CSCE608 – Project #1
SubTrackIt

Justin Lin • 433003638

(a) Project Description

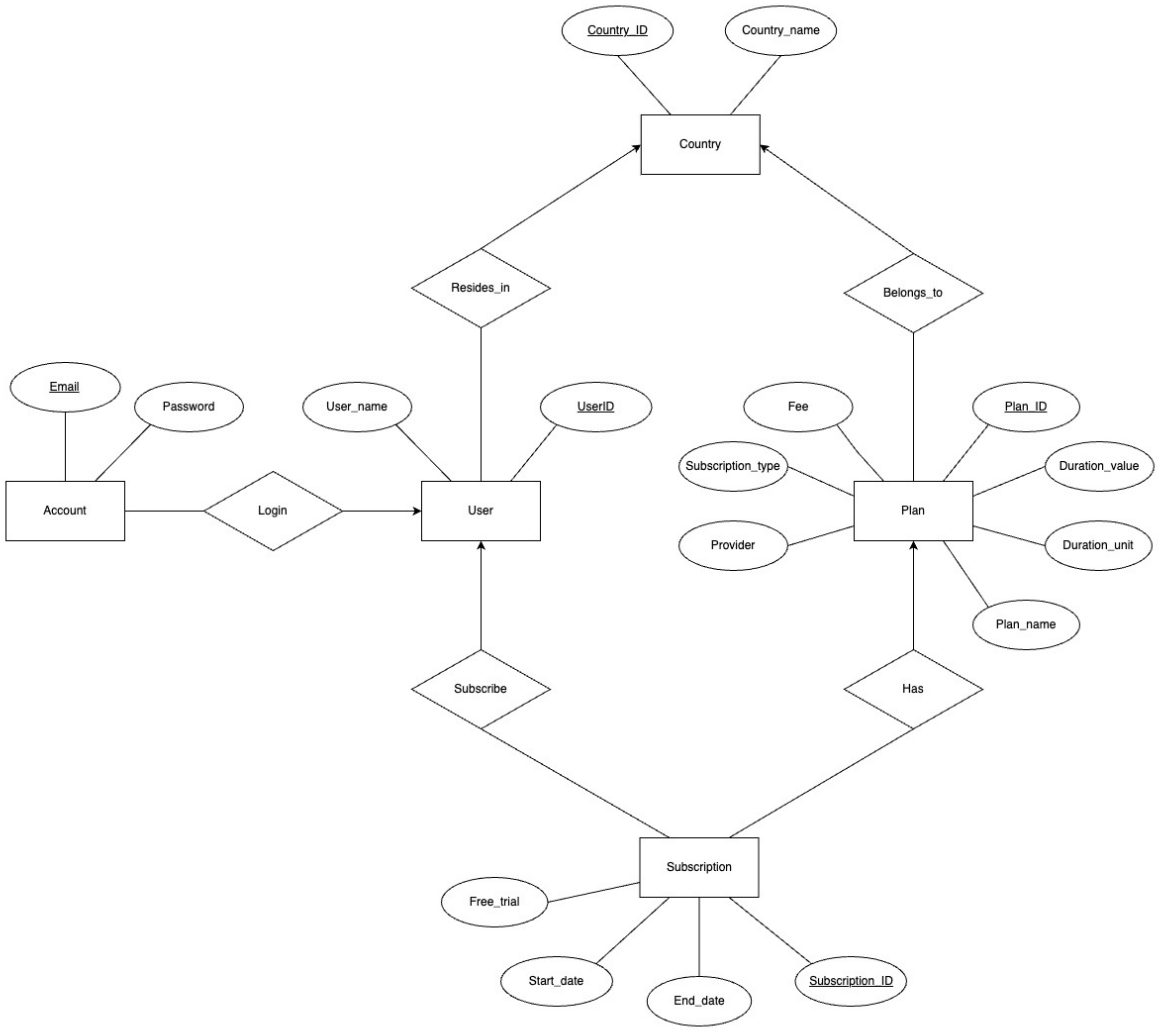
SubTrackIt is a subscription tracker that prevents accidental renewals. The idea for this project stems from hearing friends and family who forgot to cancel subscription services, leading to them paying for things they didn't need. I have also experienced this myself, which motivated me to create a solution to help prevent others from falling into the same situation. For a subscription service, there are the following characteristics: (a) providers who provide subscription plans, (b) subscription plans that allow users to subscribe to, (c) users who subscribe to plans, (d) the details of subscriptions, and (e) accounts for user login to enable data synchronization. SubTrackIt will allow users to add subscription plans to their subscription list, and the app will display renewals that are coming up within a week. Users can also view all their subscription plans in detail and update the subscription plans by resubscribing and unsubscribing. Users can also sync their data with the database by logging in with their email and password. Based on these functionalities, SubTrackIt supports: (a) query subscriptions, (b) add new subscriptions, (c) renew/delete subscriptions, and (d) login/logout/switch accounts (e) close and quit the application when not needed. These 5 functions support the Create, Read Update, and Delete (CRUD) operations, with an additional Quit operation.

For the front end, I believe that it is more logically correct to have an application on your smartphone that you can check anytime, maybe even receive notifications, rather than a website that is hard to access and cannot provide simple notifications. Considering the widespread popularity of iOS devices among the target user demographic, including myself, and my laptop is a MacBook, I decided to develop an iOS app rather than an Android app. The entire iOS app is written in Swift language. The front end will support all the 5 operations by communicating with the server, allowing the user to interact with the database without any database knowledge.

For the server side, a MySQL database is used as required, and a PHP web service is developed to support the database within the XAMPP stack. The reason for choosing PHP web service is mainly because of convenience. XAMPP integrates the PHP web service server into the MySQL database, skipping the hassle of constructing the server. However, there are indeed better alternatives than using PHP web service for this application, for example, Firebase provides a more secure and real-time solution. But for the sake of this project, it will not be deployed to production, and it will remain local, thus, PHP web service shall be sufficient. To establish connection between the app and the server, a REST API is developed using a PHP framework named Slim 4. By defining all the SQL queries on the server side, all the front end has to do is query a specific URL with some headers, and the server will return JSON data to the front end, providing a concise and easy way to send data between the front end and the server.

Additionally, there are triggers that automatically calculate the end date of the subscription based on the start data and the plan chosen. If users had to choose the end date themselves, there would be no purpose in choosing the subscription plan in this app. Through triggers, the application is more user friendly.

(b) The Entity-Relationship diagram of your database



As we discussed in the section above, there are a total of 5 entities. We can list out the attributes for each entity.

- For a **User**, it will have a unique **UserID** (incremented automatically), and a **User_name** (NULL if not logged in or not specified).
- For an **Account**, it will store the user's unique **Email**, and a hashed **Password** (storing the password in plaintext is dangerous).
- For a subscription **Plan**, it will have a unique **Plan_ID** (incremented automatically), the **Plan_name**, the **Provider**, the **Subscription_type** (Standard or Student), the **Fee,** the Duration_value and the Duration_unit.
- For each **Subscription** the user is subscribed to, there is a unique **Subscription_ID** (incremented automatically), **Start_date** (by default today's date, but users have to freedom to alter the date to make up for forgotten occasions), and an **End_date** (calculated automatically based on the Start_date and the subscription plan chosen)

- For a **Country**, there is the **Country_ID** and the **Country_name**. Country is used to decide which subscription plans to display for the user since different countries has different subscription fees and different currency, possibly different subscription plans.

We can also list out the relationships:

- A **User** can **Login** to an **Account**, as a User can have many Accounts, but one Account will only belong to a specific user. Thus, Login is a **one-to-many** relationship.
- A subscription **Plan** can be **Subscribed_for** many different **Durations**. For example, the same plan can be subscribed monthly or yearly, possibly with a reduced fee when subscribed yearly. However, one specific duration will only belong to a specific plan. Thus, Subscribed_for is a **one-to-many** relationship.
- A **User** can **Subscribe** to many different **Subscription**s, but that unique Subscription which includes the Start_date and End_date will only belong to that User. Thus, Subscribe is a **one-to-many** relationship.
- A **Plan Has** many different **Subscription**s, i.e., the same Plan can have many subscribers, but a subscription will only belong to one specific Plan. Thus, Has is a **one-to-many** relationship.
- A **Plan Belongs_to** a **Country**, but a Country can have many Plans. Thus, Belongs_to is a **many-to-one** relationship.
- A **User Resides_in** a **Country**, but a Country can have multiple Users. Thus, Resides_in is a **many-to-one** relationship.

(c) Table normalization

According to the E/R diagram in the section above, we can convert it into relational database schemas (foreign keys are included):

- User(UserID, User_name, Country_ID)
- Account(Email, Password, UserID)
- Plan(Plan_ID, Plan_name, Provider, Fee, Subscription_type, Country_ID)
- Subscription(Subscription_ID, Start_date, End_date, Free_trial, User_ID, Plan_ID)
- Country(Country_ID, Country_name)

We can also list out the functional dependencies.

- UserID → {User_name, Country}
- Email → {Password}
- Plan_ID → {Plan_name, Provider, Fee, Subscription_type, Country}
- Subscription_ID → {Start_date, End_date, Free_trial, User_ID, Plan_ID}
- Country_ID → {Country_name}

A relation R is in Boyce-Codd Normal Form (BCNF) if every nontrivial FD X→A has its left side X a superkey.

- For User, User_ID is the only key. The left sides are all superkeys, User is in BCNF.
- For Account, Email is the only key. The left sides are all superkeys, Account is in BCNF.
- For Plan, Plan_ID is the only key. The left sides are all superkeys, Plan is in BCNF.
- For Subscription, Subscription_ID is the only key. The left sides are all superkeys, Subscription is in BCNF.
- For Country, Country_ID is the only key. The left sides are all superkeys, Country is in BCNF.

By constructing the database with BCNF in mind, the tables are constructed straight forward. There are no superkeys, and only the primary keys decide the other attributes, thus eliminates all non-trivial functional dependencies. The table the table normalization process did not change the table structures at all since the table structure was already in BCNF. This implies that the table structure is also in 3NF.

(d) Data collection

The initial data was collected manually. I went on all the subscriptions services that I could think of and gathered approximately 100 data points. For the sake of the project, in another identical database named “test”, I fabricated 1000 fake data points for the tables Plan and User, and 100 fake data points for the tables Country and Account. This achieves at least two relations with thousands of tuples and at least one relation with hundreds of tuples. To fabricate the fake data points, python scripts were written for each individual table. The fake data points were generated instantly with the help of a library called Faker. The Faker library generates fake data points that make sense, and I believe that this is a better option rather than generating gibberish. In Faker, users can choose the type of data they want to generate. I used attributes such as email, password, company, country and first name. The python scripts can be found in the “RandomDataGenerator” folder.

(e) User interface

The application allows users to view their upcoming renewals within a week on their dashboard page, and their entire list of subscriptions on their subscriptions page. The dashboard page will only show the essential data, including the subscription provider name, the subscription due date, and the day count until subscription is due. On the subscriptions page, users will be provided more details about their subscription, with additional information about the subscription plan name, the subscription type, and the subscription start date.

Fig. 1. Dashboard

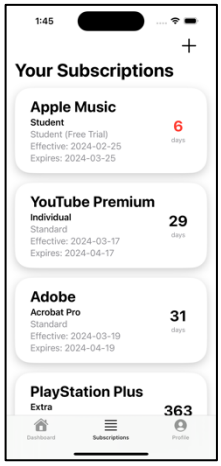


Fig. 2. Subscriptions

To obtain the subscription information, the application will send a GET request to the server, and the server will query the database for the information based on the given User_ID. The two tables Subscription and Plan is joined based on the Plan_ID; the remaining days until the end of subscription is calculated; a threshold is defined to decide what to display according to different scenarios. The entire SQL query is shown below.

```
public function getSubscriptions($id, $threshold) {
    $stmt = $this->conn->prepare("
        SELECT DISTINCT
            p.Provider,
            p.Plan_name,
            p.Subscription_type,
            s.Subscription_ID,
            s.Free_trial,
            s.Start_date,
            s.End_date,
            DATEDIFF(s.End_date, CURDATE()) AS Remaining
        FROM
            Subscription s
            JOIN Plan p ON s.Plan_ID = p.Plan_ID
        WHERE
            s.User_ID = ?
            AND (DATEDIFF(s.End_date, CURDATE()) <= ? OR ? < 0)
        ORDER BY
            Remaining;
    ");
    // Bind the id parameter to the placeholder
    $stmt->bind_param("sii", $id, $threshold, $threshold);
    $stmt -> execute();
    $subscriptions = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
    return $subscriptions;
}
```

Fig. 3. Dashboard and Subscriptions SQL query

If a subscription service has been resubscribed or deleted, users can long press on any of their subscriptions to modify the status.

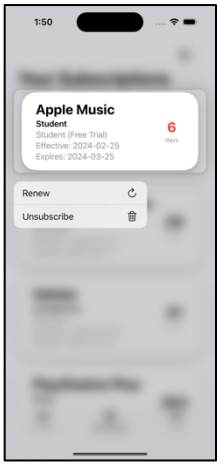


Fig. 4. Actions

For deletion, the application will send a DELETE request to the server, and the server will DELETE the attribute from the database Subscription based on the Subscription_ID. For resubscription, the application will first send a GET request to obtain the end date of the current subscription from the table Subscription based on the Subscription_ID, and then send a PUT request to the server to update the subscription start date to the end date. The end date is updated through a trigger in the database, which automatically calculates the end date according to the subscription length. The Free_trial attribute is changed to False if it was True, since a free trial only occurs once, and once resubscribed, is no longer a free trial.

```php
public function deleteSubscription($subscriptionID) {
    $stmt = $this->conn->prepare("
        DELETE FROM Subscription
        WHERE Subscription_ID = ?
    ");
    $stmt->bind_param("i", $subscriptionID);
    $stmt->execute();

    // Check if the insertion was successful
    if ($stmt->affected_rows > 0) {
        // Return success message or ID of the newly inserted row
        return "Subscription deleted successfully";
    } else {
        // Return error message or handle the case where insertion failed
        return "Failed to delete subscription";
    }
}
```

Fig. 5. Deletion SQL query

```php
public function renewSubscription($subscriptionID) {
    // Fetch End_date and Free_trial
    $stmt = $this->conn->prepare("
        SELECT End_date, Free_trial
        FROM Subscription
        WHERE Subscription_ID = ?
    ");
    $stmt->bind_param("i", $subscriptionID);
    $stmt->execute();
    $result = $stmt->get_result();
    $row = $result->fetch_assoc();
    $endDate = $row['End_date'];
    $freeTrial = $row['Free_trial'];

    // Update Start_date to be the same as End_date
    $stmt = $this->conn->prepare("
        UPDATE Subscription
        SET Start_date = ?
        WHERE Subscription_ID = ?
    ");
    $stmt->bind_param("si", $endDate, $subscriptionID);
    $stmt->execute();

    // Update Free_trial if it is 1
    if ($freeTrial == 1) {
        $stmt = $this->conn->prepare("
            UPDATE Subscription
            SET Free_trial = 0
            WHERE Subscription_ID = ?
        ");
        $stmt->bind_param("i", $subscriptionID);
        $stmt->execute();
    }
```

Fig. 6. Resubscription SQL query

Users can add a new subscription by choosing the provider, the plan they have been subscribed to, and when the subscription started. The plans will dynamically change according to the provider chosen, avoiding confusion achieving convenience. The start date of the subscription is by default the current date, which gives users the freedom to make up if they forgot to do so when they

subscribed. Users can also choose whether this is a free trial or not, which will show up in the details of the subscription as a free trial.

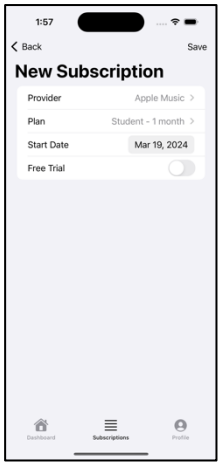


Fig. 7. New subscription

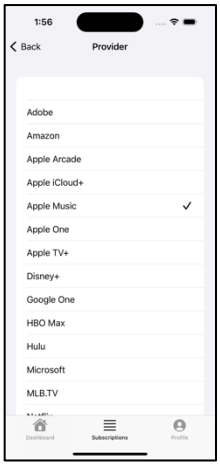


Fig. 8. Providers

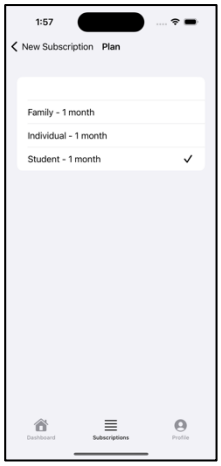


Fig. 9. Plans

The SQL query for this operation is fairly simple, which includes a simple insert operation to the database. The dynamic content is much more complicated and is handled by the front end through querying different attributes.

```php
public function newSubscription($startDate, $freeTrial, $userID, $planID) {
    $stmt = $this->conn->prepare("
        INSERT INTO Subscription (Start_date, Free_trial, User_ID, Plan_ID)
        VALUES (?, ?, ?, ?)
    ");
    $stmt->bind_param("sssi", $startDate, $freeTrial, $userID, $planID);
    $stmt->execute();
```

Fig. 10. New subscription SQL query

By default, user subscription data is saved in the database according to their device's unique identification number. Users have the freedom to choose whether they want to sync their data by registering their email account or not. By syncing their data, even if the application is deleted, or the identification number changes somehow, users can still log back in and use their old data, even if the application has been reinstalled. The sync data mechanism is very simple, users only need to enter their name, email, and password, and the system will automatically check whether this account exists or not. If exists, the system will use that account's data; if does not exist, the system will create a new account for the user. The password is hashed before sent to the database for storage to ensure security.

Fig. 11. Sync data

Fig. 12. Logged in

The SQL query for initialization is fairly simple and includes an INSERT query. There will be a POST request to create a new user, and it fails if the user already exists, checking if the current device identification number exists or not at the same time.

```
public function createUser($userID, $deviceToken) {
    $stmt = $this->conn->prepare("
        INSERT INTO User (User_ID, Device_token)
        VALUES (?, ?)
    ");
    $stmt->bind_param("ss", $userID, $deviceToken);
    $stmt->execute();
```

Fig. 13. User initialization SQL query

The process for syncing the User with their Account is more complicated. If an account exists, we want to set the user ID global variable that is used throughout the entire application to the queried value. For this, the front end sends out a GET request, and the database executes a SELECT query to query the user ID based on the email and password entered. If an account does not exist, we want to create a new account. The front end sends out a POST request, and the database executes an INSERT query to add the account information into the database. Additionally, the username is not mandatory. The user can still create and sync data with their account by only entering their email and password. If the user later decides to enter their username, which will be displayed on the profile page once logged in, they simply enter their username when logging in. The front end sends out a PUT request, and the database executes a UPDATE query to update the username based on the current user ID.

```php
public function syncData($username, $email, $password, $userID, $requestMethod) {
    if ($requestMethod === 'GET') {
            $stmt = $this->conn->prepare("
            SELECT User_ID
            FROM Account
            WHERE Email = ? AND Password = ?
        ");
        $stmt->bind_param("ss", $email, $password);
        $stmt->execute();
        $user_ID = $stmt->get_result()->fetch_all(MYSQLI_ASSOC);
        return $user_ID;

    } elseif ($requestMethod === 'POST') {
        $stmt = $this->conn->prepare("
            INSERT INTO Account (Email, Password, User_ID)
            VALUES (?, ?, ?)
        ");
        $stmt->bind_param("sss", $email, $password, $userID);
        $stmt->execute();

        if ($stmt->affected_rows > 0) {
            // User created successfully, retrieve the newly generated userID
            return "Account created successfully";
        } else {
            // Failed to create user
            return "Failed to create account";
        }
    } elseif ($requestMethod === 'PUT') {
        $stmt = $this->conn->prepare("
            Update User
            SET User_name = ?
            WHERE User_ID = ?
        ");
        $stmt->bind_param("ss", $username, $userID);
        $stmt->execute();
```

Fig. 14. Data sync SQL query

(f) Project source code

The entire source code including the frontend and server is on my GitHub page:
https://github.com/JgtL13/SubTrackIt.git
It will also be uploaded on Canvas. The data points in the database however is not provided.

(g) Discussion

This application was a first of everything for me. I have never extensively used SQL, PHP, nor Swift language before, and this presented me a significant learning curve. However, I view this as an opportunity for me to step outside of my comfort zone and improve my skill set. I have been spending additional time to master the fundamentals of these technologies. The most difficult part was using Slim 4 to develop a RESTful API, Slim 4 does a great job meeting my needs for this project, it's just that connecting the database through a PHP web service with the iOS application was not easy. The iOS application did not see PHP as a secure way of passing data, and it kept blocking me from doing so. After that was taken care of, the wrong configurations of Slim 4 also prevented the server from going online, thus no data could be received by the server. Postman is a very useful tool during this testing process, and it helped me a lot figuring out which side of the system is causing the problem by analyzing JSON data. After the connection of the front end and server side was established, another major problem was the front end design, especially with me not familiar with Swift. Not only was making sure everything is in the right position where I want it to be problematic, behavioral issues also started to emerge with the application not showing expected data. Luckily, nothing was unfixable with time. I was able to find the bugs in the program by spending extra time on it. Other than that, the constructing of the database and the SQL queries were actually fairly simple and straight forward.

There are some features that I have in mind that have not yet been implemented due to time constraint:

- A notification system that notifies the user when a subscription renewal is coming up even when the application is not running.
- Display different plans according to the user's country.
- A web crawler to scrape and update periodically all the subscription services and plans.
- A statistics page that calculates how much money the user has saved by using SubTrackIT.

I believe that these functions, although nice to have, are not essential for the sake of this project. I plan to continuously develop this application, implement the functions listed above, improve the stability of the system, maybe switch to using Firebase, and one day hopefully publish it to the App Store.

This project really honed my SQL technical skills and . In the end, I was able to write a query based on my needs without the help of online resources. I was also more familiar with front end and server interaction, especially with RESTful APIs. I conclusion, this project benefited me a lot, and was also fun to work on since it was a topic I was interested in and was solving a real world problem.