

DiscordBotRPG - FightRPG

Jean-Charles TECHER, Anthony LEBIAN

28 avril 2018

Résumé

Dans le cadre de l'UE de développement mobile 2, j'ai décidé d'étendre la portée d'un projet déjà en cours en lui développant des applications mobiles (iOS et Android).

Introduction

DiscordBotRPG ou FightRPG est un gros projet débuté en Janvier 2018, c'est un mmorpg textuel inspiré des jeux par navigateurs, disponible par le biais de [Discord](#), qui est un logiciel de VoIP (ex : skype) mais qui permet aussi d'héberger gratuitement des serveurs vocaux et textuels. Discord met à disposition des développeurs une [API](#) et un système de bot qui nous permet de réaliser des choses de manière automatique. Ce bot a pour but final d'être l'un des bots les plus utilisés de la plate-forme. Nous allons voir au fil de ce rapport, comment est développé le serveur, quelles sont les technologies utilisés. Puis nous approfondirons le développement des applications mobiles, respectivement Android et iOS.

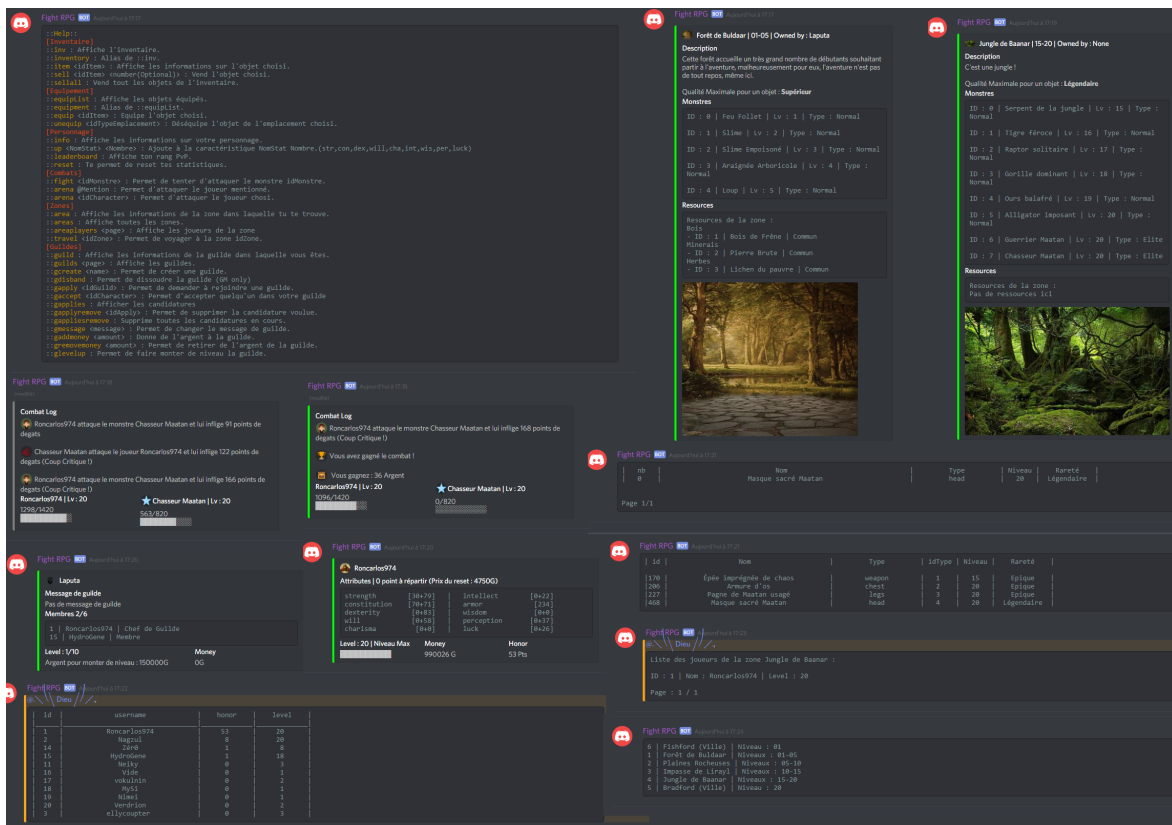
1 Serveur

1.1 Les technologies utilisées

Le serveur est développé en JavaScript avec [Node.js](#) et utilise la librairie [Discord.js](#) pour communiquer avec l'api de discord. Une base de données MySQL est aussi utilisé pour la persistance des informations. Express.js est aussi utilisé pour pouvoir avoir un serveur web.

1.2 Les fonctionnalités

Le bot dispose des fonctionnalités les plus basiques pour un mmorpg, comme par exemple, l'évolution du personnage, les combats ou encore la gestion de guildes. Je ne vais pas plus détaillé son fonctionnement, sachez cependant que le code est disponible dans la partie Serveur du bitbukket et que si vous avez des questions n'hésitez pas à me les poser. Voici une image résumant grossièrement ses fonctionnalités actuelles.



1.3 API

Pour pouvoir communiquer avec les applications mobiles le serveur met ainsi à son tour en place une api, en utilisant Express.js. Toutes les routes sont préfixées pas /api/. Voici la liste des routes ainsi que les fonctions qu'elles assurent :

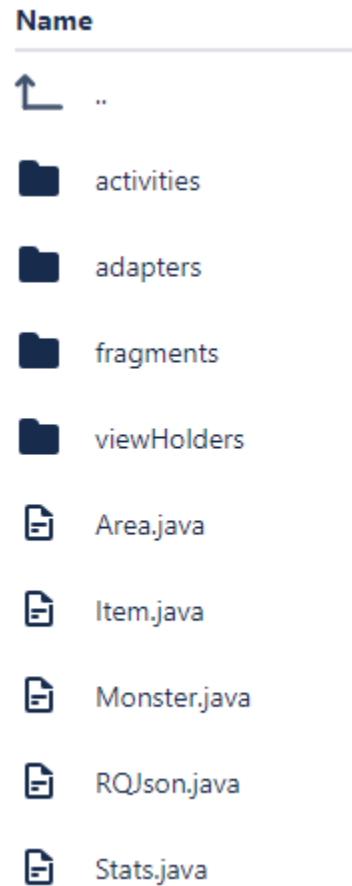
- GET - / - Test Connexion
- GET - /onlineplayers - Debug/Test Only (List of PPlayers)
- GET - /areas - Renvoi un JSON de la liste des zones par catégories
- GET - /area - Renvoi un JSON de la zone dans lequel le joueur est
- GET - /character - Renvoi un JSON avec les informations du joueurs
- GET - /character/inventory - Renvoi un JSON de l'inventaire du personnage
- GET - /character/item - Renvoi un JSON de l'objet demandé
- GET - /character/equipment - Renvoi un JSON avec les informations des objets équipés
- POST - /character/upstat - Permet de monter une caractéristique
- POST - /character/reset - Permet de réinitialisé ses caractéristiques
- POST - /character/sellitem - Permet de vendre un objet de l'inventaire
- POST - /character/sellallitems - Permet de vendre tous les objets de l'inventaire
- POST - /character/equip - Permet d'équiper un objet
- POST - /character/equip - Permet déséquiper un objet
- POST - /character/travel - Permet de voyager d'une zone à une autre
- POST - /fightpve - Permet de combattre un monstre / Renvoi un json contenant les informations du combat

Pour que les requêtes puissent être acceptés par le serveur, il faut que le joueur fasse passer un identifiant unique qu'il peut récupérer grâce au bot. Puis pour certaines routes il y a des paramètres à faire passer.

2 Android Studio - Développement en Java

2.1 Architecture

Avant toute chose voici un aperçu de la structure de mon application :



- activities - Contient toutes les activités de l'application
- adapters - Contient les adapters (Utilisés par les listes / tabbed activities)
- viewHolders - Contient les viewHolders (Utilisés par les listes / tabbed activities)
- fragments - Contient les fragments d'activités, notamment pour la tabbed activity
- Area,Item,Monster,Stats - Objets pour représenter certains objets json récupérés (utilisés par les listes)
- RQJson - Objet qui va nous permettre de faire les requêtes vers le serveur web et de récupérer la réponse en json

2.2 Requêtes à l'api

Comme vous l'avez remarqué il m'a fallu pouvoir faire des requêtes à un serveur web, pour ce faire j'ai utilisé un objet dédié à ça RQJson :

```

private RQJson(Context context) {
    mCtx = context;
    mRequestQueue = getRequestQueue();

    mImageLoader = new ImageLoader(mRequestQueue,
        new ImageLoader.ImageCache() {
            private final LruCache<String, Bitmap>
                cache = new LruCache<String, Bitmap>(20);

            @Override
            public Bitmap getBitmap(String url) {
                return cache.get(url);
            }

            @Override
            public void putBitmap(String url, Bitmap bitmap) {
                cache.put(url, bitmap);
            }
        });
}

public static synchronized RQJson getInstance(Context context) {
    if (mInstance == null) {
        mInstance = new RQJson(context);
    }
    return mInstance;
}

private RequestQueue getRequestQueue() {
    if (mRequestQueue == null) {
        // getApplicationContext() is key, it keeps you from leaking the
        // Activity or BroadcastReceiver if someone passes one in.
        mRequestQueue = Volley.newRequestQueue(mCtx.getApplicationContext());
    }
    return mRequestQueue;
}

```

Dans les faits il ne fait qu'utiliser une librairie qui s'appelle Volley pour faire des requêtes. Il s'utilise de la façon suivante :

```

public void connect(View view) {
    String url = baseUrl + "api?secretid=" + input_token.getText().toString(); -> ON DEFINI L'URL, C'EST ICI QUE VOUS VERREZ LES PARAMETRES DES REQUÊTES
    JsonObjectRequest jsonObjectRequest = new JsonObjectRequest -> ON CREER LA REQUÊTE
        (Request.Method.GET, url, null, new Response.Listener<JsonObject>() { ON DEFINI UN LISTENER EN FONCTION ANONYME

        @SuppressWarnings("CommitPrefEdits")
        @Override
        public void onResponse(JsonObject response) {
            // Hide error
            text_error.setVisibility(View.GONE);
            // Hide login button and show reset button
            btn_connection.setVisibility(View.GONE);
            btn_resettoken.setVisibility(View.VISIBLE);
            btn_start.setVisibility(View.VISIBLE);

            // Hide input text
            input_token.setVisibility(View.GONE);

            // Write Welcome message | connected
            title_login.setText(getString(R.string.info_connectedjoinworld));

            SharedPreferences.Editor editor = token.edit();
            editor.putString("token", input_token.getText().toString()); -> NOTEZ ICI L'UTILISATION DES SHARED PREFERENCES POUR LA PERSISTANCE DU TOKEN
            editor.apply();
        }
    }, new Response.ErrorListener() {

        @Override
        public void onErrorResponse(VolleyError error) { -> S'IL Y A UNE ERREUR SUR LA REQUÊTE
            // Show error when attempting to connect with bad id
            text_error.setText(getString(R.string.error) + " : " + getString(R.string.error_connectionfailed) + ".");
            text_error.setVisibility(View.VISIBLE);
        }
    });

    //jsonObjectRequest.setRetryPolicy(new DefaultRetryPolicy(3000,DefaultRetryPolicy.DEFAULT_MAX_RETRIES, DefaultRetryPolicy.DEFAULT_BACKOFF_MULT));

    RQJson.getInstance(this).addToRequestQueue(jsonObjectRequest); -> ICI ON AJOUTE A NOTRE OBJET LA REQUÊTE JSON QUI VA S'EXECUTER EN ASYNC
}

```

2.3 Utilisation des listes / TabbedActivity

Pour plusieurs activités j'ai dû utiliser des listes, comme par exemple l'inventaire. Pour ce faire j'ai utilisé des adapters et des viewholders pour chacune des listes. Le rôle de l'adapter et de prendre une liste d'objets et de générer la vue globale de la liste contenant toutes les lignes. Il est aidé par une viewholder qui va pour chaque objet bind une vue et y mettre des informations.

Les TabbedActivities marchent de la même façon que les listes à ça près que ce ne sont plus des viewholder qui vont gérer chacun des onglets mais des fragments qui sont en quelques sortes des activités à part entière dont on ajoute la vue de manière dynamique pour chaque onglets.

2.4 Aperçu de l'application

Voici comme pour le bot un résumé des visuels de l'application.


















3 XCode - Développement en Swift

3.1 Architecture

Cette fois l'architecture est un peu différente, en effet il n'est plus question de séparer en fonction des classes mais en fonction des fonctionnalités.

- Area/Character/Fight/Items - Contiennent tous les trois les classes des fonctionnalités liés aux différentes catégories.
- GFunc.swift - Contient une extension pour la classe UIColor (pour pouvoir init avec une chaîne de caractères du type #FFFFFF) et une pour la classe UIImageView pour pouvoir charger une image depuis un serveur distant.
- GVariables.swift - Contient une variable globale (L'adresse url du serveur à contacter)
- LoginScreen/MainMenu/MainMenuController sont les seuls à ne pas avoir de groupes.

Voici un aperçu de la structure :

Name	
	..
	Area
	Assets.xcassets
	Base.lproj
	Character
	DiscordBotRPG.xcdatamodeld
	Fight
	Items
	AppDelegate.swift
	GFunc.swift
	GVariables.swift
	Info.plist
	LoginScreen.swift
	MainMenu.swift
	MainMenuController.swift

3.2 Requêtes à l'api

Comme pour Android il m'a fallu utiliser une classe particulière pour pouvoir accéder à l'api. Voici comment j'ai fais :

```

func sellAll() {
    var token = UserDefaults.standard.string(forKey: "token")
    if(token == nil) {
        token = ""
    }
    var text: String = GVariables.url + "character/sellallitems?secretid=" + token! -> ICI JE DEFINIS L'URL
    text = text.replacingOccurrences(of: " ", with: "%20") -> JE REMPLACE LES ESPACES PAR %20
    let session = URLSession(configuration: .ephemeral, delegate: nil, delegateQueue: OperationQueue.main)
    var url = URLRequest(url: URL(string: text)!) -> JE CREER UNE SESSION PUIS UNE REQUETE
    url.httpMethod = "POST"
    let task = session.dataTask(with: url, completionHandler: { (data: Data?, response: URLResponse?, error: Error?) -> Void in
        guard let data = data else { -> S'IL N'ARRIVE PAS A SE CONNECTER
            self.alerterr(title: "Oups", message: "Impossible de se connecter au serveur !")
            return
        }
        guard let json = try? JSONSerialization.jsonObject(with: data, options: .mutableContainers) else { -> TRANSFORMATION EN JSON
            self.alerterr(title: "Oups", message: "Impossible de récupérer d'informations sur le serveur !") -> SI LE JSON INCORRECT ON
            return -> ABANDONNE LA REQUETE
        }
        if let dict = json as? [String: Any] { -> JE REAGIT A LA REPONSE EN JSON
            if(dict["error"] != nil) {
                self.alert(title: dict["error"] as! String, message: "")
            } else {
                self.maxPage = 1
                self.actualPage = 1

                self.navigationItem.prompt = String(format: "Page %d / %d", self.actualPage, self.maxPage)
                self.title = "Inventaire (Vide)"
                let itemvalue = dict["itemValue"] as! Int
                self.alert(title: String(format: "Vous avez vendu tous vos objets pour : %d G", itemvalue), message: "")
                self.inventory.removeAll()
                self.tableView.reloadData()
            }
        }
    })
    task.resume() -> JE LANCE LA REQUETE
}

```

3.3 Utilisation des TableView

Pour pouvoir afficher des listes j'ai décidé d'utiliser des tableviews. Prenons par exemple l'inventaire :

- Je défini une structure Item ainsi qu'un Array<Item>

```

var inventory : Array<Item> = []

struct Item {
    let id : Int
    let title: String
    let subtitle: String
    let color: String
    let image: String
}

```

- Je load les items, puis je update la table view avec .reloadData()
- Je défini que lorsque l'on clique sur la cellule on se dirige vers la vue de l'objet

```

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    //sliderButton(statIndex: indexPath.row)
    let mainStoryboard = UIStoryboard(name: "Main", bundle: Bundle.main)
    guard let itemViewController = mainStoryboard.instantiateViewController(withIdentifier: "ItemViewController") as? ItemViewController else {
        return
    }

    itemViewController.idItem = String(inventory[indexPath.row].id)
    navigationController?.pushViewController(itemViewController, animated: true)
}

```


4 Conclusion

Pour conclure le développement android et iOS sont très différents, mais très intéressants, l'utilisation d'une api pour récupérer des informations, permet d'avoir une perspective différente par rapport à une application ou toute la logique se fait directement en local. Cela permet aussi de n'avoir à programmer qu'une seule fois la logique pour pouvoir la distribuer à n'importe quelle plate-forme de la même manière. Nous avons vu un développement dépassant le simple développement local d'une application, notamment avec une cette interaction client/server.

4.1 Notes à part

Le développement Android a évoluer et peut maintenant se faire en Kotlin, ce langage est très intéressant et je pense que pour les prochains étudiants il sera opportun d'étudier ce langage (Surtout qu'ils ont normalement déjà fait du Java). Après avoir développé sur iOS je me rends compte que supprimer son étude serait sûrement une erreur, l'environnement de travail sous mac pour iOS est très complet et montre un langage plutôt exotique et de ce fait une façon un peu différente de réfléchir.