Manuel Technique

Hisoka challenge

QUETIER Thomas, WALCAK Ladislas, RIBARDIERE Tom

I -

Serveur distant et API **4**

II - Base de donnée local et RequestWraper **8**

1 - Base de données locale 8
2 - RequestWrapper 9

III - Les activités et leurs méthodes 10
1 - Au lancement 10
A - OnCreate 10
B - OnResume 11
2- Les activités liées aux challenges 14
A - Affichage 14
B - Détail d'un challenge 17
C - Participation 18
D - Le parcours des participations d'un challenge 18
3 - Le profil 20
4 - L'inscription, connexion et déconnexion 20
A - Inscription 20
B - Connexion 20
C - Déconnexion 22

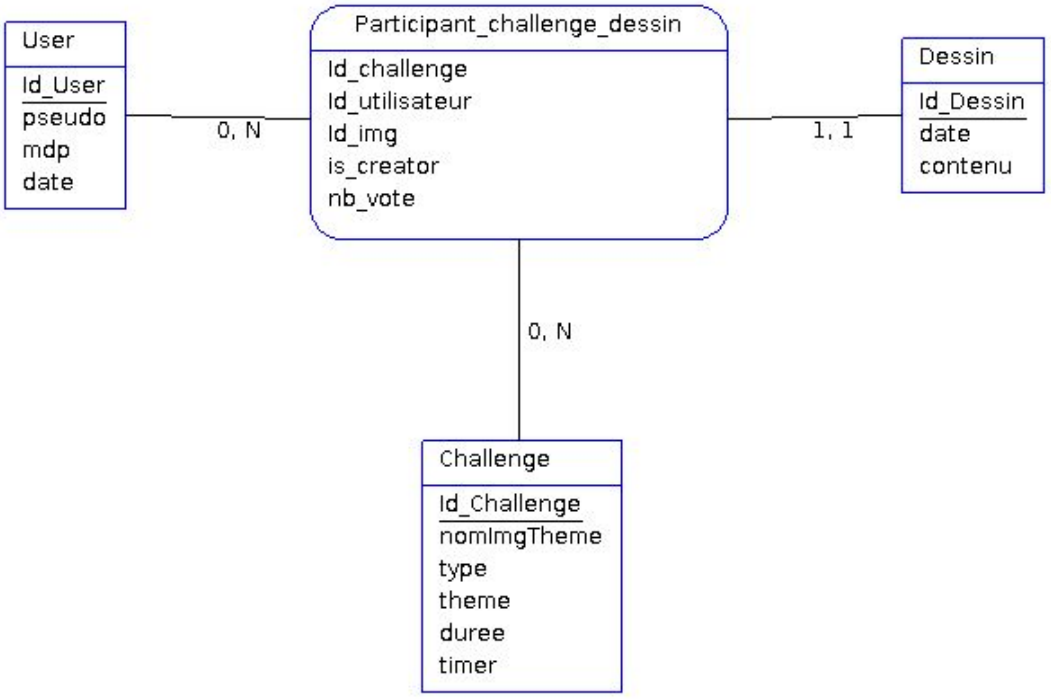
IV - Perspectives d'améliorations 22

Ce document répertorie les fonctions principales de l'application, ainsi que les classes et leurs fonctionnalités. La javadoc est disponible en plus de ce document.

I - Serveur distant et API

Pour ce qui est du serveur distant de l'application, il a été développé sous Python grâce au framework web Flask. Le serveur à été conçu comme une API CRUD (Create, Read, Update, Delete) pour manipuler les informations présentes dans la base de données, et ne fait quasiment aucun traitement sur les informations.

La base de données est composée des 4 entités User, Challenge, Drawing et Participation. Grâce à ces éléments, l'on peut correctement modéliser les participations des utilisateurs de l'application aux différents challenges.



MCD de la base de données distante

Pour manipuler les informations de la base de données, une série de routes a été créée. Toutes ces routes ont comme préfix “/api/”, et sont pour la plupart suivies par l’entité concernée, puis pour finir l’action souhaitée (Ex: /api/user/save). De plus, un certain nombre de routes supplémentaires ont été créées, comme par exemple la route /api/login, permettant la connexion d’un utilisateur.

Afin d’utiliser cette API, toutes les requêtes doivent contenir dans leur entête un champ *apiKey*. La clef d’API doit être définie sur le serveur dans un fichier *.env*, dans la variable API_KEY. Lors de toutes les requêtes, la première action du serveur est de tester la présence et la validité de cette clef. Si la clef n’est pas présente ou pas valide, le serveur retournera une réponse 403, avec {'error': 'Invalid api_key'} dans le body.

Pour finir, afin de ne pas avoir à gérer l’envoi et le téléchargement d’images dans la base de données, nous avons décidé d’utiliser l’API du site Imgur, permettant d’envoyer une image vers leurs serveurs afin qu’elle y soit hébergée. Nous ne stockons dans la base de données que le lien vers cette image.

La liste des routes :

Route	Method	Paramètres	Utilisation
/login	POST	username password	Permet la vérification des informations de connexion d'un utilisateur
/password	POST	username old new	Permet la modification du mot de passe d'un utilisateur
/full	GET		Récupération de l'ensemble des informations contenues dans la base de données
/update	GET	max_drawing max_challenge	Permet d'obtenir toutes les nouvelles informations de la base de données (les deux paramètres permettent de ne pas obtenir tous les Challenge et Drawing déjà présent en base de données locale)
ENTITY/getall	GET		Permet d'obtenir la table *ENTITY* au complet
ENTITY/get	GET	User: - username Drawing & Challenge: - id Participation - u_id - d_id - c_id	Permet d'obtenir l'entité correspondant à la clef primaire donnée
ENTITY/getwhere	POST		Permet d'obtenir toutes les entitées correspondant aux critères passés dans le corps de la requête
ENTITY/save	POST		Permet de sauvegarder dans la base de données distante l'entité passé dans le corps de la requêtes (au format JSON)
ENTITY/delete	GET	User: - username Drawing & Challenge: - id Participation - u_id - d_id - c_id	Permet de supprimer l'entité correspondant à la clef primaire donnée

Liste des erreurs:

Erreur	Code
Clef API invalide	403
Argument(s) manquants	400
Impossible de sauvegarder l'entité (pour les routes *save*)	409
Utilisateur inconnu	400
Participation inconnue	400
Mauvais mot de passe	400
Requête valide mais rien n'a été trouvé dans la base de données	204
Requête valide avec données	200

II - Base de donnée local et RequestWraper

1 - Base de données locale

Toutes les méthodes de manipulation de la base de données locale ont été regroupées dans une classe singleton nommée DB. Cette classe, qui implémente l'interface SQLiteOpenHelper fournie par Android afin de faciliter la gestion de la base de données, doit être initialisée grâce à la fonction *init()* lors du lancement de l'application.

La base de données locale a été créée dans le but d'agir comme un cache de la base de données distante, permettant l'affichage d'informations, même en cas où l'utilisateur n'aurait pas accès à internet. Elle possède donc le même schéma que la base de données distante, à la seule différence que les mots des passe des utilisateurs n'y sont pas stockés, pour des raisons de sécurité.

Afin de faciliter la manipulation des données locales, des classes équivalentes à chacunes de tables de la base de données ont été créées, permettant également de mimer la base de données distante, et ainsi harmoniser toute la partie base de données de l'application.

Comme dit plus haut, les informations de connexion des utilisateurs ne sont pas stockées dans la base de données. Lors de la création d'un compte, le mot de passe de l'utilisateur est hasher grâce à l'outil BCrypt, puis envoyé à la base de données distante, qui ne connaît donc pas le véritable mot de passe. Ne sont gardés dans la base de données locale seulement le nom de compte de l'utilisateur, ainsi que le hash de son mot de passe.

2 - RequestWrapper

Afin de créer une couche d'abstraction pour l'envoi et la réception de requêtes vers le serveur et l'API d'Imgur, afin de les faciliter et d'harmoniser le code, nous avons créé la classe RequestWrapper. C'est dans cette classe que se trouve toutes les fonctions d'envoi de requêtes vers l'extérieur de l'application.

Depuis Android API 30, les requêtes utilisant l'interface Android *AyncTask* sont dépréciées. Nous avons donc utilisé la bibliothèque *AndroidNetworking* afin de nous simplifier le travail. Cette bibliothèque fournit un ensemble de fonctions permettant la création, construction et l'envoi de requêtes. Grâce à un système de *callbacks* très similaire au promesses JavaScript, cette bibliothèque permet de passer en paramètre une fonction qui sera exécutée lors de la résolution de cette requêtes, mais également une fonction qui sera exécutée si la requête échoue (HTTP Status code différent de 2XX).

Cela permet donc à l'application de ne pas mettre totalement en pause le Thread UI lors de l'attente de réponse de la requête et donc de pouvoir effectuer du traitement en parallèle.

La fonction la plus utilisée est la fonction *get()*, permettant de synchroniser la base de données locale avec la base de données distante. Toutes les utilisations, ou du moins toutes les utilisations nécessitant une base de données à jour, devraient être faites au sein du *callback* d'une requête *get()* (Notamment la connexion d'un utilisateur).

III - Les activités et leurs méthodes

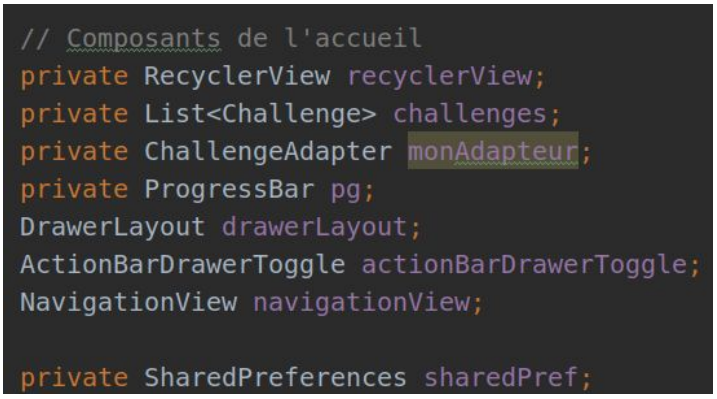
1 - Au lancement

Au lancement de l'application, la première page à apparaître sera notre page d'accueil ou comme nous l'avons nommée, la *MainActivity*.

Son rôle est d'être le point névralgique de l'application, c'est-à-dire, que toutes les autres activités doivent être accessibles par cette page principale.

Premièrement, nous allons commencer par parler des attributs qui se trouvent dans cette classe.

Pour la plupart des attributs, ceux-ci sont pour nous aider à gérer les éléments graphiques tels que le recyclerView ou la progressBar. Mais nous en avons aussi qui nous permettent d'interagir avec les SharedPref.



```
// Composants de l'accueil
private RecyclerView recyclerView;
private List<Challenge> challenges;
private ChallengeAdapter monAdapteur;
private ProgressBar pg;
DrawerLayout drawerLayout;
ActionBarDrawerToggle actionBarDrawerToggle;
NavigationView navigationView;

private SharedPreferences sharedPref;
```

Attributs de MainActivity

A - OnCreate

Ensuite nous avons le *OnCreate*, qui nous sert pour initialiser l'application, nous récupérons tous les éléments XML que nous avons besoins, initialisons la BD local, mettons en place la vérification de connexion et nous référençons les options possibles (activités accessible via le tiroir de navigation).

```
navigationView.setNavigationItemSelectedListener(menuItem -> {
    switch (menuItem.getItemId())
    {   // si case vrai alors lancé une autre activity
        case  R.id.nav_connexion:
            Intent intent = new Intent( packageContext: this, ConnexionView.class);
            startActivity(intent);
            finish();
            break;
```

Switch pour les choix du menu

Il reste une chose que nous faisons dans le OnCreate de notre accueil, nous gérons la nôtre toolbar via une fonction que nous avons faite.

```
/**
 * configuration de ma Toolbar et de tiroir de navigation
 */
public void setUpToolbar() {
    drawerLayout = findViewById(R.id.drawerLayout);
    actionBarDrawerToggle = new ActionBarDrawerToggle( activity: this, drawerLayout, R.string.app_name, R.string.app_name);
    actionBarDrawerToggle.setDrawerIndicatorEnabled(true);
    drawerLayout.addDrawerListener(actionBarDrawerToggle);
    actionBarDrawerToggle.getDrawerArrowDrawable().setColor(getResources().getColor(R.color.black));
    actionBarDrawerToggle.syncState();
    getSupportActionBar().setHomeButtonEnabled(true);
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    getSupportActionBar().setHomeAsUpIndicator(R.drawable.ic_baseline_dehaze_24);
}
```

Gestion de la toolbar

Mais revenons sur comment nous faisons pour savoir si un utilisateur est connecté ou non. Nous utilisons tout simplement les SharedPreferences, avec un champ dédié au pseudo de notre utilisateur connecté.

```
// Récupération des ShardPref
this.sharedPref = getSharedPreferences( name: "session", Context.MODE_PRIVATE);

// test pour savoir si l'user est connecté
if( !(this.sharedPref.getString( key: "username", defValue: "").equals(""))){
    //If user connecté
    navigationView.getMenu().setGroupVisible(R.id.groupeConnecter, visible: true);
    navigationView.getMenu().setGroupVisible(R.id.groupeDeco, visible: false);
}else{
    navigationView.getMenu().setGroupVisible(R.id.groupeConnecter, visible: false);
    navigationView.getMenu().setGroupVisible(R.id.groupeDeco, visible: true);
}
```

Gestion de l'utilisateur connecté

Et en fonction de si notre utilisateur est connecté ou non alors nous ne lui affichons pas les même choix dans le tiroir de navigation.

B - OnResume

Dans le OnResume, on peut trouver l'affichage de notre recyclerView et presque tout est géré par la fonction *loadChallenge*.

```
@Override
protected void onResume() {
    super.onResume();

    // Test si user connecté
    if( !(this.sharedPref.getString( key: "username", defValue: "").equals(""))){
        //If user connecté
        navigationView.getMenu().setGroupVisible(R.id.groupeConnecter, visible: true);
        navigationView.getMenu().setGroupVisible(R.id.groupeDeco, visible: false);
    }else{
        navigationView.getMenu().setGroupVisible(R.id.groupeConnecter, visible: false);
        navigationView.getMenu().setGroupVisible(R.id.groupeDeco, visible: true);
    }

    // Mise à jour base de données locale
    loadChallenge();
}
```

Gestion de l'affichage et du rafraîchissement des challenges

Nous pouvons constater que nous avons encore un test pour savoir si l'utilisateur est toujours connecté, ensuite l'appel à la fonction *loadChallenge* qui nous permet d'afficher et de mettre à jour l'affichage des challenges dans le recyclerView de la page d'accueil.

Dans la fonction *loadChallenge*, nous avons une synchronisation de la base de données distante et locale pour que nous soyons sur d'afficher tous les challenges.

Le *OnResponse* du *RequestWrapper* ne s'exécute que si la requête réussit, c'est la même chose avec le OnError, qui ne s'exécute que si la requête échoue.

```
/**
 * Gestion de l'affichage des challenge
 */
public void loadChallenge() {
    // Synchronisation BD local et distante
    new RequestWrapper().get(new Callback() {
        @Override
        public void onResponse() { // Si les 2 BD arrivent à être synchro alors

            // Mise à jour de l'affichage des challenges après mise à jour de la base de données locale
            displayChallenges();

            pg.setVisibility(View.GONE); // disparition de la progress bar = fin synchro BD
        }

        @Override
        public void onError(ANError error) {
            Toast.makeText(getApplicationContext(), "Couldn't load challenges from server", Toast.LENGTH_LONG).show();
            pg.setVisibility(View.GONE); // disparition de la progress bar = fin synchro BD
        }
    });
    pg.setVisibility(View.VISIBLE); // tant que synchro pas fini progress bar visible
```

Fonction d'affichage des challenges

Nous voyons que dans le code nous appelons 2 fois la fonction *displayChallenges*, elle nous permet de ne pas dupliquer le code et de produire un code plus lisible.

Il ne faut pas oublier, la *progressBar* qui nous permet d'informer l'utilisateur sur le statut de la requête, si elle est toujours en cours ou si elle est finie si celle-ci disparaît.
Si la requête ne fonctionne pas alors que nous affichons un Toast pour informer l'utilisateur de l'erreur qui vient d'arriver.

Pour la fonction *displayChallenge*, c'est elle qui nous permet de mutualiser du code, elle nous permet de faire une requête à la BD locale, de trier la liste reçue par date et de gérer l'affichage des challenge dans le recyclerView.

```
public void displayChallenges() {
    challenges = DB.getInstance().getAllChallenges(); // récupération de tous les challenges

    sortList(challenges); // trie challenge par date

    // affectation challenge adapteur
    monAdapteur = new ChallengeAdapter(challenges);
    // gestion recyclerView
    recyclerView.setLayoutManager(new LinearLayoutManager( context: this));
    recyclerView.setAdapter(monAdapteur);
    // mise en place du listener sur l'adapteur
    monAdapteur.setOnItemClickListener(position -> {
        Challenge chall = challenges.get(position);
        Intent gotoChall = new Intent( packageContext: this, onChallenge.class);
        gotoChall.putExtra( name: "idchall", chall.getId());
        startActivity(gotoChall);
    });
}
```

Gestion de l'affiche des challenge

Nous pouvons voir que dans le code précédent, nous avons un sortList qui est une fonction que j'ai fait pour que l'on puisse facilement trier une liste de challenge en fonction de leur date de fin. J'implémente un comparateur dans la fonction, qui me permet d'avoir les challenge les plus récents au début de la liste et les plus anciens à la fin.

```java
/**
 * Trier la liste de challenge en fonction des date
 * @param list, liste de challenge
 */
public void sortList(List<Challenge> list){
    list.sort((o1, o2) -> {
        String dateString1 = o1.getDate();
        String dateString2 = o2.getDate();
        DateTimeFormatter formatter = DateTimeFormatter.ISO_DATE_TIME;
        LocalDateTime dateTime1 = LocalDateTime.parse(dateString1, formatter);
        LocalDateTime dateTime2 = LocalDateTime.parse(dateString2, formatter);
        if(dateTime1.isAfter(dateTime2)) {
            Log.d( tag: "Sort", msg: "1");
            return -1;
        } else if(dateTime1.isBefore(dateTime2)) {
            Log.d( tag: "Sort", msg: "-1");
            return 1;
        } else {
            Log.d( tag: "Sort", msg: "0");
            return 0;
        }
    });
}
```

Fonction de trie des challenges

2 - Les activités liées aux challenges

Les challenges sont les éléments principaux de l'application. Tout d'abord, ils sont représentés en objet par l'entitée Challenge dans la base de données. Pour pouvoir utiliser l'application correctement, on doit pouvoir les afficher, les créer et y participer.

A - Affichage

D'abord, concernant l'affichage des challenges, ils sont visibles depuis l'activité principale (*MainActivity*). À ce stade l'utilisateur peut cliquer sur un challenge pour lancer l'activité *onChallenge* (grâce au *OnItemClickListener* défini dans *ChallengeAdapter*)

```
// un listener sur le click des dessins, peut importe si dessin ou thème de challenge
dessinAdapter.setOnItemClickListener(
        position -> {
            // Lancer l'activité de la descritpion d'un challenge si click sur un dessin du
            // dit challenge
            Intent gotoChall = new Intent(this.context, onChallenge.class);
            gotoChall.putExtra( name: "idchall", challenge.getId()); // passage de l'id du challenge
            context.startActivity(gotoChall);
        }
);
```

Changement d'activité lors du clique sur un challenge

Nous avons plusieurs Adapteurs qui nous permettent d'afficher proprement les différents objets avec lesquels nous travaillons.
Commençons avec l'adapteur des challenges, celui qui est utilisé dans l'accueil (Main Activity). Dans cette classe nous avons 2 attributs, une liste de challenge et un *clickListener*.

Quand l'on veut afficher des challenges, il faut que l'on crée un *ChallengeAdapteur* en lui passant une liste de challenge. Il s'occupera automatiquement de les afficher via la fonction OnBindViewHolder.

```
/**
 * Lancement de l'affichage des challenges
 * @param holder
 * @param position, int qui represente l'index du challenge traité
 */
@Override
public void onBindViewHolder(@NonNull ChallengeAdapter.MyViewHolder holder, int position) {
    holder.display(this.challenges.get(position));
    Log.d( tag: "bonjour",  msg: "onBindViewHolder: display");
}
```

Fonction OnBindViewHolder

Nous pouvons voir que dans la fonction OnBindViewHolder, nous avons pour tous les éléments de la liste un appel à la fonction display de classe static MyViewHolder.

Cette fonction nous permet de décrire l'affichage d'un challenge précisément. J'affiche d'abord le titre puis je fais une requête à la BD pour récupérer les dessins d'un challenge et je crée une paire pour pouvoir garder les infos de l'image de thème du challenge.

Ensuite, je teste s'il y a des dessins dans le challenge, s'il y en a pas je remplis ma paire avec les infos du thème du challenge, et je rajoute un dessin fictif dans la liste des dessins du challenge. Par la suite, je trie la liste des dessins du challenge et je fais en sorte qu'il n'y ai que 5 dessins à afficher par challenge sur la page d'accueil.

Je crée un *ImageDessinAdapteur* en fonction du nombre de dessins du challenge.

Je n'oublie pas de mettre un listener sur l'adapteur de mes dessin pour que je puisse cliquer sur les dessins d'un challenge. Pour finir, j'initialise mon carrousel de challenge.

Quand l'on crée un *ImageDessinAdapteur*, pour l'affichage on lance la fonction *SetLocationData*, qui premièrement test si les composantes du thème sont null et ensuite remplie les éléments xml soit en fonction du thème soit d'une requête BD qui récupère la participation du dessin pour que je puisse afficher l'utilisateur qui a soumis la participation.

```
void setLocationData(Drawing dessin){
    // si les élements de la paire thème ne sont pas null alors
    // nous n'avons pas de dessin, et nous devons affiché le thème
    if ( !(Objects.isNull(theme.first) && Objects.isNull(theme.second))){
        Picasso.get().load(this.theme.first).into(imageView); // chargement de l'image avec le premier composant du thème
        textTitle.setText("Theme"); // affichage du Titre qui est Thème
        // Récupération et formatage de la date du thème
        LocalDateTime themeTime = this.theme.second;
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy', 'HH'h'mm");
        String dateTheme = themeTime.format(formatter);
        textLocation.setText(dateTheme);// Affichage de date
        itemView.findViewById(R.id.votes).setVisibility(View.GONE); // Disparition du vote, car l'on ne peut pas voter sur un 1 thème
    }else{ // si nous avons des dessins
        // Recherche dans la BD de la  participation du dessin au challenge
        HashMap<String, Pair<String, String>> mapParticipation = new HashMap<>();
        mapParticipation.put(Tables.PARTICIPATION_DRAWING_ID, new Pair(Tables.OPERATOR_EQ, dessin.getId().toString()));
        ArrayList<Participation> participations = new ArrayList<>(DB.getInstance().getParticipations(mapParticipation));

        // Recherche dans la BD de l'user qui a fait le dessin
        User user = DB.getInstance().getUserFromDrawing(dessin.getId());
        // affichage du dessin
        Picasso.get().load(dessin.getLink()).into(imageView);
        textTitle.setText(user.getUsername());
        textLocation.setText(dessin.getFormattedDate());
        // Mise en place de l'affichage des votes avec les infos de la participation
        textStarRating.setText(String.valueOf(participations.get(0).getVotes()));
    }
```

Affichage d'un drawing (dessin)

B - Détail d'un challenge

Une fois dans l'activité *onChallenge*, on récupère le challenge avec son identifiant depuis la base de donnée local (actualisé avec la base de donnée distante depuis la *MainActivity*). Grâce à cela, on remplit la vue avec les informations du challenge.

```
Challenge chall = db.getChallenge(this.idchall);
if(chall != null){
    TextView title = findViewById(R.id.nomChall);
    title.setText(chall.getName());
    Picasso.get().load(chall.getTheme()).into(iv);

    TextView timer = findViewById(R.id.timer);
    timer.setText(chall.getTimer()+"minutes");

    TextView dateFin = findViewById(R.id.dateFin);
    dateFin.setText("End the :"+chall.getFormattedDate());

    TextView desc = findViewById(R.id.textView2);
    desc.setText(chall.getDesc());
}else{
    TextView title = findViewById(R.id.nomChall);
    title.setText("Error : This challenge does not exist.");
}
```

Ensuite, on vérifie si l'utilisateur connecté à déjà participé. Si cela est le cas, alors le bouton "Participer" restera inutilisable.

```
new RequestWrapper().get(new Callback() {
    @Override
    public void onResponse() {
        //Check si la date de fin du challenge est dépassée
        Challenge challenge = DB.getInstance().getChallenge(chall.getId());
        DateTimeFormatter formatter = DateTimeFormatter.ISO_DATE_TIME;
        LocalDateTime dateTime = LocalDateTime.parse(challenge.getDate(), formatter);

        if( !(sharedPref.getString( key: "username", defValue: "").equals(""))) {
            if (LocalDateTime.now().isBefore(dateTime)){
                User user = DB.getInstance().getUser(sharedPref.getString( key: "username",  defValue: ""));

                HashMap<String, Pair<String, String>> map = new HashMap<>();
                map.put(Tables.PARTICIPATION_CHALLENGE_ID, new Pair(Tables.OPERATOR_EQ, String.valueOf(idchall)));
                map.put(Tables.PARTICIPATION_USER_ID, new Pair(Tables.OPERATOR_EQ, user.getUsername()));

                List<Participation> participations = DB.getInstance().getParticipations(map);
                if (participations.isEmpty()) {
                    //Si l'utilisateur n'a pas encore participer, alors il peut
                    findViewById(R.id.btnParticipe).setEnabled(true);
                }
            }
        }else{
            //Si l'utilisateur est déconnecté, alors il peut cliquer sur participer mais il sera redirigé
            //Vers la connexion
            findViewById(R.id.btnParticipe).setEnabled(true);
        }
    }
```

Dans onChallenge, test si l'utilisateur peut participer

Si l'utilisateur courant est déconnecté, alors le bouton Participer est cliquable, mais envoie vers l'inscription et non la participation.

Depuis cette activité, il est donc possible d'aller sur l'activité *ConnexionView* (si l'utilisateur est non connecté et clique sur Participer), sur *onParticiperChrono* (s'il n'a pas encore participé) et sur *ParcoursParticipation* (en cliquant sur Parcourir).

C - Participation

Activité qui affiche une timer a l'utilisateur, ainsi qu'un thumbnail du dessin à reproduire. Une fois le timer à 0, l'activité Caméra se lance automatiquement (*onFinish* du timer (objet *CountDownTimer*). Il est aussi possible pour l'utilisateur de lancer la prise de photo en cliquant sur le bouton caméra (fonction appelée : *onPhoto*). Une fois la photo prise, l'utilisateur est envoyé sur l'activité de confirmation. Une fois dans cette activité, il peut reprendre une photo si la précédente ne lui convient pas, et confirmer l'envoie (fonction *onConfirmer*).

D - Le parcours des participations d'un challenge

Pour le parcours des participants, Tout se déroule dans la classe ParcoursParticipation. Nous avons également, dans le OnCreate, comme dans le MainActivity, la récupération de tous les éléments xml, le test pour savoir si l'utilisateur est connecté et la gestion de la toolbar avec la gestion des choix dans le tiroir de navigation.

Nous avons aussi une synchronisation de la BD pour toujours être à jour sur les participation du challenge. Dans le OnResponse de cette synchronisation, nous avons une requête à la BD local qui récupère la liste des participations pour un challenge. Ensuite, on trie la liste par date et après je teste s'il y a bien des participations pour le challenge. J'adapte mon affichage en fonction.

```
// si des participation
if (!Objects.isNull(participations[0])) {
    if (participations[0].size() > 0) {
        // gestion de l'affichage des participation
        recyclerView.setVisibility(View.VISIBLE);
        pasParticipation.setVisibility(View.GONE);
        monAdapteur[0] = new ParticipationAdapteur(context, participations[0]);

        recyclerView.setLayoutManager(new GridLayoutManager(context, spanCount: 2));
        recyclerView.setAdapter(monAdapteur[0]);
    } else {// si pas de participation
        recyclerView.setVisibility(View.GONE);
        pasParticipation.setVisibility(View.VISIBLE);

        txt[0] = "There is no participation for this challenge";
        pasParticipation.setText(txt[0]);
    }
}
```

Gestion de l'affichage en fonction des participation d'un challenge

Si la requête ne fonctionne pas alors nous testons des codes d'erreurs pour afficher un message cohérent.

Pour l’affichage, j’utilise comme toujours un adapteurs qui gère l’affichage des participations et qui me permet également de gérer le clique sur une participation pour pouvoir augmenter le vote de cette participation.

```
SharedPreferences sharedPref = mContext.getSharedPreferences( name: "session", Context.MODE_PRIVATE);
if( !(sharedPref.getString( key: "username", defValue: "").equals(""))) { // si connecté
    holder.participationItem.setOnClickListener(v -> { // mettre un listener

        //TODO Use request
        try { // synchro BD local et distante
            new RequestWrapper().vote(mParticipation.get(position).toJson(), sharedPref.getString( key: "username", defValue: ""), new JSONObjec
                @Override
                public void onResponse(JSONObject response) {
                    try {
                        // gestion du vote
                        DB.getInstance().update(Participation.fromJson(response));
                        // remplacement de paraticipation avec l'ancien vote avec le nouveau
                        mParticipation.remove(position);
                        mParticipation.add(position,Participation.fromJson(response));
                        // maj affichage vote
                        holder.votes.setText(mParticipation.get(position).getVotes().toString());
                    } catch (JSONException e) {
                        Log.d( tag: "bonjour",  msg: "onResponse: salut");
                        e.printStackTrace();
                    }
                }
```

Gestion du vote sur une participation

J’utilise un RequestWrapper pour synchroniser la BD et être sur d’avoir le bon nombre de votes sur une participation. Ensuite, je modifie l’affichage de la participation en fonction de ce que me renvoie le OnResponse de la RequestWrapper.

3 - Le profil

Pour le Profil, nous ne pouvons modifier que le mot de passe de l'utilisateur connecté. Pour la page de base, je récupère juste le pseudo de l'utilisateur connecté via les SharedPreference pour l'afficher puis quand il clique sur le bouton "modifier mot de passe" alors je fais appel à une fonction qui me cache le bouton pour en afficher un autre, changer le texte et afficher un editText pour qu'il vérifie son mot de passe actuel et ensuite il pourra entrer son nouveau mot de passe qui sera envoyé à la BD distante. Il y aura un message de réussite si le nouveau mot de passe est bien accepté et un message d'erreur pour le cas contraire. D'ailleurs le mot d'erreur est personnalisé en fonction de l'erreur que nous rencontrons.

l'ordre de déclenchement des fonction pour le profil est le suivant :

1) verif_mdp
2) new_mdp
3) valider_new_mdp

4 - L’inscription, connexion et déconnexion

A - Inscription

L’activité *InscriptionActivity* gère l’inscription d’un nouvel utilisateur dans la base de données distante et locale. La fonction *onInscrire* consiste en une requête à l’api distante pour vérifier si l’username n’est pas déjà pris, et ensuite ajouter l’utilisateur dans la base de donnée distante. Il est ensuite ajouté dans la base de données locale.

Les différents code d’erreur renvoyés par l’api permette d’afficher un label d’erreur correspondants aux différents problèmes rencontrables (username déjà pris, erreur coté serveur) et ainsi afficher un message d’erreur cohérent à l’utilisateur.

B - Connexion

Une fois l’inscription réalisée avec succès, l’utilisateur est redirigé vers l’activité *ConnexionView* afin de pouvoir se connecter. C’est la fonction *onConnexion* qui gère les appels à l’api. Une fois les requête effectué, on récupère l’utilisateur et on le stock dans une sharedpreferences nommé “session” qu’on utlise dans les autre activitées pour voir si l’utilisateur courant est connecté (et savoir qui il est).

```
public void onResponse(JSONObject response) {
    try {
        //Une fois le mot de passe validé, on ajoute l'utilisateur
        //dans le shared pref session sous la clé "username"
        pg.setVisibility(View.INVISIBLE);
        SharedPreferences.Editor editor = sharedPref.edit();
        editor.putString("username", response.getString( name: "username"));
        editor.apply();
        finish();
        //Redirection vers l'accueil
        startActivity(home);
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Stockage de l’utilisateur dans la session

C - Déconnexion

L’activité *DeconnexionView* est responsable de la déconnexion. Cette activité vide la Sharedpreferences “session” et renvoie l’utilisateur sur l’activité principale (*MainActivity)*.

IV - Perspectives d’améliorations

Les choses que l’on pourrait ajouter sont :

- Une partie de statistiques avec le nombre de votes par participation ou juste le nombre de participation
- Une page qui recense toutes nos participations.
- Une page qui recense les challenges que nous avons créer

Ce qui pourrait être amélioré dans le code :

- Faire un code plus générique
- Faire plus de fonctions pour éviter le code redondant
- Optimisation de la complexité des fonctions
- Faire en sorte d’utiliser l'élément fictif ayant les bonnes informations de ma liste de dessin pour afficher le thème d’un challenge au lieu d'utiliser une paire en plus.
- Une meilleure gestion des requêtes vers la base de données distante. Moins de requêtes, et lors de la synchronisation de la base de données locale, éviter le plus possible d’obtenir des informations déjà contenues en local.
- Ne pas stocker le hash des utilisateurs en local.