToCollection(String token, CardsmuleGame game, String collectionName, int cardId, Grade grade, String description)`: consente agli utenti di aggiungere una carta di loro proprietà alla collezione. Questo metodo richiede il token di autenticazione, il tipo di gioco, il nome della collezione, l'ID della carta, lo stato e la descrizione della carta.

Il metodo ottiene innanzitutto l'e-mail dell'utente dal token, contemporaneamente alla convalida del token. Esso quindi controlla la coerenza dei parametri di input. Dopodichè utilizza l'e-mail per identificare la collection dell'utente specifico e inserisce la carta con l'id indicato nel mazzo. Insieme a questa operazione inserisce anche i parametri obbligatori per la carta fisica, come stato e descrizione. Se l'operazione di aggiunta delle carte al mazzo ha esito positivo, viene restituito il valore true, in caso di fallimento viene restituito il valore false.



```
CollectionServiceImpl.java

public List<CollectionVariationPayload> removeOwnedCardFromCollection(String token, String collectionName, OwnedCard ownedCard) throws GeneralException{
```

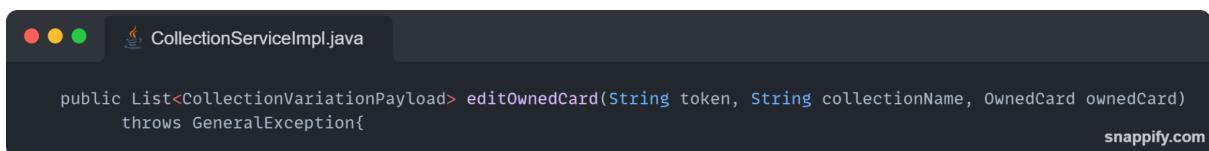
snappify.com

```
removeOwnedCardFromCollection(String token, String collectionName, OwnedCard ownedCard)
```

: consente agli utenti di rimuovere una carta da una collezione. Questo metodo richiede il token di autenticazione, il nome della collezione e la carta da rimuovere.

Il metodo ottiene innanzitutto l'e-mail dell'utente dal token, contemporaneamente alla convalida del token.

Successivamente si identifica il mazzo dell'utente dal quale si vuole rimuovere la carta e si procede alla sua rimozione. Se l'operazione di rimozione della carta dal mazzo ha successo, viene restituito un valore vero, in caso di fallimento il valore viene restituito false.



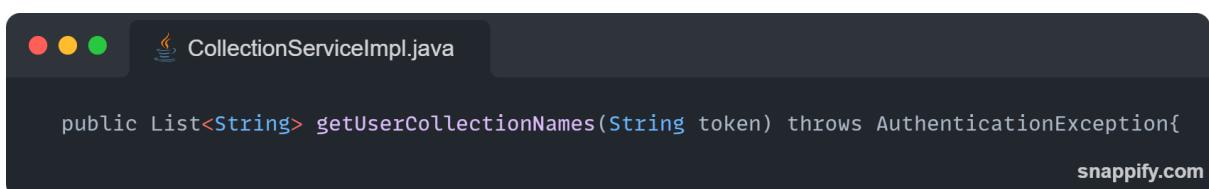
```
CollectionServiceImpl.java
```

```
public List<CollectionVariationPayload> editOwnedCard(String token, String collectionName, OwnedCard ownedCard)
throws GeneralException{}
```

snappyf.com

```
editOwnedCard(String token, String collectionName, OwnedCard ownedCard)
```

: consente agli utenti di modificare una carta della propria collezione. Questo metodo richiede il token di autenticazione, il nome della collezione e la carta da modificare. Il metodo ottiene innanzitutto l'e-mail dell'utente dal token, contemporaneamente alla convalida del token. Esso quindi verifica i parametri di input, in particolare chiama un metodo privato per verificare collectionName, verifica che il mazzo nella richiesta è un mazzo predefinito ("Posseduto" o "Desiderato"), che la carta fisica consistente e infine che la carta fisica non è presente in una proposta. Se tutti i controlli hanno successo, si occupa di modificare le proprietà della carta, rimuovendo prima la carta dal mazzo e reinserirendo quella appena modificata, infine restituisce l'elenco dei mazzi modificati.



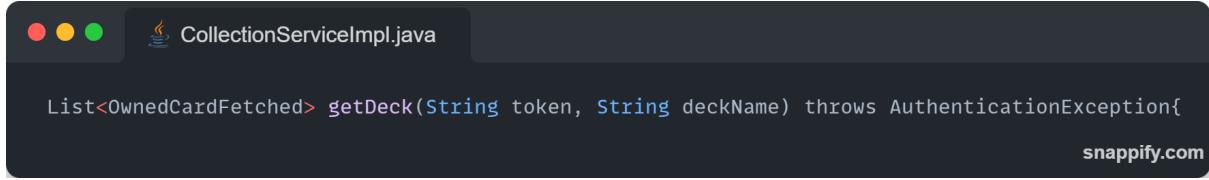
```
CollectionServiceImpl.java
```

```
public List<String> getUserCollectionNames(String token) throws AuthenticationException{}
```

snappyf.com

```
getUserCollectionNames(String token)
```

: il metodo utilizza innanzitutto la mappa di accesso per estrarre l'e-mail dell'utente dal token, controllandone al contempo validità. Successivamente, il metodo estrae dalla mappa della collection del DB l'elenco dei nomi di tutti i deck di quell'utente specifico e li restituisce come risposta.



```
List<OwnedCardFetched> getDeck(String token, String deckName) throws AuthenticationException{
```

snappyf.com

`getDeck(String token, String deckName)`: restituisce tutte le carte di un mazzo specifico dell'utente.

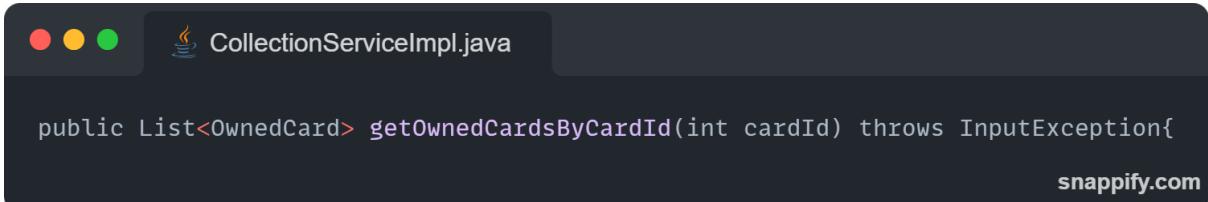
Ci sono due parametri: token, che viene utilizzato per convalidare la richiesta ed estrarre l'e-mail, e deckName per filtrare attraverso i mazzi dell'utente e trovare quello richiesto. Viene restituitoun elenco di oggetti OwnedCard in caso di successo.



```
public List<OwnedCardFetched> getUserCollection(String email) throws AuthenticationException{
```

snappyf.com

`getUserCollection(String email)`: restituisce tutte le carte nella collezione di un utente specifico.

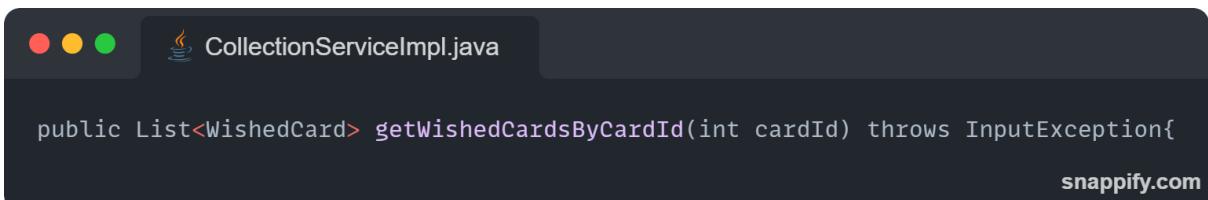


```
public List<OwnedCard> getOwnedCardsByCardId(int cardId) throws InputException{
```

snappyf.com

`getOwnedCardsByCardId(int cardId)`: restituisce le carte di proprietà dell'utente in base all'ID della carta specificata.

Come prima operazione si estrae tutte le carte contenute in uno specifico mazzo indicato e poi le relative carte che corrispondono all'id passato in input. Si restituisce un elenco di OwnedCards, accompagnato dall'email dell'utente che ne è proprietario.

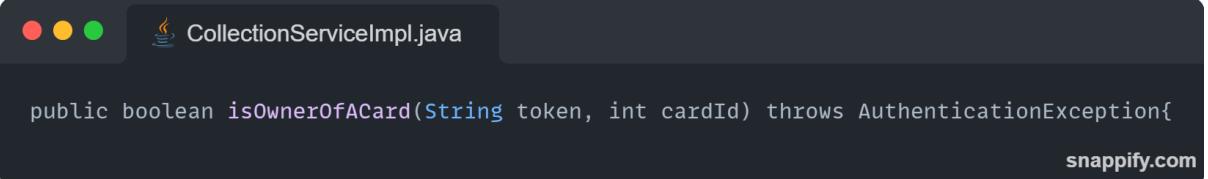


```
public List<WishedCard> getWishedCardsByCardId(int cardId) throws InputException{
```

snappyf.com

`getWishedCardsByCardId(int cardId)`: restituiscono le carte desiderate dell'utente in base all'ID della carta specificato.

Come prima operazione si estraе tutte le carte contenute in uno specifico mazzo indicato e poi le relative carte che corrispondono all'id passato in input. Restituisce quindi un elenco di OwnedCards, accompagnato dall' email dell'utente che le possiede.



```
CollectionServiceImpl.java

public boolean isOwnerOfACard(String token, int cardId) throws AuthenticationException{
```

snappyf.com

`isOwnerOfACard(String token, int cardId)`: verifica se l'utente identificato dal token di autenticazione possiede una specifica carta di gioco.



```
CollectionServiceImpl.java

public List<OwnedCardFetched> addOwnedCardsToDeck(String token, String deckName, List<OwnedCard> ownedCards)
throws GeneralException{
```

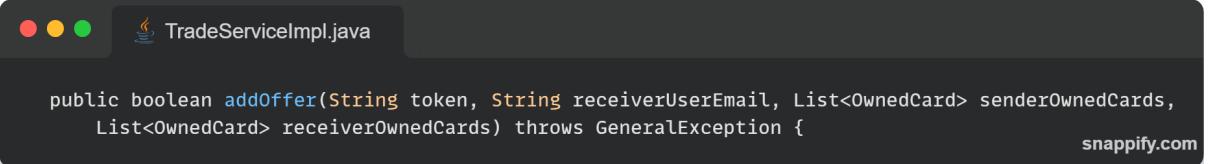
snappyf.com

`addOwnedCardsToDeck(String token, String deckName, List<OwnedCard> ownedCards)`: consente agli utenti di aggiungere carte a un mazzo specifico.

Il metodo inizia verificando la validità del token fornito che restituisce l'e-mail dell'utente associato al token.

Si scorre attraverso l'elenco delle carte e le si inserisce nel mazzo indicato, restituendo un elenco di carte con nome.

TradeServiceImpl: fornisce un'implementazione server-side delle funzionalità di gestione degli scambi di carte.



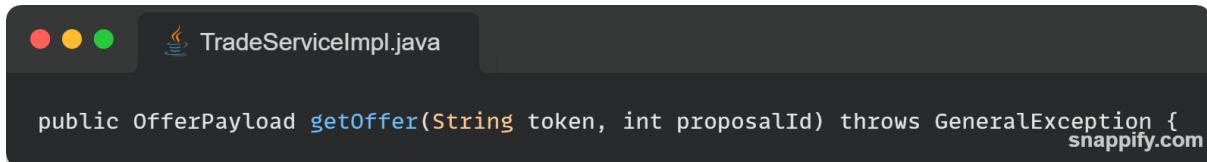
```
TradeServiceImpl.java

public boolean addOffer(String token, String receiverUserEmail, List<OwnedCard> senderOwnedCards,
List<OwnedCard> receiverOwnedCards) throws GeneralException {
```

snappyf.com

`addOffer()`: questo metodo gestisce l'aggiunta di una nuova offerta di scambio tra due utenti. Verifica la validità del token di autenticazione, tramite `checkTokenValidity()`, successivamente implementa una serie di controlli: sulla validità della mail del destinatario dell'offerta, se l'email del destinatario è

uguale a quella del mittente, se il mittente ha almeno una carta da scambiare e infine se il destinatario ha delle carte possedute da poter scambiare. Crea un nuovo oggetto Offer, al cui interno vengono inseriti tutti i dati dell'offerta di scambio e infine lo inserisce nella mappa persistente delle offerte, così da poter salvare la richiesta.



```
TradeServiceImpl.java

public OfferPayload getOffer(String token, int proposalId) throws GeneralException {
```

`getOffer()`: questo metodo recupera i dettagli di un'offerta specifica identificata da un ID proposta. Verifica la validità del token, tramite `checkTokenValidity()`, e se l'utente è coinvolto in un'offerta. Restituisce infine un nuovo oggetto OfferPayload al quale vengono inserite le varie caratteristiche dell'offerta comprese le email dei soggetti in causa.



```
TradeServiceImpl.java

public boolean acceptOffer(String token, int offerId)
    throws AuthenticationException, OfferNotFoundException {
```

`acceptOffer()`: il metodo in questione gestisce l'accettazione di un'offerta specifica. Verifica il token di autenticazione, tramite `checkTokenValidity()`, e il coinvolgimento dell'utente autenticato come destinatario dell'offerta. Aggiorna le collezioni di carte degli utenti coinvolti nell'offerta accettata, tramite due put in cui si aggiornano le carte dei rispettivi utenti e rimuove l'offerta dalla mappa persistente delle offerte.

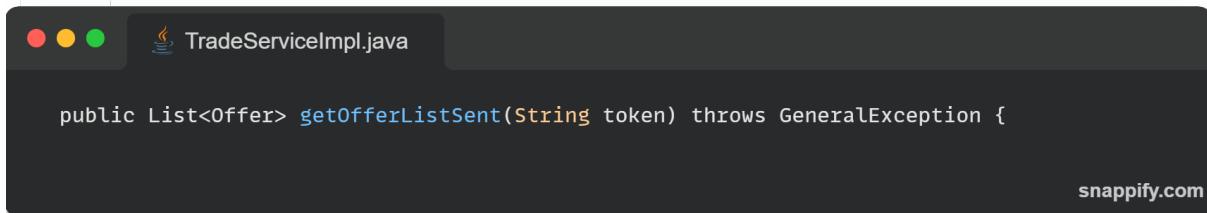


```
TradeServiceImpl.java

public List<Offer> getOfferListReceived(String token) throws GeneralException {
```

`getOfferListReceived()`: il metodo restituisce una lista di offerte ricevute dall'utente autenticato. Filtra le offerte in base alla mail dell'utente autenticato confrontandola con tutte le email di destinatari della mappa che gestisce le offerte,

per ogni email trovata aggiunge ad una lista l'offerta e alla fine del metodo ritorna la lista.



A screenshot of a Java code editor window titled "TradeServiceImpl.java". The code shown is:

```
public List<Offer> getOfferListSent(String token) throws GeneralException {
```

The window has a dark theme with red, yellow, and green window control buttons. The status bar at the bottom right shows "snappyf.com".

`getOfferListSent()`: il metodo restituisce una lista di offerte inviate dall'utente autenticato. Filtra le offerte in base alla mail dell'utente autenticato come mittente con tutte le mail di utenti che hanno inviato offerte di scambio, questo grazie ai valori all'interno della mappa che gestisce le offerte, come per il metodo precedente ogni offerta trovata la si aggiunge ad una lista e infine la si ritorna.



A screenshot of a Java code editor window titled "TradeServiceImpl.java". The code shown is:

```
public boolean refuseOrWithdrawOffer(String token, int offerId) throws GeneralException {
```

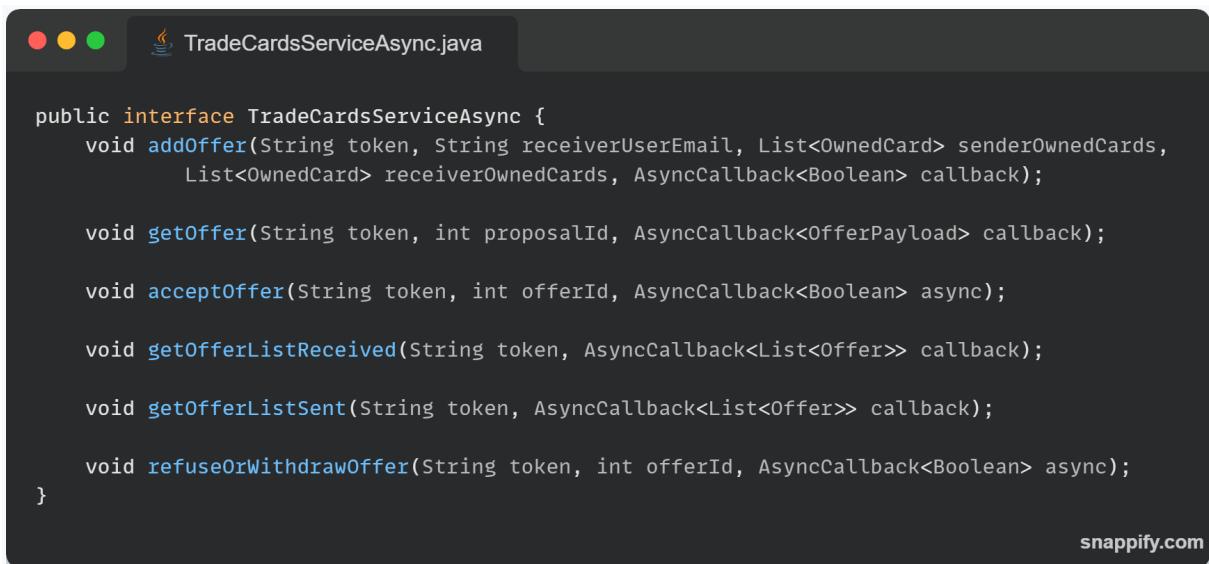
The window has a dark theme with red, yellow, and green window control buttons. The status bar at the bottom right shows "snappyf.com".

`refuseOrWithdrawOffer()`: il metodo gestisce il rifiuto o il ritiro di un'offerta specifica. Attraverso l'id passato come parametro di input al metodo si ricava l'offerta che si vuole eliminare, si attuano poi due controlli: il primo è se l'oggetto che contiene l'offerta di scambio è nullo, in tal caso si lancia un'eccezione per mostrare all'utente che l'offerta non esiste, il secondo controllo invece lo si fa sull'email dell'utente per verificare se è effettivamente mittente o destinatario dell'offerta di scambio. Superati i controlli si rimuove l'offerta dalla mappa persistente delle offerte.

SHARED

- Models
- Throwables

Al di fuori di questi due package sono dichiarati nella parte Shared una serie di interfacce molte delle quali sono asincrone (servizio RPC), queste interfacce sono utilizzate nelle varie classi della parte activity per poter effettuare chiamate al server in maniera asincrona, quindi senza bloccare l'interfaccia utente mentre si attende la risposta dal server.



The screenshot shows a Java code editor window with a dark theme. The title bar says "TradeCardsServiceAsync.java". The code defines an interface with several methods:

```
public interface TradeCardsServiceAsync {
    void addOffer(String token, String receiverUserEmail, List<OwnedCard> senderOwnedCards,
                  List<OwnedCard> receiverOwnedCards, AsyncCallback<Boolean> callback);

    void getOffer(String token, int proposalId, AsyncCallback<OfferPayload> callback);

    void acceptOffer(String token, int offerId, AsyncCallback<Boolean> async);

    void getOfferListReceived(String token, AsyncCallback<List<Offer>> callback);

    void getOfferListSent(String token, AsyncCallback<List<Offer>> callback);

    void refuseOrWithdrawOffer(String token, int offerId, AsyncCallback<Boolean> async);
}
```

In the bottom right corner of the code editor, there is a watermark that says "snappify.com".

Ogni metodo ha un parametro AsyncCallback che sarà richiamato quando l'operazione asincrona invocata sarà completata, in questo caso se l'operazione è stata eseguita con successo verrà eseguito il codice all'interno del metodo onSuccess dell'interfaccia AsyncCallback, in caso contrario verrà eseguito il metodo onFailure.

In più in al di fuori dei due package sono stati implementati anche i payloads, sono utilizzati per semplificare il trasferimento di informazioni tra diverse parti dell'applicazione, garantendo che solo ciò che è strettamente necessario venga inviato, migliorando così l'efficienza del sistema.

Quindi al fine di dover ogni volta inviare dati al client strutturati in maniera diversa utilizziamo i payloads, così da ridurre anche al minimo l'utilizzo delle operazioni sul database.

I payloads sono stati implementati mantenendo la struttura della classe originale aggiungendo i dati che servono per il contesto in cui verrà utilizzato il payload.



The screenshot shows a Java code editor window titled "OfferPayload.java". The code defines a class "OfferPayload" that implements the "Serializable" interface. It contains private fields for sender and receiver emails, and lists of owned cards. A constructor takes these parameters and initializes the fields. It also includes methods to get the sender and receiver emails, and lists of their respective cards.

```
public class OfferPayload implements Serializable {
    private String senderEmail;
    private String receiverEmail;
    private List<OwnedCardFetched> senderCards;
    private List<OwnedCardFetched> receiverCards;

    public OfferPayload(String senderEmail, String receiverEmail,
        List<OwnedCardFetched> senderCards, List<OwnedCardFetched> receiverCards) {
        this.senderEmail = senderEmail;
        this.receiverEmail = receiverEmail;
        this.senderCards = senderCards;
        this.receiverCards = receiverCards;
    }

    public OfferPayload() {
    }

    public String getSenderEmail() {
        return senderEmail;
    }

    public String getReceiverEmail() {
        return receiverEmail;
    }

    public List<OwnedCardFetched> getSenderCards() {
        return senderCards;
    }

    public List<OwnedCardFetched> getReceiverCards() {
        return receiverCards;
    }
}
```

snappify.com

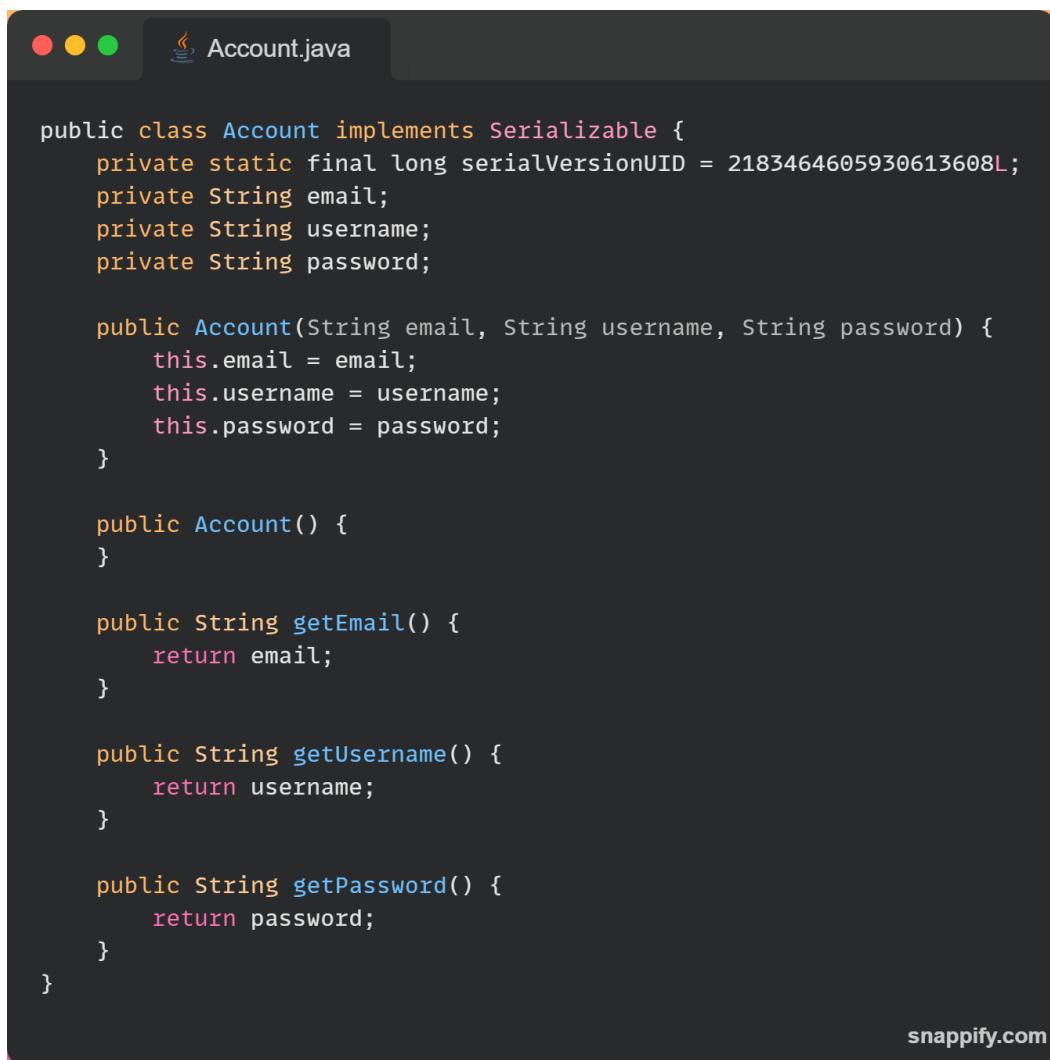
Nel caso di OfferPayload.java vengono personalizzati i dati dell'offerta, eliminando l'id dell'offerta di scambio e anche la relativa data di proposta. Questa omissione di dati è dovuta al fatto che per la generazione della trade, nella classe TradeServiceImpl.java, i dati da adoperare sono: l'email dell'utente sender, l'email dell'utente receiver e le loro rispettive liste di carte possedute per la trade.

MODELS

Le classi all'interno del package models seguono un approccio di progettazione orientato agli oggetti per rappresentare concetti chiave nell'applicazione. Ogni classe è progettata per modellare un aspetto specifico del dominio dell'applicazione, come utenti, carte, offerte di scambio, mazzi, ecc...

Queste classi "models" svolgono un ruolo centrale nell'architettura dell'applicazione, fornendo un modo coeso di rappresentare e manipolare i dati fondamentali. Sono progettate per essere facilmente scambiate tra il client e il server, supportando le operazioni di comunicazione e contribuendo alla coerenza dei dati nel sistema.

Eccone un esempio:

A screenshot of a code editor window titled "Account.java". The code is written in Java and defines a class "Account" that implements the "Serializable" interface. The class has four private fields: "email", "username", and "password", and a static final long variable "serialVersionUID". It includes a constructor that takes three parameters and initializes the fields, a no-argument constructor, and three getter methods for each field. The code is color-coded with syntax highlighting.

```
public class Account implements Serializable {
    private static final long serialVersionUID = 2183464605930613608L;
    private String email;
    private String username;
    private String password;

    public Account(String email, String username, String password) {
        this.email = email;
        this.username = username;
        this.password = password;
    }

    public Account() {
    }

    public String getEmail() {
        return email;
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }
}
```

snappify.com

In questo esempio si mostra la classe Account che rappresenta un singolo account utente all'interno dell'applicazione. Un account è caratterizzato da un indirizzo email univoco, un nome utente distintivo e una password segreta.

Grazie ai metodi dichiarati è possibile utilizzare le informazioni dell'account sia lato server, sia lato client.

Per ogni concetto chiave dell'applicazione abbiamo creato la rispettiva classe all'interno del package “models”, questi sono i concetti chiave da noi identificati:

- Gioco di carte: CardsmuleGame.java.
- Collezione di carte: Collection.java.
- Stato fisico della carta: Grade.java.
- Offerta di scambio: Offer.java.
- Carta: SwengCard.java (classe astratta)
- Carta di Pokemon: SwengPokemonCard.java.
- Carta di YuGiOh: SwengYuGiOhCard.java.
- Carta di Magic: SwengCardMagic.java.
- Carta posseduta da un utente: OwnedCard.java
- Carta desiderata: WishedCard.java

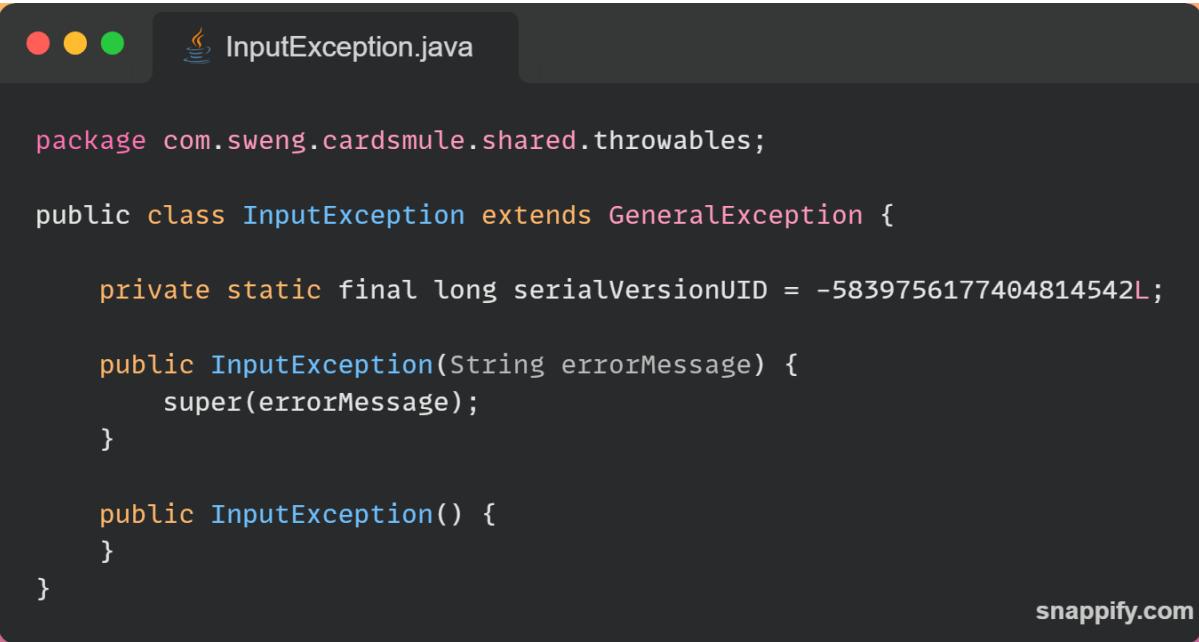
THROWABLES

In questo package vengono implementate classi che rappresentano una eccezione personalizzata (throwable) relativa a errori legati agli input nell'applicazione.

Le eccezioni personalizzate consentono di comunicare in modo chiaro agli utenti specifici tipi di errori. I messaggi di errore associati a ciascuna eccezione possono essere presentati direttamente agli utenti, facilitando la comprensione del problema.

La struttura modulare delle eccezioni personalizzate rende il sistema facilmente estensibile. Nuovi tipi di eccezioni possono essere aggiunti senza impattare negativamente la gestione degli errori esistente.

In conclusione, il package throwables rappresenta un'implementazione mirata alla gestione degli errori, migliorando la robustezza dell'applicazione e offrendo all'utente una navigazione più trasparente e di facile comprensione.



```
package com.sweng.cardsmule.shared.throwables;

public class InputException extends GeneralException {

    private static final long serialVersionUID = -5839756177404814542L;

    public InputException(String errorMessage) {
        super(errorMessage);
    }

    public InputException() {
    }
}
```

snappify.com

La classe InputException.java rappresenta un'eccezione dovuta ad input non validi o scorretti. Questa eccezione può essere utilizzata quando si verificano errori dovuti a dati inseriti dall'utente che non soddisfano i requisiti previsti.

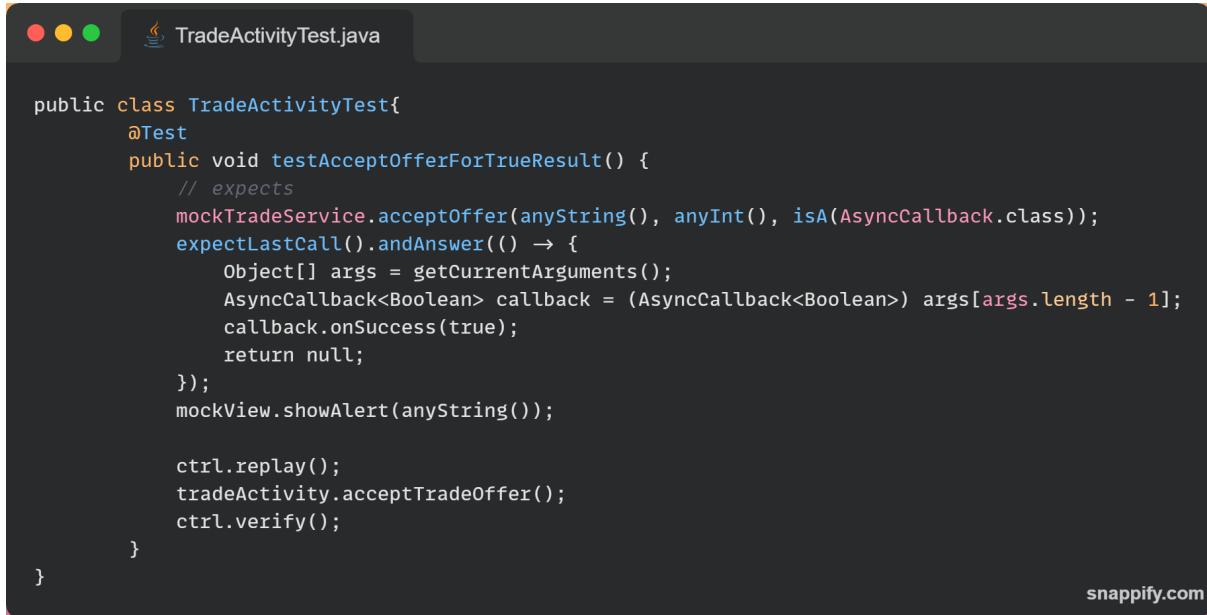
TEST

- Client
- Server
- Shared

CLIENT

Per quanto riguarda la parte di test abbiamo suddiviso i package esattamente come per l'applicazione, puramente per una questione organizzativa.

Nel package client abbiamo testato soprattutto le classi activity, segue un esempio di test sulla classe TradeActivity, nello specifico è il test sul metodo `acceptTradeOffer()`:



The screenshot shows a Java code editor with a dark theme. The title bar says "TradeActivityTest.java". The code is a JUnit test for a class named TradeActivity. It uses EasyMock to mock a TradeService and a View. The test checks if acceptOffer is called with specific arguments and if showAlertDialog is called with any string. The code includes annotations like @Test and expects, and methods like ctrl.replay() and ctrl.verify().

```
public class TradeActivityTest{
    @Test
    public void testAcceptOfferForTrueResult() {
        // expects
        mockTradeService.acceptOffer(anyString(), anyInt(), isA(AsyncCallback.class));
        expectLastCall().andAnswer(() -> {
            Object[] args = getCurrentArguments();
            AsyncCallback<Boolean> callback = (AsyncCallback<Boolean>) args[args.length - 1];
            callback.onSuccess(true);
            return null;
        });
        mockView.showAlertDialog(anyString());

        ctrl.replay();
        tradeActivity.acceptTradeOffer();
        ctrl.verify();
    }
}
```

snappyf.com

`mockTradeService.acceptOffer(anyString(), anyInt(), isA(AsyncCallback.class))`: Questa riga specifica l'aspettativa che il metodo acceptOffer del mockTradeService venga chiamato con argomenti specifici: qualsiasi stringa (anyString()), qualsiasi intero (anyInt()), e un oggetto che implementa l'interfaccia AsyncCallback (isA(AsyncCallback.class)).

`expectLastCall().andAnswer(() -> {...})`: Questo blocco di codice specifica cosa fare quando il metodo acceptOffer viene chiamato. In questo caso, viene utilizzato andAnswer per fornire un'implementazione personalizzata. Quando il metodo viene chiamato, vengono ottenuti gli argomenti della chiamata e viene estratta la callback. Successivamente, viene chiamato il metodo onSuccess(true) sulla callback, simulando il completamento con successo dell'operazione.

`mockView.showAlertDialog(anyString())`: Si aspetta che il metodo showAlert del mock della vista (mockView) venga chiamato con qualsiasi stringa come argomento.

`ctrl.replay()`: Questo indica a EasyMock di iniziare la fase di "replay", durante la quale il framework registra tutte le chiamate ai mock che verranno successivamente verificate.

`tradeActivity.acceptTradeOffer()`: Questa è l'invocazione effettiva del metodo che si sta testando, ovvero acceptTradeOffer sulla tradeActivity.

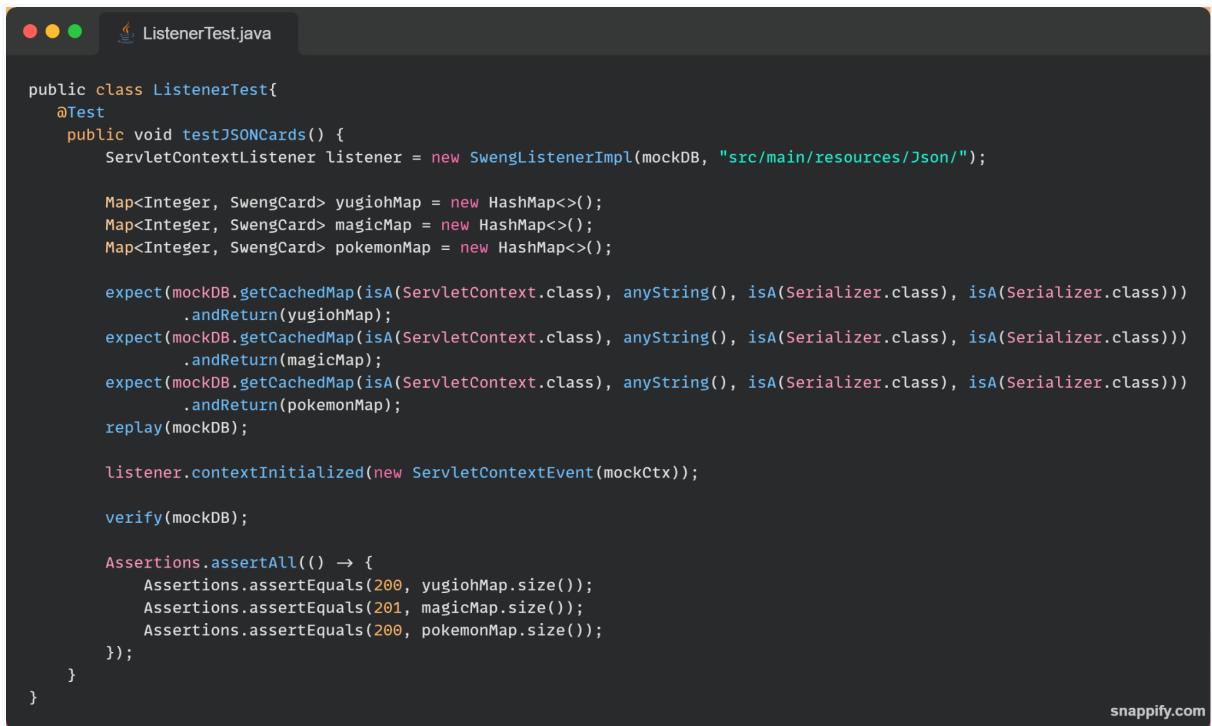
`ctrl.verify()`: Dopo l'esecuzione del metodo, questa chiamata verifica che tutte le aspettative definite precedentemente siano state soddisfatte. Se una di esse non è stata soddisfatta, il test fallirà.

In sintesi, questo test simula il comportamento di chiamata al servizio acceptOffer con successo, dove il risultato è true. Dopo questa chiamata, si aspetta che il metodo showAlert della vista venga chiamato e, infine, il test verifica che tutte le aspettative siano state soddisfatte.

SERVER

per la parte del server dei test abbiamo cercato di testare il più possibile tutto il codice

Il test “testJSONCards” è progettato per verificare l'adeguato funzionamento del listener di contesto servlet, specialmente durante la fase iniziale di inizializzazione. L'obiettivo principale è assicurarsi che le interazioni con il mock del database avvengano come previsto e che le mappe interne siano inizializzate con le dimensioni corrette. Il superamento di questo test implica che il listener è configurato correttamente e in grado di gestire in modo efficace la preparazione iniziale delle risorse di gioco in particolare, il test si concentra sulla corretta configurazione e inizializzazione delle mappe interne rappresentanti diverse categorie di carte di giochi (Yu-Gi-Oh, Magic, Pokémon).



```
● ● ● ListenerTest.java

public class ListenerTest{
    @Test
    public void testJSONCards() {
        ServletContextListener listener = new SwengListenerImpl(mockDB, "src/main/resources/Json/");

        Map<Integer, SwengCard> yugiohMap = new HashMap<>();
        Map<Integer, SwengCard> magicMap = new HashMap<>();
        Map<Integer, SwengCard> pokemonMap = new HashMap<>();

        expect(mockDB.getCachedMap(isA(ServletContext.class), anyString(), isA(Serializer.class), isA(Serializer.class)))
            .andReturn(yugiohMap);
        expect(mockDB.getCachedMap(isA(ServletContext.class), anyString(), isA(Serializer.class), isA(Serializer.class)))
            .andReturn(magicMap);
        expect(mockDB.getCachedMap(isA(ServletContext.class), anyString(), isA(Serializer.class), isA(Serializer.class)))
            .andReturn(pokemonMap);
        replay(mockDB);

        listener.contextInitialized(new ServletContextEvent(mockCtx));

        verify(mockDB);

        Assertions.assertAll(() -> {
            Assertions.assertEquals(200, yugiohMap.size());
            Assertions.assertEquals(201, magicMap.size());
            Assertions.assertEquals(200, pokemonMap.size());
        });
    }
}
```

snappify.com

Il test inizialmente crea un'istanza del listener SwengListenerImpl passando un mock dell'oggetto MapDB (mockDB) e specificando un percorso alla directory delle risorse JSON ("src/main/resources/Json/") dove si trovano i file json contenenti le descrizioni delle carte, ci sono tre file Json, uno per gioco. Tre mappe vuote vengono inizializzate, ciascuna rappresentante un tipo di gioco: yugiohMap, magicMap, e pokemonMap.

Il test configura le aspettative del mock del database (mockDB) riguardo alle chiamate al metodo `getCachedMap()`. Si prevede che questo metodo venga chiamato tre volte con parametri specifici, e ogni chiamata dovrebbe restituire una delle mappe create in precedenza.

La fase di "replay" del mock viene attivata, indicando al framework di mocking di registrare le chiamate ai mock. Successivamente, viene invocato il metodo contextInitialized del listener passando un evento di contesto servlet (ServletContextEvent). Questo dovrebbe attivare l'inizializzazione del contesto servlet secondo le aspettative configurate sul mock.

Dopo l'esecuzione del metodo contextInitialized, viene verificato che tutte le aspettative definite sul mock (mockDB) siano state soddisfatte mediante la chiamata a verify(mockDB).

Infine, vengono eseguite assertions per verificare che le dimensioni delle mappe interne (yugiohMap, magicMap, pokemonMap) siano giuste rispetto a ciò che è contenuto all'interno dei file.

SHARED

Nella parte shared abbiamo testato il models e i throwables.



The screenshot shows a Java code editor window with the title bar "OwnedCardTest.java". The code is a JUnit test for a GsonSerializer. It includes imports for `java.io.DataInput2`, `java.io.DataOutput2`, and `com.google.gson.Gson`. The test method `testEqualsSerializer` creates a `Gson` object, a `GsonSerializer<OwnedCard>`, and a `DataOutput2` object. It then serializes an `OwnedCard` object into the output stream. After that, it creates a `GsonSerializer<OwnedCard>` and a `DataInput2` from the serialized bytes. Finally, it uses `Assertions.assertEquals` to compare the original `OwnedCard` and the deserialized one. The code is annotated with `@Test`.

```
public class OwnedCardTest{
    @Test
    public void testEqualsSerializer() throws IOException {
        Gson gson = new Gson();
        GsonSerializer<OwnedCard> serializer = new GsonSerializer<>(gson);

        DataOutput2 out = new DataOutput2();
        serializer.serialize(out, oCard);

        byte[] data = out.copyBytes();
        GsonSerializer<OwnedCard> deserializer = new GsonSerializer<>(gson);
        OwnedCard deserializedOCard = deserializer.deserialize(new DataInput2.ByteArray(data), 0);

        Assertions.assertEquals(oCard, deserializedOCard);
    }
}
```

In questo esempio se il test ha successo, significa che il processo di serializzazione e deserializzazione attraverso Gson preserva la coerenza delle informazioni dell'oggetto OwnedCard, e che il metodo equals della classe è implementato correttamente per confrontare le istanze prima e dopo la serializzazione.

nello specifico: viene istanziato un oggetto GsonSerializer utilizzando l'istanza di Gson e la classe OwnedCard come tipo generico.

Viene creato un nuovo oggetto DataOutput2 che simula l'output di dati, e il serializer Gson (serializer) viene utilizzato per serializzare l'istanza oCard.

L'output della serializzazione è un array di byte (data).

Viene creato un nuovo GsonSerializer per deserializzare l'oggetto OwnedCard.

Viene utilizzato un oggetto DataInput2.ByteArray per fornire l'array di byte serializzato (data) come input per la deserializzazione, quindi si crea un nuovo oggetto OwnedCard deserializzato (deserializedOCard).

L'asserzione finale verifica che l'istanza originale oCard sia uguale all'istanza deserializzata deserializedPCard questo verifica che il processo di serializzazione e deserializzazione non ha alterato lo stato dell'oggetto e che il metodo equals della classe OwnedCard funzioni correttamente.