Elias SAAD
Antoine POULAIN MAIN4

PROJET C++

Les Jeux Olympiques

Pour ce projet de développement en C++ sur le thème des jeux olympiques, nous avons eu envie de créer notre propre application de paris sportifs portant uniquement sur les épreuves olympiques. Un de nos objectifs d'implémentation -en plus de ceux requis- a été de fournir une approche réaliste du développement d'une application. Nous avons donc choisi de la séparer en un frontend et un backend pour simuler le comportement réel d'une application de paris sportifs en ligne reliée à un serveur et à une base de données. Nous nous sommes donc séparé les tâches : Elias pour le frontend et Antoine pour le backend. De plus, nous avons aussi porté une attention particulière sur les enjeux de sécurité notamment des mots de passe et de protection contre la falsification d'information côté frontend. C'est pour cela que nous avons travaillé main dans la main pour fournir une interface design et fonctionnelle liée à une API hébergée sur un serveur distant (une machine du LIP6). Nous commencerons par présenter le côté utilisateur de l'application, puis nous nous focaliserons sur les différents aspects techniques.

I] ASPECT UTILISATEUR :

D'un point de vue purement utilisateur, l'application n'est pas compliquée. Celui-ci se crée un compte à l'aide d'une adresse mail valide. Le client fournira aussi un pseudonyme et un mot de passe pour son compte. Ensuite, l'utilisateur peut importer des fonds dans l'application afin de pouvoir les utiliser pour commencer à parier sur les matchs en cours ou sur ceux débutant prochainement. Sur la page d'accueil, le client se verra proposer toute une sélection de paris disponibles. Il peut donc en choisir autant qu'il souhaite (à condition d'avoir les fonds suffisants évidemment). Il ne lui reste qu'à attendre que le match se termine pour récupérer ses gains ou pleurer son argent parti trop tôt. De plus, s'il possède un compte premium, il peut regarder le match directement dans l'application ou bien profiter d'autres avantages. Tout cela se résume dans le diagramme de flot suivant :

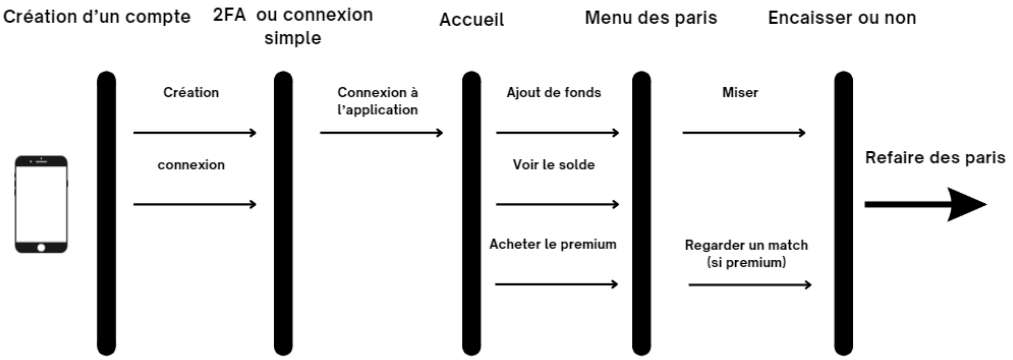


Figure 1 : Diagramme de flot dans l'application

II] ASPECT TECHNIQUE :

Pour présenter le procédé de création de notre application, il nous semble judicieux de commencer par la face cachée de l'iceberg, le backend. En effet, c'est sur celui-ci que nous nous sommes basés pour développer l'interface utilisateur étant donné que nous voulions garder le plus de contrôle possible côté serveur sur les actions que peut effectuer le client. Toujours dans une optique de sécurité et de vérification des agissements de l'utilisateur.
La principale utilité du serveur est l'implémentation de la logique d'arrière-plan, qui nous permet de faire la quasi-totalité des actions nécessaires au cheminement de l'utilisateur. Ces actions sont dirigeables depuis le frontend par le biais d'une API implémentée avec la bibliothèque C++ CROW. Ce sont les routes de l'API qui nous permettent d'agir selon les instructions envoyées par l'utilisateur. Voici à quoi ressemble globalement l'interface du serveur :

```
crow::SimpleApp app;
ServerManager::log("App initialized");

const int port=18080;

DatabaseManager DBM;
ServerManager::log("Database Manager initialized");

CROW_ROUTE(app, "/api/post").methods(crow::HTTPMethod::Post)([](const crow::request& req){···
});

CROW_ROUTE(app, "/api/signup").methods(crow::HTTPMethod::Post)([&DBM](const crow::request& req){//On rec
});

CROW_ROUTE(app, "/api/2fa").methods(crow::HTTPMethod::Post)([&DBM](const crow::request& req){···
});

CROW_ROUTE(app, "/api/signin/salt").methods(crow::HTTPMethod::Post)([&DBM](const crow::request& req){//c
});

CROW_ROUTE(app, "/api/signin").methods(crow::HTTPMethod::Post)([&DBM](const crow::request& req){//on rec
});

CROW_ROUTE(app, "/api/publicinfo").methods(crow::HTTPMethod::Post)([&DBM](const crow::request& req){···
});

CROW_ROUTE(app, "/api/bet").methods(crow::HTTPMethod::Post)([&DBM](const crow::request& req){···
});
```

Figure 2 : Capture d'écran de l'interface serveur

L'intérieur du code des routes suit globalement toujours le même principe. D'abord on récupère le corps de la requête HTTP provenant du frontend. On le passe à un format JSON correct. Ensuite on fait les traitements voulus avec (souvent un appel à la base de données). Enfin, si toutes les étapes précédentes ont été un succès, on peut renvoyer ce que nous avons besoin d'envoyer au frontend accompagné d'un code d'erreur 200 (OK). Si une des étapes n'a pas abouti correctement, on renvoie un code d'erreur 400. Pour nous aider à faire toutes ces conversions string/JSON et effectuer les appels à la base de données, nous utilisons deux classes mères ayant une classe fille chacune. D'abord la classe ServerManager (qui n'implémente que des méthodes static) permet tout simplement de faire des logs propres des succès et des erreurs. Voici un exemple de logs que nous retrouvons pendant les périodes d'activité du serveur :

```
(2024:02:01 19:10:58) App initialized
(2024:02:01 19:10:58) Open database successfully
(2024:02:01 19:10:58) Database Manager initialized
(2024:02:01 19:10:58) olympicstake@gmail.com : Data inserted successfully.
(2024:02:01 19:10:58) Listening on port : 18080
(2024-02-01 19:10:58) [INFO    ] Crow/0.1 server is running at 0.0.0.0:18080 using 8 threads
(2024-02-01 19:10:58) [INFO    ] Call `app.loglevel(crow::LogLevel::Warning)` to hide Info level logs.
(2024-02-01 19:11:01) [INFO    ] Request: 37.170.100.17:55302 0x562dea226ad0 HTTP/1.1 POST /api/signup
(2024:02:01 19:11:01) saadeliaswd@gmail.com : Data inserted successfully.
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   110    0     0  100   110      0    103  0:00:01  0:00:01 --:--:--   103
(2024:02:01 19:11:02) saadeliaswd@gmail.com : 2FA email sent
Data updated successfully.
(2024-02-01 19:11:02) [INFO    ] Response: 0x562dea226ad0 /api/signup 200 0
(2024-02-01 19:11:15) [INFO    ] Request: 37.170.100.17:55266 0x7ff804000c10 HTTP/1.1 POST /api/2fa
(2024:02:01 19:11:15) saadeliaswd@gmail.com : Data inserted successfully.
(2024:02:01 19:11:15) Data succesfully inserted in user: compareCode not an error
(2024:02:01 19:11:15) saadeliaswd@gmail.com : True 2FA
(2024-02-01 19:11:15) [INFO    ] Response: 0x7ff804000c10 /api/2fa 200 0
(2024-02-01 19:13:48) [INFO    ] Request: 37.170.100.17:55178 0x7ff804002190 HTTP/1.1 POST /api/signin/salt
(2024:02:01 19:13:48) mail : saadeliaswd@gmail.com, Ok salt
(2024-02-01 19:13:48) [INFO    ] Response: 0x7ff804002190 /api/signin/salt 200 0
(2024-02-01 19:13:48) [INFO    ] Request: 37.170.100.17:55179 0x7ff804003660 HTTP/1.1 POST /api/signin
(2024:02:01 19:13:48) saadeliaswd@gmail.com signed in !
(2024-02-01 19:13:48) [INFO    ] Response: 0x7ff804003660 /api/signin 200 0
(2024-02-01 19:13:48) [INFO    ] Request: 37.170.100.17:55180 0x7ff804002190 HTTP/1.1 POST /api/publicinfo
{"betdispo":[{"choice":"Russie","closetime":"2024:02:01 19:24:56","cote":"1.792328","eventid":"1","id":"1"},{"ch
```

Figure 3 : Journal de Logs du serveur

Cette classe s'avère donc extrêmement utile pour le débogage. Sa classe fille est un helper nommé JSONhelper qui sert notamment à effectuer les conversions de string à JSON et inversement. Des opérations fréquentes lors de la manipulation de requêtes HTTP.

La plus grosse partie du backend repose sur les actions que nous effectuons sur la base de données. Elle nous sert à stocker nos utilisateurs, les évènements sportifs, les paris et les joueurs des équipes (nous n'utilisons pas encore cette table, elle est là pour le futur). Sur la figure suivante, un diagramme la représentant :

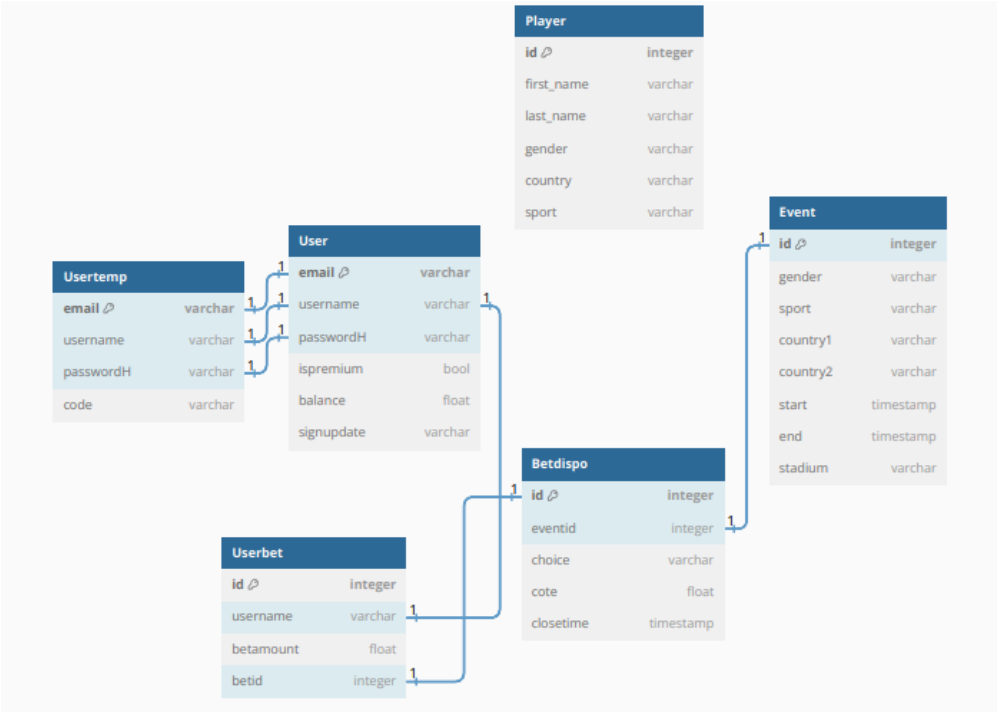


Figure 4 : Diagramme de la base de données

Nous avons choisi d'utiliser la bibliothèque sqlite3 pour faire nos requêtes (SQL) sur celle-ci. Le module ne propose ni le type booléen ni le type timestamp, nous les avons donc remplacés par des tinyint ou des chaînes de caractères. Pour agir sur elle, nous avons défini la classe DatabaseManager qui implémente en méthode toutes les fonctions dont nous avons besoin pour fournir l'application en données ainsi que pour garder en mémoire les différents attributs des utilisateurs. Cette classe possède une fille : Spawner, qui est utile pour générer une base de données comportant des évènements, des paris et de joueurs. Effectivement, nous générons aléatoirement des joueurs et joueuses à l'aide de dictionnaires de noms stockés dans des std::vector. De la même façon, nous générons des rencontres sportives au hasard, sur lesquelles nous proposons des paris (avec des côtes générées uniformément entre 1.1 et 2.1). De plus, nous créons un superuser avec un compte premium et beaucoup de fonds pour pouvoir faire nos tests et ne pas avoir à refaire tout le processus de création de compte à chaque lancement, ce qui peut être fastidieux, surtout avec la vérification à deux facteurs.

Parmi nos plus belles fiertés dans ce projet, il y a la création des comptes. Deux points majeurs nous ont particulièrement plus, ce sont le stockage des mots de passe dans la base de données et la vérification à deux facteurs. Commençons par le premier point. Toujours dans notre tentative de reproduction de la réalité, nous avons vu qu'il était interdit, pour convenir aux normes de sécurité, de stocker des informations en clair. Nous nous sommes juste concentrés sur le stockage du mot de passe. Une méthode assez classique est de stocker un hash (SHA-256) du mot de passe dans la base de donnée lors de la création du compte, ce qui permettra de vérifier en comparant les hash que l'utilisateur soumet bien le bon mot de passe en se connectant. Seulement, il y a un problème avec cette méthode : sa sensibilité aux attaques par dictionnaires. C'est pourquoi nous avons ajouté un sel cryptographique au moment de produire le hash. Le problème de cette opération est qu'elle complexifie légèrement le schéma de requêtes nécessaire pour la connexion. En voici un diagramme :

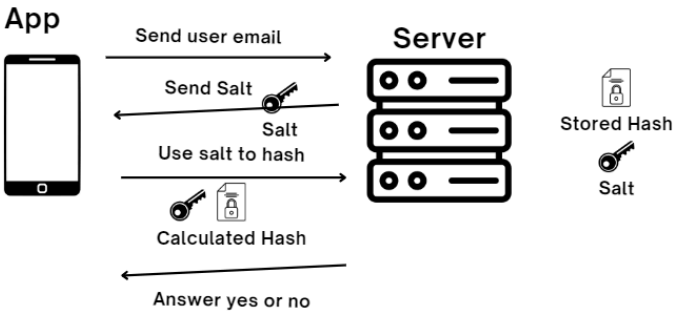


Figure 5 : Schéma de connexion avec hash+sel

Le deuxième point qui nous a tenu à cœur a été l'authentification à deux facteurs lors de la création du compte. Ce procédé consiste à envoyer un mail sur l'adresse que le client fourni pour créer son compte afin de vérifier qu'il s'agit bien de son email. Nous sommes donc assez fiers du rendu qui est plutôt réaliste :

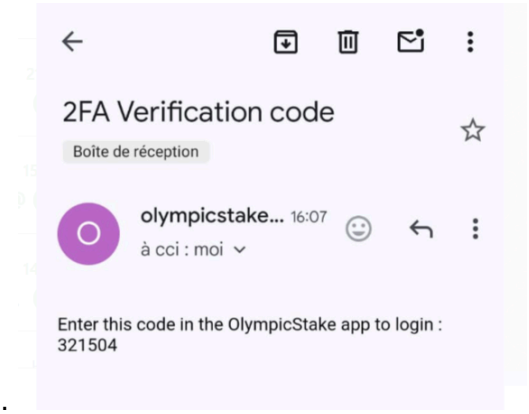
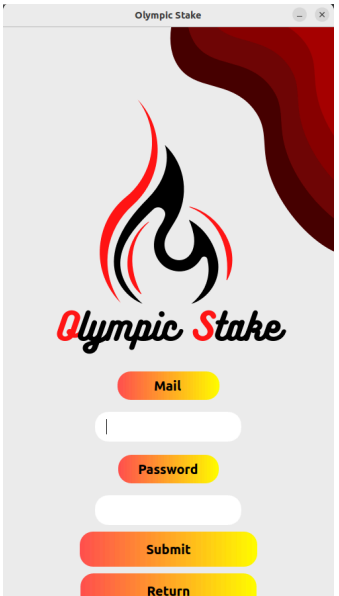


Figure 6 : Mail reçu lors du protocole 2FA

Nous allons maintenant présenter l'interface utilisateur. Après avoir téléchargé l'application et ouvert le serveur (voir les README), l'utilisateur peut l'interface client. Celle-ci a été produite avec Qt.
Voici quelques images de l'interface utilisateur :



Ce que nous voyons ci-dessus, sont les 3 premiers écrans. Le premier à gauche est l'écran d'accueil. Le client peut choisir s'il veut créer un compte ou se connecter. Ensuite, le deuxième montre la page "Sign Up". Ici il est attendu du client un email valide (sur lequel il pourra se connecter pour la 2FA) ainsi qu'un nom d'utilisateur. S'ils ne sont pas uniques (i.e ils sont déjà dans la base de données), l'utilisateur devra les changer. De plus, l'utilisateur rentre un mot de passe qui doit remplir certaines conditions (longueur minimale, chiffres, majuscules, caractères spéciaux). Le message est ensuite hashé sur le frontend pour que le mot de passe ne se retrouve jamais ailleurs que sur le téléphone de l'utilisateur. Celui-ci est ensuite envoyé au serveur qui le stockera dans la base de données temporaire des utilisateurs. Cette base de données sert à stocker les utilisateurs en cours de vérification, i.e ceux qui n'ont pas encore renvoyé leur code de vérification 2FA. Un mail 2FA est donc envoyé au client, il n'a plus qu'à renseigner le bon code pour rentrer dans l'application. La page "Sign-Up", elle, demande un email et un mot de passe.

Nous avons consacré beaucoup de temps à la conception de l'application et tous les éléments présents sur notre interface sont le fruit de notre propre développement. Concernant l'architecture des classes et la hiérarchie, nous avons opté pour une approche pragmatique dans l'utilisation de notre base de données réelle. Nous avons décidé de ne pas créer de classes qui se limiteraient à dupliquer les attributs de la base de données. À la place, nous avons privilégié le développement de classes spécialisées dans la manipulation directe des requêtes et de la base de données elle-même. Cependant, voici un diagramme UML des classes présentes sur le frontend :

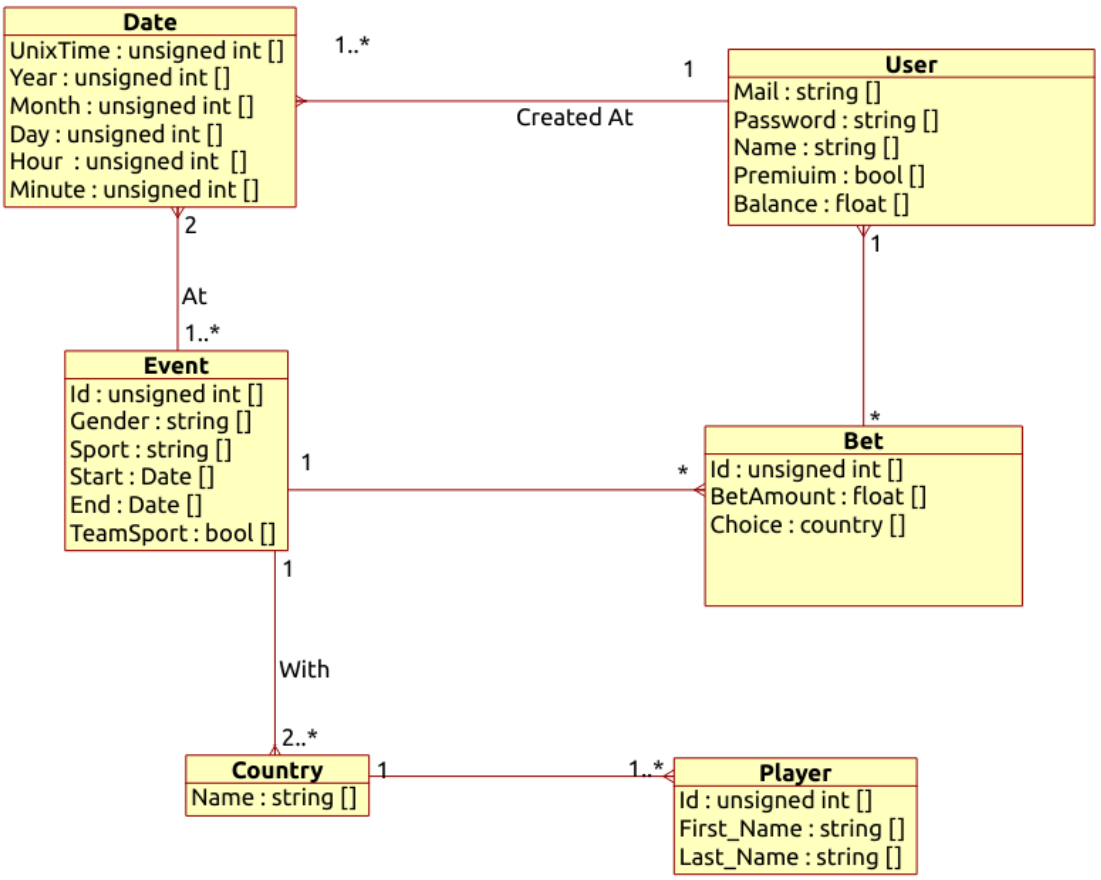


Figure 7 : Diagramme UML des classes du frontend

III] Conclusion :

Nous estimons que ce travail a été très enrichissant de par la diversité des sujets qu'il aborde ; à savoir l'accessibilité d'une application, les réseaux, la sécurité, la cryptographie, les bases de données et le design. Nous avons l'impression d'avoir appris énormément et d'avoir étendu notre culture informatique. Cependant, nous avions mal mesuré l'ampleur du travail et nous n'avons pas pu aller jusqu'au bout de ce que nous voulions faire… La création d'une page de connexion fonctionnelle et sécurisée a quand même été un exercice très intéressant.