Developer Guide

Group: Alessio Arcara, Alessia Crimaldi, Davide Fermi, Matteo Sacco
Project SWENG - a.a. 2022/2023

Quick Start Guide

To successfully launch, test, and use the application, it is important to follow some simple steps:

- Have the Java 11 or higher SDK installed
- Clone repository from GitHub at URL: https://github.com/alessioarcara/project_sweng.git
- Open a terminal in the root folder of the downloaded source code and then run the following commands:
  `mvn clean` to clean up any previous program compilations
  `mvn package` to create the program build and run it
  `mvn gwt:devmode` to run the program in development mode

At this point, "Jetty" will open, a control window of the GWT execution environment, which will allow you to start the application through the "Launch Default Browser" button. Once this is done, you can use any browser to open the application page at the address "127.0.0.1:8888" (this operation is usually automatically performed by Jetty when you click the "Launch Default Browser" button).

Test Driven Development TDD

The importance given to Test Driven Development (TDD) in this project is of a higher degree than any other practice adopted. In fact, application performance has often been sacrificed in order to make the most of the business logic for functional requirements testable.
With its over 380 unit tests, the code of this application has been tested everywhere possible. The entire server side of the application has a 100% Code Coverage, while the client side, of course, sees this number decrease, although it remains very high for being client side tests, in fact, it is possible to count on a coverage of almost 50%. Even the model side, like the server side, has been thoroughly tested, reporting a 100% coverage.

Application section	Classes coverage	Methods coverage	Lines coverage
Overall	66 %	54 %	55 %
Client	46 %	29 %	28 %
Server	100 %	95 %	92 %
Shared	100 %	82 %	87 %

It has been possible to test the client-side part of the project thanks to the separation of concern obtained through the use of the MVP model: it's in fact been possible to test every Presenter of the application and consequently the entire business logic of the client.

The structure chosen to build the tests is that of Unit Tests, allowing tests in a sealed compartment on the individual methods that were chosen to be tested: in this case, every testable method within the application was tested, excluding the private methods that would have required an additional library called PowerMock. The Unit Test type of test was chosen so that it would be possible to have the best coverage by isolating any problem and having the ability to test by eliminating functional dependencies within the tests themselves.

To make this possible, several choices have been made within the code, such as allowing each service to have a constructor specifically for tests. In fact, one of the two constructors provided to the services specifically allows passing the instance of the Database to be used within the class. This allowed us to create two objects that simulated the real database exclusively for testing purposes: the first, used in all tests that do not include writeOperation, is a mockDB, while the second used specifically for tests that include writeOperation is the fakeDB class.

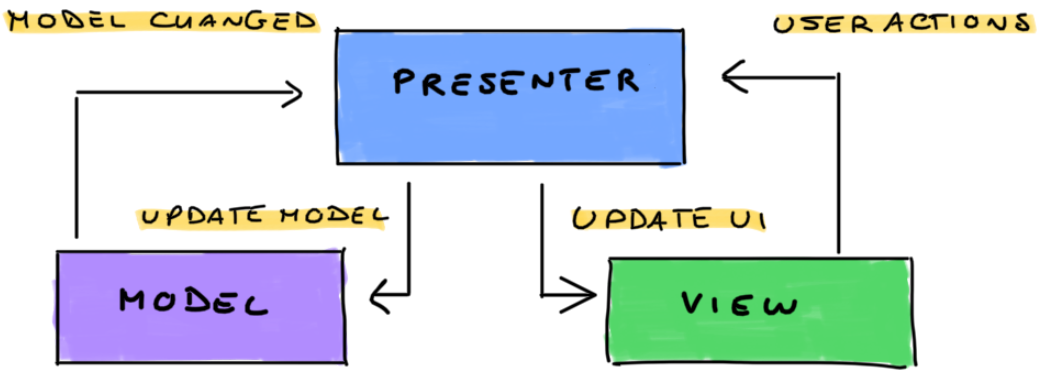
The difference between the two types of databases is essentially in the type of response provided. While the first simulates being a database, and as a result simulates the responses to be provided to the client, the second is actually a real database, but fully cached, with predefined data and of significantly smaller dimensions than the real one. However, being a real database nonetheless, fakeDB does not have to simulate responses to the client, but can instead provide the real expected responses also from the database used in the actual application.

MVP

To facilitate the testing process of the entire application, i.e. to be able to test the client logic as well, it was chosen to structure the entire application according to a precise model, namely Model View Presenter (MVP).
This allows the client to perform no logic, but only to display the data loaded from the server through the presenters, called Activity in this application.
All activities have been tested in all their public methods thanks to the use of the EasyMock library, specially designed to perform tests with the JUnit framework.

MVP is a design pattern that separates the logic of a user interface from the business logic. It consists of three main components:

- Model: represents the business objects.
- View: responsible for displaying data and handling user interactions. Views have no notion of the model.
- Presenter: acts as a bridge between the Model and the View. It handles all of the logic for our application.



Client

Activities and Places

In our application has been used a built-in framework for browser history management: the Activity and Places framework.
The presenters in our web application will be represented by `activities` and they will have the purpose of containing both the business logic and a set of life cycles. A `place` is a POJO (Plain Old Java Object) representing a particular state or location within the application. A Place can be converted to and from a URL history token.

The `PlaceController` is a crucial component for Activities and Places framework. It is responsible for coordinating the navigation between different places in the application. It does this by updating the browser's URL and the browser's history stack, so that the user can use the browser's back and forward buttons to navigate through the application.

The PlaceController also acts as a bridge between the places and the activities. When a user navigates to a new place, the PlaceController will call the appropriate activity to handle the transition to the new place.

ClientFactory, AppPlaceHistoryMapper and AppActivityMapper

The ClientFactory class is used within the application as a container of all dependencies necessary for the class AppActivityMapper, meaning it contains all the views, the PlaceController and the AuthSubject.
Instantiated at application startup, it creates instances of all these objects available in the application navigation, and then allows the AppPlaceHistoryMapper to control navigation just like a Routes Manager: by passing the correct references of desired Views to their respective Activities based on the Place defined in the URI.

The AppPlaceHistoryMapper allows the correct Places to be called based on the URI used to navigate the application.
From the present address a token gets parsed and extracted, then it is compared with the possible

constants defined in the RouteConstants class: depending on the token found, it either returns the desired view by creating the relevant Place or redirects to the default view (Home).

After creating the Place, it is passed to the AppActivityMapper: this class manages the received Place and based on the comparison result, returns the corresponding activity.

This is where the functionality of the ClientFactory comes into play, as it is used to pass the right components to the selected Activity, such as PlaceController, AuthSubject and Views. By using the method described earlier, the dependencies are always injected into the Activity, which will not need to recreate the necessary objects: this allows for the use of AuthSubject, PlaceController, and Views as Singletons.

The use of Singletons has been chosen because performing repeated instantiation operations in GWT during the application's use by the user would be very costly, as operations on the DOM are quite expensive considering the GWT framework directly interfaces with the DOM itself.

```
public interface ClientFactory {
    EventBus getEventBus();
    PlaceController getPlaceController();
    HomeView getHomeView();
    CardView getCardView();
    AuthView getAuthView();
    DecksView getDecksView();
    AuthSubject getAuthSubject();
    NewExchangeView getNewExchangeView();
    ExchangesView getExchangesView();
}

public class AppActivityMapper implements ActivityMapper {
    public Activity getActivity(Place place) {
        if (place instanceof HomePlace)
            return new HomeActivity(...);
        if (place instanceof CardPlace)
            return new CardActivity(...);
        if (place instanceof AuthPlace)
            return new AuthActivity(...);
        if (place instanceof DecksPlace)
            return new DecksActivity(...);
        if (place instanceof NewExchangePlace)
            return new NewExchangeActivity(...);
        if (place instanceof ExchangesPlace &&
            ((ExchangesPlace) place).getProposalId() == null)
            return new ExchangesActivity(...);
        if (place instanceof ExchangesPlace)
            return new ExchangeActivity(...);
        return null;
    }
}
```

Share the AuthSubject through the application

For the AuthSubject and authentication the Design Pattern Observer has been used, this allowed the application to react to events triggered internally by the user based on whether the user is authenticated or not. In fact, all classes that need to interface both client side and server side with the user's access token are AuthSubject Observers.

An example of the active reaction of the application seen above can be found in the management of Routing, since every time the URI of the page is updated or undergoes a change, not only the token that allows the Place to be managed is checked, but also, if necessary the AuthSubject to decide whether the user is allowed to visit the requested page or not.

If this level is exceeded by the user who has correctly logged into the system, the same AuthSubject will be shared with the Activities, which in turn will be able to manage certain behaviours related to the offered functionalities, such as displaying the right information and operating the right functionalities. Finally, through the same AuthSubject the token will be passed to the server during the execution of the functions requested by the client.

Binding the UI

One of the decisions made in client-side development relates to UIBinder, which represents a binding layer between interfaces defined statically, as templates, and the logic that governs them. In this application, there are XML files that are exclusively responsible for defining the layout and style of the interface (the Views), and their corresponding Presenters (the Activities) that contain all the logic that communicates the client with the server, enabling users to take advantage of the operations offered by the system.

Within the views, we used additional templates that we defined for the objects that were intended to display in various parts of the application (such as Decks, PhysicalCards and Cards). These, called Widgets, were defined by a formatting file with the extension '.xml' and a corresponding file that defines the possible interaction with the logic required by the client to perform the application's functions through the Activities.

Indeed, there is a tree hierarchy among these three mentioned components, in which functional logic is defined in the Activity and passed to the View and if necessary to the Widget used to display the data inside the View. To achieve this, interfaces implemented by the Views were used based on the functionalities that needed to be invoked within them. These interfaces are called 'ImperativeHandlers'.

ImperativeHandlers

Within the client, as explained above, event management was handled entirely through the creation of specially constructed interfaces to receive the possible events arising within the application from the user's interaction with it.

These are precisely the ImperativeHandlers, and here follows a mention of those created and used:

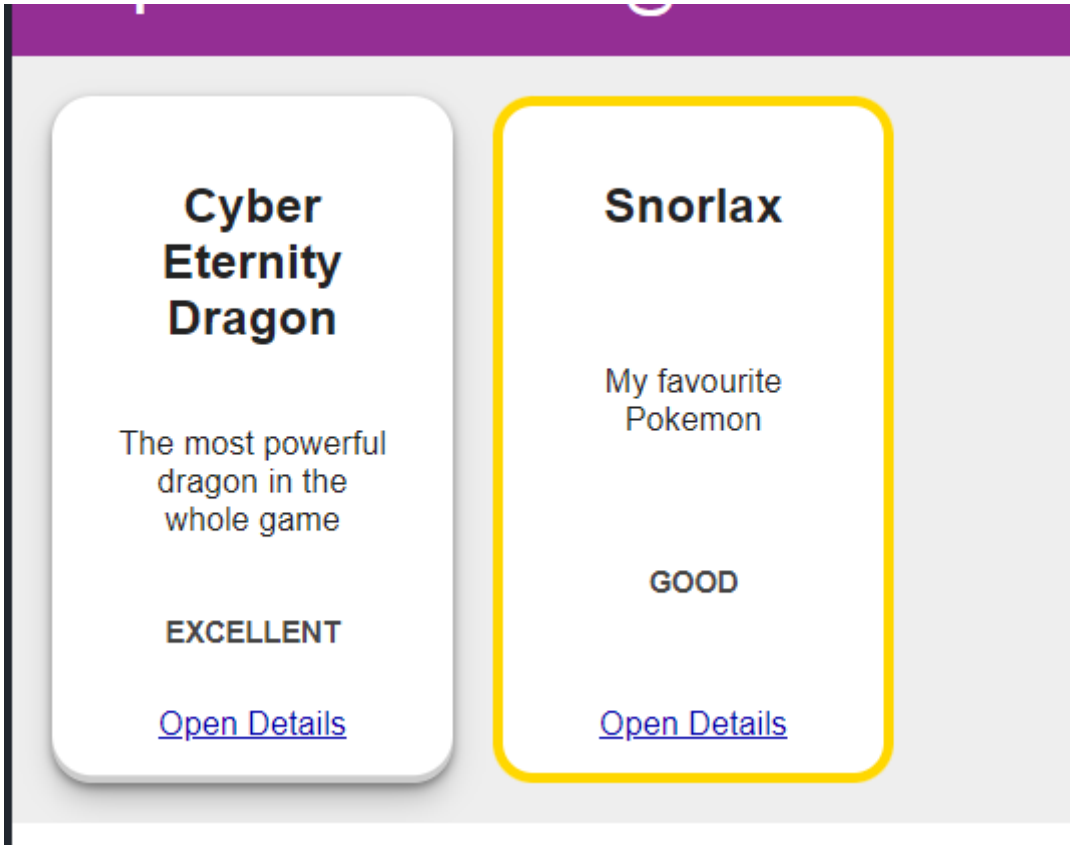
- ImperativeHandleAddCardToDeck
- ImperativeHandleAddCardToDeckModal
- ImperativeHandleCard
- ImperativeHandleCustomDeck
- ImperativeHandleDeck
- ImperativeHandlePhysicalCardEdit
- ImperativeHandlePhysicalCardRemove
- ImperativeHandlePhysicalCardSelection
- ImperativeHandleProposalList
- ImperativeHandleSidebar
- ImperativeHandleUserList

Example of one important widgets

PhysicalCardWidget

The CardWidget, as just seen, allows for the display of a physical card's content, but besides this function it plays an important role in the Exchange pages and in the display of decks in their page, specifically in the display of the cards in the Owned deck.

By clicking on a card with the mouse, the user will be able to change its status between selected and unselected. This allows the application to provide feedback by changing the outer border of the card from null to green to signal to the user which ones are selected, and keep track of these cards during the proposal sending phase: only the selected cards will in fact be sent to the server for the creation of the actual exchange proposal.



In addition, the same "selected" property can be used in the deck display page. In this way, the user will be able to click on the cards in the "Owned" deck to select them and add them to the desired Custom deck.

The card widget contains other features inside it that can only be seen and used while inside the Decks Page: specifically, the possibility of removing the card from the deck ("X" in the top right), which removes the card from the selected deck, and if this is the "Owned" deck, the card is also removed from all custom decks that contain it.

Finally, a property that can only be used in the Default decks (Owned and Wished) is the ability to edit the card's content: Status and Description. This property is identified by a blue pencil in the top right next to the button to delete the card.

Server

MapDB

MapDB was used for the database, and it was chosen to make it fully transactional in write operations, as suggested by its own documentation. This implementation choice offers the possibility of having atomicity in write operations and avoiding concurrent problems on the database itself. An even more important feature is the ability to perform a rollback if a problem is detected or if the changes made should not be persisted. Finally, the most important feature offered by making the database transactional is the ability to ensure the integrity of the database itself in case of a system crash, even with the database opened. For this reason, the `db.commit();` operation is never performed at the end of the write operations within the code that calls them.

The concrete database class, **MapDBImpl**, is used to instantiate and manage the database itself and, within it, it overrides the `writeOperation` method twice. This choice allows write operations on the database to be called both from any server-side method and from inside a method that has already requested the context to access the database. The second Override actually allows passing a Runnable Object to the function: this might be a callback function that calls a method which in turn already requests access to the database to perform write operations.

```
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_TYPE);
    try {
        T value = operation.apply(db.hashMap(mapName,
                                             keySerializer,
                                             valueSerializer
```

```
                                                ).createOrOpen());
            db.commit();
            return value;
        } catch (Exception e) {
            db.rollback();
            return null;
        }
    }

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

MapDB usages

In order to thoroughly test the Services and server RPCs, each Service has two constructors that allow it to be instantiated by creating its own instance of MapDBImpl, or by passing a db into the constructor to allow it to save the db reference. This way of instantiating and calling Services is used in the test classes by passing two types of databases, a mock and a fake one, in order to control the responses received and the behaviors of the service without touching the real database. Moreover this method allows to do Dependency Injection of the DB reference without recreating it every time.

Parsing JSON files to import them

The class JSONParser is responsible for importing JSON files from resources and converting them into an array of Card objects specific to each game. The constructor of the class JSONParser accepts an instance of a class implementing the CardParseStrategy interface, which allows it to use a custom `execute()` method to obtain a Card object from the JSON object passed to the parser. It also requires an instance of Gson to parse the JSON file.

The `parseJSON()` method accepts a file path as an argument and uses GSON to parse the JSON file, returning an array of JSON objects. This array is then passed, one element at a time, to the CardParseStrategy's `execute()` method, which returns an array of Card objects.

Along with JSONParser, the CardParseStrategy is defined with the implementation of a Strategy Pattern split in four classes (one for each game and an interface):

- YuGiOhCardParseStrategy -> for YuGi-Oh cards
- MagicCardParseStrategy -> for Magic cards

- PokemonCardParseStrategy -> for Pokemon cards
- CardParseStrategy -> Interface for the Strategies

Here's a sample code for one of the strategies:

```
public class MagicCardParseStrategy implements CardParseStrategy {
    public MagicCard 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 MagicCard(name, description, types, artist, rarity,
                             variants.toArray(new String[0]));
    }
}
```

How imported JSON are loaded

When the application is loaded, thanks to the Parse Strategy just seen, the card data for each game are loaded from JSON resources. These are saved in memory immediately after loading, but the choice made in this case was to use a MemoryDB, i.e. to save the data on a cache memory. This allows, during the operation of the application and use by the user, very fast and efficient data retrieval, without the need to contact the physical database by performing I/O operations at each read operation.

The information needed to persist the data for the catalogue of available playing cards is given by the same JSON files present in the project resources (subdivided by game).

GsonSerializer

```
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);
    }
}
```

The GsonSerializer class allows objects of a generic type T to be serialized and deserialized using the GSON library.

The `serialize` method takes an object, than converts the value to a JsonObject using the `toJsonTree()` method from GSON. It then adds a property to the JsonObject called "classType"

which stores the class name of the object. Finally, it writes the JsonObject as a UTF string to the DataOutput2 object.

The `deserialize` reads the UTF string from the DataInput2 object and converts it back into a JsonObject using the `fromJson()` method offered by GSON. It then retrieves the "classType" property and uses it to find the class object of the retrieved value. Finally, it uses GSON to deserialize the JsonObject to an object of the appropriate class using the class object as the type parameter.

The double constructor of the GsonSerializer class allows the serializer to perform serialization and deserialization not only for Java objects, but also for collections. For example, decks are saved in the database as `Map<String, Map<String, Deck>>` and the nested second map would be lost for type erasure if it was not possible to define it as a type in the serializer. The double constructor specifically allows to assign `Map<String, Deck>` as the type to be used in serialization and therefore retrieve data from the database.

Saving Card objects in MapDB

The class ListenerImpl, which is a ServletContextListener, is used to listen when the web application is started. The `contextInitialized` method is called when the web application is started and it overrides the one in ServletContextListener.

It first initialises the memory MapDB database and creates some maps to store Card objects. It also creates instances of the GSON library, JSONParser, and the different concrete strategies to parse JSON files and store the data in the maps.

```
public class ListenerImpl implements ServletContextListener, MapDBConstants {
    private void uploadDataToDB(Map<Integer, Card> map, Card[] cards) {
        for (Card 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<Card> cardSerializer = new GsonSerializer<>(gson);

        Map<Integer, Card> yuGiOhMap = db.getCachedMap(sce.getServletContext(),
                                YUGIOH_MAP_NAME, Serializer.INTEGER,
cardSerializer);
        Map<Integer, Card> magicMap = db.getCachedMap(sce.getServletContext(),
                                MAGIC_MAP_NAME, Serializer.INTEGER,
cardSerializer);
```

```
            Map<Integer, Card> pokemonMap =
db.getCachedMap(sce.getServletContext(),
                                POKEMON_MAP_NAME, Serializer.INTEGER,
cardSerializer);

        JSONParser parser = new JSONParser(new YuGiOhCardParseStrategy(), gson);
        try {
            uploadDataToDB(yuGiOhMap, parser.parseJSON(path +
"yugioh_cards.json"));
            parser.setParseStrategy(new MagicCardParseStrategy());
            uploadDataToDB(magicMap, parser.parseJSON(path +
"magic_cards.json"));
            parser.setParseStrategy(new PokemonCardParseStrategy());
            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. ***");
    }
}
```

Authentication and authorization

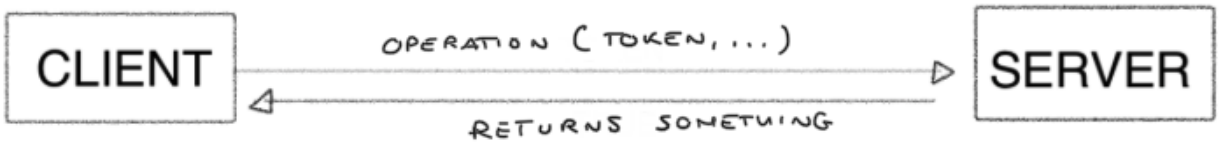
Login and registration flows

1. When a user attempts to login or register, the server first checks if the provided credentials are valid. If the user is registering, a new account is created with an encrypted password to protect sensitive user information.
2. A token is then generated, which is the SHA-256 cryptographic hash of the user's ID and a secret value. This token serves as a unique identifier for the user's session and will expire after 7 days.
3. That token is used to create a LoginInfo map, which contains the user's ID and the time the token was created in UNIX time. the LoginInfo map is used to keep track of the user's login status and to ensure that the token it is valid. Without knowing the secret value, it would not be possibile to generate a valid key for LoginInfo map and access a valid entry.
4. The token is then returned to the client through an RPC call, and is stored in two places:
    - A cookie, which allows the user to stay logged in even if they reload the page
    - A subject, which is part of the **observer pattern**, is passed to presenters and *placeHistoryMapper* through dependency injection. They can attach to the subject, allowing

them to be notified of changes in the user's login status.

5. The user is now considered logged in and can access the protected parts of the application. When a call that requires authorisation is made to the server, the token is sent along with it and the server can use it to verify the user's identity and ensure that they have the necessary permissions to access the requested resources.

Session flow and Token Validity



After a user has authenticated and uses the user reserved pages and services of the website, he sends in each request the session token to the server in order to certify his session. Server side a method verifies that the token is still valid: its creation date must be less than seven days.

Logout flow

1. When a user log out, the client sends the token to the server. The server then uses the token to locate the corresponding entry in the LoginInfo map and removes it, invalidating the token and preventing others from using it to access protected resources.
2. The client also removes the token from the subject, notifying other parts of the application of the user's logout status.
3. Finally, the client removes the token from the cookie.

In this way, the logout ensures that both the client and the server reflect the change in the user's login status and the session is securely terminated.

Shared

Exceptions

In order to communicate clearly with the user of the application, it was decided to utilize custom checked exceptions thrown by the server in case of malfunction due to incorrect user inputs, unauthorized operations or missing data searches by the client. In fact, five classes have been created that represent the exceptions that the user could trigger, namely:

- AuthException

- InputException
- DeckNotFoundException
- ProposalNotFoundException
- ExistingProposalException

All the classes listed above extend a type of exception created specifically to represent the simplified exceptions searched for. In fact, this class extends Exception itself and implements only one method, to return its only parameter `errorMessage`, from which the `getErrorMessage()` method.

This greatly improves the user experience, as it will simply be necessary to throw the desired exception to immediately communicate to the client what went wrong during the operation.

```
public class BaseException extends Exception implements Serializable {
    //...
    private String errorMessage;

    public BaseException(String errorMessage) {
        this.errorMessage = errorMessage;
    }

        //...

    public String getErrorMessage() {
        return errorMessage;
    }
}
```

Models

Card

Abstract class:

- Card

Sub classes:

- YuGiOhCard
- MagicCard
- PokemonCard

The Card object is the most widely used within the application, being a portal for card exchange. To create a model that could be adequate, it was thought of a series of common fields for all cards in each game and a series of "particular" properties of each card, assigned based on the game of belonging. In order to represent all the cards, a model was thought of with the following fields:

- name

- description
- type

The Boolean attributes of each card different for each game were thought of as a set of strings and called `variants`. If the string representing the variant is present in the set for that card, it means that the card itself possesses that attribute. This made it possible to use a much more flexible scheme and to comply with *Open Close Principle*.

```
public Card(String name, String description, String type, String... variants) {
    this.id = uniqueId.incrementAndGet();
    this.name = name;
    this.description = description;
    this.type = type;
    this.variants = new HashSet<>();
    Collections.addAll(this.variants, variants);
}
```

An example of a card for the game **Magic**, in this specific case the variants set corresponds to multiple fields of which `{"hasFoil", "isAlternative", "isFullArt", "isPromo", "isReprint"}`:

```
public MagicCard(String name, String description, String type,
                 String artist, String rarity, String... variants) {
    super(name, description, type, variants);
    this.artist = artist;
    this.rarity = rarity;
}
```

Physical card model

After being able to handle the cards objects in the application, to respect the functional requirements and giving the possibility to the users of importing and managing their physical cards, two models have been created.

The specific model used for the Physical Card is very simple but efficient at the same time, it uses a field that represents the Card Catalog ID that is passed in the constructor as a reference to the specific object. Then it creates an personal ID with a character representing the game type and an incremental id, a status [VeryDamaged(1), Damaged(2), Fair(3), Good(4), Excellent(5)] and a description that are different from any other physical object of the same type or of the same "Card".

```
public PhysicalCard(Game game, int cardId, Status status, String description) {
    this.id = (game == Game.MAGIC ? "m" :
               game == Game.POKEMON ? "p" :
               game == Game.YUGIOH ? "y" : ""
              ) + uniqueId.getAndIncrement();
    this.cardId = cardId;
```

```
    this.status = status;
    this.description = description;
}
```

The Physical card also allows creating a copy of itself by using the specific constructor: this is useful for the operations server side to allow the system to manage cards between decks without recreating their UniqueId, otherwise the created card would be different from the previous.

```
// create a copy
protected PhysicalCard(String id, int cardId, Status status, String description)
{
    this.id = id;
    this.cardId = cardId;
    this.status = status;
    this.description = description;
}
```

Through this method, changing the card is done immutably, a copy of the physical card is created by editing the description and status fields on the copy.

```
public PhysicalCard copyWithModifiedStatusAndDescription(Status newStatus,
                                                         String newDescription) {
    return new PhysicalCard(getId(), getCardId(), newStatus, newDescription);
}
```

Deck model

This model represents the decks of physical cards that a user can have, to integrate the two default decks ("Owned" and "Wished") that a user must have and can not delete, has been introduced a variable called `isDefault`: this variable is a boolean field that indicate if the deck is one of the two default decks that can't be deleted or renamed.

The structure of the deck is very simple, other than isDefault field, it has a name and a Set collection that contains the physical cards owned by the user.
The constructor can be different because the default one sets the "isDefault" field to false. The necessity of having two constructors is given by the fact that the one not default is used a lot more often than the other.

```
public void setName(String name) {
    if (!isDefault) {
        this.name = name;
    }
}

public Deck(String userEmail, String name) {
    this.userEmail = userEmail;
    this.name = name;
```

```
        this.isDefault = false;
        physicalCards = new LinkedHashSet<>();
    }

    public Deck(String userEmail, String name, boolean isDefault) {
        this.userEmail = userEmail;
        this.name = name;
        this.isDefault = isDefault;
        physicalCards = new LinkedHashSet<>();
    }
```

The class implements the standard getter methods, a single setter for the field name and also the methods to access the information of the cards list:

```
    public boolean addPhysicalCard(Integer physicalCardId) {
        return physicalCards.add(physicalCardId);
    }

    public boolean removePhysicalCard(Integer physicalCardId) {
        return physicalCards.remove(physicalCardId);
    }

    public boolean containsPhysicalCard(Integer physicalCardId) {
        return physicalCards.contains(physicalCardId);
    }
```

Thank to the LinkedHashSet object the methods to access, push and remove data from the deck can be simplified just by returning the corresponding method offered by the LinkedHashSet.

Proposal model

This model represents the proposal object: in which are stored the data needed to exchange cards between users. This model expects four parameters in the constructor: a senderUserEmail, a receiverUserEmail and two `List<PhysicalCardImpl>` objects in which are stored the Physycal Cards the users want to exchange between each other. It only implements the standard constructor and getter methods.

```
    public Proposal(String senderUserEmail, String receiverUserEmail,
                    List<PhysicalCard> senderPhysicalCards,
                    List<PhysicalCard> receiverPhysicalCards) {
        this.id = uniqueId.getAndIncrement();
        this.senderUserEmail = senderUserEmail;
        this.receiverUserEmail = receiverUserEmail;
        this.senderPhysicalCards = senderPhysicalCards;
        this.receiverPhysicalCards = receiverPhysicalCards;
        this.date = System.currentTimeMillis();
    }
```

Payloads

Sometimes, it's necessary to perform join operations on raw data from different maps in order to provide the client with slightly different information than what is stored in the database. However, making multiple fetches to the database from the client can be inconvenient. To avoid this, specially-designed objects have been created to return information in a more convenient way.

These objects are called Payloads and they are an elaboration of the models seen previously, with additional information added to them.

For instance, the ProposalPayload consists of a Proposal, but instead of the List<*PhysicalCard*> present in the Proposal model, it returns List<*PhysicalCardWithName*> objects. This approach minimizes the number of times the information needs to be saved in the database, while providing the client with all the required information.

Every PhysicalCard in fact contains within it the id of the Card it refers to. It wouldn't be optimal to save all the information of the Card (i.e. the Name) in the PhysicalCard model as well.

There are three Payloads, one for authentication which returns the user's email and token to the AuthSubject, one for modified decks (used after removing and modifying cards from decks) which returns a deck with PhysicalCardWithName instead of PhysicalCard, and finally the proposal payload mentioned before.

CardExchange API

We have chosen to document the RPCs because it allows other developers to understand the available methods that can be called remotely and facilitate collaboration among team developers, including those who may be less familiar with the underlying architecture.

Authorization:

signup

```
public AuthPayload signup(String email, String password) throws AuthException
```

- Purpose: User registration
- Input: *email, password* (provided by user in registration form)
- Output: *AuthPayload*

After the usual checks on the validity of the input parameters, the method checks whether the user is already present in the DB, if not, it saves the data and calls an external method to create the default decks. It returns *AuthPayload*, after creating and saving the login *token* via the *generateAndStoreLoginToken* method.

signin

```
public AuthPayload signin(String email, String password) throws AuthException
```

- Purpose: User login
- Input: *email, password* (provided by user in login form)
- Output: *AuthPayload*

The method, after verifying that the *user* is present and the *password* matches that decrypted from the database, generates and registers the session *token* returns the *AuthPayLoad*.

logout

```
public Boolean logout(String token) throws AuthException
```

- Purpose: User logout
- Input: *token*
- Output: *Boolean*

The method removes the user's specific *token* from the list of authorised tokens, effectively preventing new session authorisations with that *token*. If it is not found, a specific exception is thrown.

me

```
public String me(String token) throws AuthException
```

- Purpose: Providing the client with information on the ownership of a token
- Input: *token*
- Output: *user email*

It is a simple method that provides information about the owner from a *token*, specifically the *user email*.

Decks operations:

getMyDeck

```
public List<PhysicalCardWithName> getMyDeck(String token, String deckName)
    throws AuthException
```

- Purpose: Providing the client with the list of physical card inside a specific user deck.
- Input: *token, deckname*
- Output: *List of PhisicalCard*

This defines an endpoint getMyDeck that can be called using the RPC method. It takes two required parameters: *token*, which is used to validate the request and extract the email, and *deckName* to filter through the user's existing decks and find the required one. It will return a list of PhysicalCard objects in case of success.

addDeck

```
public boolean addDeck(String token, String deckName) throws AuthException
```

- Purpose: Create in DB a specific user deck.
- Input: *token, deckname*
- Output: *Boolean*

The method first uses the *login map* to extract the *user email* from the *token*, while checking its validity.
Next, the method extracts the *user's deck map* from the DB and checks whether the deck being created already exists. In this case it returns *false*. If, on the other hand, that deck does not yet exist for that specific user, it creates it and returns *true*.

getUserDeckNames

```
public List<String> getUserDeckNames(String token) throws AuthException
```

- Purpose: Providing the client with the list of all decks of specific user.
- Input: *token*
- Output: *List of user decks*

The method first uses the *login map* to extract the *user email* from the *token*, while checking its validity. Next, the method extracts from the *deck map* in DB the list of the names of all decks of that specific user and return it as response.

getUserOwnedDeck

```
public List<PhysicalCardWithName> getUserOwnedDeck(String email) throws
AuthException
```

- Purpose: Providing the client with the list of all physical card in "Owned" user deck.
- Input: *user email*
- Output: *List of physical cards*

By providing a *user email*, the method is used to extract the list of physical cards owned by a given user. This method is also used in the exchange phase, to understand the physical cards owned by the user I am proposing the exchange to, which can then be included in the proposal.

addPhysicalCardToDeck

```
public boolean addPhysicalCardToDeck(String token, Game game, String deckName,
int cardId,
    Status status, String description) throws AuthException, InputException
```

- Purpose: Add a specific physical cart to one specific user deck.
- Input: *token, game, deckname, cardId, status of physicalCard, description of physicalCard*
- Output: *Boolean*

The method first obtains the *user's email* from the *token*, at the same time as validating the *token*. It then checks the consistency of the input parameters. After It then uses the email to identify the *deck map* of the specific user and inserts the card with the indicated id into the deck indicated as a parameter. Along with this operation it also inserts the mandatory parameters for the *physical card*, like state of preservation and description. If the operation of adding cards to the deck is successful, a value of *true* is returned, in the event of failure, the value *false* is returned. In specific error cases (e.g. when the deck does not exist), a specific *exception* is thrown.

removePhysicalCardFromDeck

```
public boolean removePhysicalCardFromDeck(String token, String deckName,
    PhysicalCard pCard) throws AuthException, InputException
```

- Purpose: Remove a specific physical cart from one specific user deck.
- Input: *token, game, deckname, physical cardId*
- Output: *Boolean*

The method first obtains the *user's email* from the *token*, at the same time as validating the *token*. Next, the method takes care of checking the consistency of the input parameters. It then identifies the user's deck from which it removes the *physical card* and proceeds to remove it. If the operation to remove cards from the deck is successful, a *true* value is returned, in the event of failure the value *false* is returned. In specific error cases (e.g. when the deck does not exist), a specific *exception* is thrown.

getOwnedPhysicalCardsByCardId

```
public List<PhysicalCardWithEmail> getOwnedPhysicalCardsByCardId(int cardId)
    throws InputException
```

- Purpose: Given a cardid, retrieve all physical cards owned by all users with that specific cardId.
- Input: *cardId*
- Output: *List of physical cards with email*

As a first operation, the method checks the consistency of the input parameter. It then calls a private method (*getPhysicalCardByCardIdAndDeckName*) which, having supplied a *cardId* and a *deckName*, extracts all the physical cards contained in that specific indicated deck and related to that specific indicated card. It then returns a list of *PhisicalCards*, accompanied by the email of the user who owns them. This is an extract of the private method *getPhysicalCardByCardIdAndDeckName*:

```
private List<PhysicalCardWithEmail> getPhysicalCardByCardIdAndDeckName(int
cardId,
                                                                        String
deckName)
```

removeCustomDeck

```
public boolean removeCustomDeck(String token, String deckName) throws
AuthException
```

- Purpose: Given a deckName, remove specific custom deck.
- Input: *token, deckName*
- Output: *Boolean*

The method first obtains the *user's email* from the *token*, at the same time as validating the *token*. Then, the method takes care of checking the consistency of the input parameter, read from db the *user decks map* and remove the custom deck specified in the request. Finally, the method returns a true value if the request was successful, false if the deck was not found or there is not a *user decks map* required, or an exception if there were problems with removal the deck from the db.

editPhysicalCard

```
public List<ModifiedDeckPayload> editPhysicalCard(String token, String deckName,
    PhysicalCard pCard) throws AuthException, InputException,
ExistingProposalException
```

- Purpose: Given a deckName, edit the properties of the physical card contained within.
- Input: *token, deckName, physicalCard*
- Output: `List<ModifiedDeckPayload>`

The method first obtains the *user's email* from the *token*, at the same time as validating the *token*. It then verifies the input parameters, specifically calls a private method to verify the *deckName*, verifies that the deck in the request is a *default deck* ("Owned" or "Wished"), that the physical card is consistent and finally that the physical card is not present in a current proposal. If all the checks are successful, it takes care of modifying the properties of the physical card in the db, first removing the card from the deck and reinserting the newly modified one, returning edited decks lists throw a list of *ModifiedDeckPayload*.

getListPhysicalCardWithEmailDealing

```
public List<PhysicalCardWithEmailDealing> getListPhysicalCardWithEmailDealing(
    String token, Game game, int cardId)
    throws AuthException, InputException, DeckNotFoundException
```

- Purpose: This method retrieves a list of PhysicalCardWithEmailDealing objects for a specific card in a specific game, based on a user's authentication token.
- Input: *token, game, cardId*
- Output: `List<PhysicalCardWithEmailDealing>`

Initially, the method checks the validity of the token and also extracts the user's e-mail.
It then checks the consistency of the other input parameters and retrieves the deck of cards owned by the user.

The method creates an array of 5 elements, *ownedMatchingPCardId*, to store the ids of physical cards that match the specified parameter cardId and are of different statuses. The method loops through each physical card in the user's deck and if a card with the specified ID is found, its position in the *ownedMatchingPCardId* array is determined based on its status. The array position is then filled with the id of the physical card, if it is not already filled. The loop continues until all positions in the array are filled or all physical cards in the user's deck have been checked. With this array you have direct access to the physical card via its status, knowing the idCard.

```
for (PhysicalCard pCard : userDeck.getPhysicalCards()) {
    if (cardId == pCard.getCardId()) {
        position = pCard.getStatus().getValue() - 1;
        //If position for this Status is empty, fill with this Pcard
        if (ownedMatchingPCardId[position] == null) {
             ownedMatchingPCardId[position] = pCard.getId();
            //if all Status positions are filled, exit from cycle
            if (Arrays.stream(ownedMatchingPCardId).allMatch(Objects::nonNull))
                break;
            }
        }
    }
}
```

Finally, for every wished card in all user's wished deck with the specified parameter cardId, the method check if this user have a specific physical card in his own deck, but only if it meets the status requirements.

getWishedPhysicalCardsByCardId

```
public List<PhysicalCardWithEmail> getWishedPhysicalCardsByCardId(int cardId)
    throws InputException
```

- Purpose: Given a cardid, retrieve all physical cards wished by all users with that specific cardId.
- Input: *cardId*

- Output: *List of physical card with email*

As a first operation, the method checks the consistency of the input parameter. It then calls a private method (*getPhysicalCardByCardIdAndDeckName*) which, having supplied a *cardId* and a *deckName* (in this case "Wished" one), extracts all the physical cards contained in that specific indicated deck and related to that specific indicated card. It then returns a list of *PhisicalCards*, accompanied by the email of the user who owns them.

addPhysicalCardsToCustomDeck

```
public List<PhysicalCardWithName> addPhysicalCardsToCustomDeck(
    String token, String customDeckName, List<PhysicalCard> pCards)
    throws AuthException, InputException, DeckNotFoundException
```

- Purpose: Given a list of physical cards, insert the in a custom deck.
- Input: *toke, cardId, list of physical cards*
- Output: *List of physical card with name*

The method starts by checking the validity of the token provided through the *checkTokenValidity* method, which returns the email of the user associated with the token.
After the usual checks on the parameters supplied and the consistency of the data on the db, it scrolls through the list of physical cards supplied and inserts them into the custom deck indicated, returning a list of physicalCardWithName.

Cards operations:

getGameCards

```
List<Card> getGameCards(Game game);
```

- Purpose: Providing the client with the list of all cards of specific game.
- Input: *game*
- Output: *List of cards*

This defines an endpoint GetGameCards that can be called using the RPC method. It takes a single required parameter *game*, which is a enum and must be one of the values "Magic", "Pokemon" or "Yugioh". It will return a list of *Card objects* in case of success and in case of invalid argument (game == null) will return a message error.

```
public List<Card> getGameCards(Game game)
```

getGameCard

```
public Card getGameCard(Game game, int cardId)
```

- Purpose: Providing the client with the list of all details of specific card, idetified by cardId.
- Input: *game, cardId*
- Output: *Card*

This defines an endpoint *GetGameCard* that can be called using the RPC method. It takes two parameters, a required parameter *game* which is a enum and must be one of the values "Magic", "Pokemon" or "Yugioh", and a required parameter *cardId* which is an integer representing the unique identification of the *card*. It will return a single *Card* object in case of success and in case of invalid argument provided (game == null or cardId <= 0) it will return a message error.

Exchanges operations:

addProposal

```
public boolean addProposal(String token, String receiverUserEmail,
                           List<PhysicalCard> senderPhysicalCards,
                           List<PhysicalCard> receiverPhysicalCards)
    throws AuthException, InputException
```

- Purpose: Add a new proposal in database.
- Input: *token, receiverUserEmail, list of senderPhysicalCards, list of receiverPhysicalCards*
- Output: *boolean*

This method allows the user to add a trade proposal to the database, after selecting from their deck and from the deck of the recipient of the proposal, the cards they want to trade.

getProposal

```
public ProposalPayload getProposal(String token, int proposalId)
    throws AuthException, InputException, ProposalNotFoundException
```

- Purpose: Given a Proposal id, returns the proposal details
- Input: *token, proposalId*
- Output: *ProposalPayload* (Card lists are PhysicalCardWithName)

This method allows the client to retrieve the data related to a proposal. Given a proposal ID, it is requested to return the entire object related to it. This allows to populate the page of each individual proposal in order to display it to the recipient user and to allow him to evaluate it before accepting or rejecting it.

acceptProposal

```
public boolean acceptProposal(String token, int proposalId)
    throws AuthException, ProposalNotFoundException
```

- Purpose: Given a Proposal id, perform exchange in owned user decks and clean the wished ones.
- Input: *token, proposalId*
- Output: *boolean*

This method is used to allow the user to accept the received trade proposal. It can only be called by a user who is the recipient of the proposal and not the sender. If there is an exception where the sender is able to call this method, the server will check the sender's email and return an error as the sender cannot accept the proposal made by them.

refuseOrWithDrawProposal

```
public boolean refuseOrWithdrawProposal(String token, int proposalId)
    throws AuthException, InputException, NullPointerException,
ProposalNotFoundException
```

- Purpose: Given a Proposal id, delete the proposal on the database to refuse it.
- Input: *token, proposalId*
- Output: *boolean*

This method can be called by both users involved in a trade proposal. For the proposing user, it will work as a method to withdraw from the previously made proposal, while for the receiving user it will be a method to reject the received proposal.

getProposalListReceived & getProposalListSent

```
public List<Proposal> getProposalListReceived(String token) throws AuthException
```

```
public List<Proposal> getProposalListSent(String token) throws AuthException
```

- Purpose: Given a User token, retrive information about the received / sent proposals.
- Input: *token*
- Output: *List of proposals*

These two methods are used by the client to populate the user's Exchanges page. When called, they respectively check for received and made proposals, returning two lists with the corresponding objects inside, thus correctly populating the lists on each user's page.