Réaliser la synchronisation des deux bases de données en utilisant des Déclencheurs et des Fonctions :

Les déclencheurs (triggers) :

Sont des mécanismes dans une base de données qui permettent d'exécuter automatiquement une action lorsqu'un événement spécifique se produit sur une table (INSERT, UPDATE, DELETE). Pour synchroniser deux bases de données, on peut utiliser des déclencheurs afin de s'assurer que toute modification effectuée dans une base soit répliquée dans l'autre. Et comme ça l'utilisateur pourra toujours s'authentifier avec le même identifiant et mot de passe sur les différents projet.

- Comment ça fonctionne ?

1. Création des déclencheurs :

On définit des déclencheurs sur les tables de la première base de données (Prevention). Ces déclencheurs surveillent les opérations INSERT, UPDATE et DELETE.

2. Exécution d'une fonction :

Lorsqu'un événement se produit, un déclencheur appelle une fonction stockée qui enregistre ou applique ces modifications dans la deuxième base (Server).

3. Synchronisation totale :

La création des déclencheurs et l'exécution des fonctions se fera dans les deux sens.

I. Synchronisation de la bdd du projet Server d'après le projet Prévention :

1. Création d'un compte utilisateur sur Server après sa création sur Prévention :

- *Trigger :*

```
create trigger after_insert_prevention_users after
insert on
    prevention.users for each row execute function sync_user_to_server()
```

- *Fonction :*

```
create or replace function public.sync_user_to_server()
 returns trigger
 language plpgsql
as $function$
begin


        -- Insérer un nouvel utilisateur dans server.users
    insert into server.users (id, lastname, firstname, email, password)
    values (new.id, new.lastname, new.firstname, new.email, new.password)


        -- si l'utilisateur existe déjà, ignorer l'insertion.
    on conflict (id) do nothing;


    return new;
end;
$function$;
```

2. Modification d'un compte utilisateur sur Server après sa modification sur Prevention :

- Trigger :

```
create trigger after_insert_prevention_users after
insert on
    prevention.users for each row execute function sync_user_to_server()
```

- Fonction :

```
create or replace function public.update_user_in_server()
 returns trigger
 language plpgsql
as $function$
begin


    -- mettre à jour les données dans server.users
    update server.users
    set firstname = new.firstname,
                        lastname = new.lastname,
        email = new.email
    where id = new.id;


    return new;
end;
$function$;
```

3. Suppression d'un compte utilisateur sur Server après sa suppression sur Prevention :

- Trigger :

```
create trigger after_delete_prevention_users after
delete on
prevention.users for each row execute function delete_user_in_server();
```

- Fonction :

```
create or replace function delete_user_in_server()
returns trigger as $$
begin
    -- supprimer l'utilisateur dans server.users
    delete from server.users
    where id = old.id;
    return old;
end;
$$ language plpgsql;
```

II. Synchronisation de la bdd du projet Prevention d’aprés le projet Server :

1. *Création d'un compte utilisateur sur Prevention après sa création sur Server :*

- Trigger :

```
create trigger after_insert_server_users after
insert on
  server.users for each row execute function sync_user_to_prevention();
```

- Fonction :

```
create or replace function sync_user_to_prevention()
returns trigger as $$
begin
        -- insérer un nouvel utilisateur dans prevention.users
   insert into prevention.users (id, username, email)
   values (new.id, new.username, new.email)
         -- si l'utilisateur existe déjà, ignorer l'insertion.
   on conflict (id) do nothing;
   return new;
end;
$$ language plpgsql;
```

2. *Modification d'un compte utilisateur sur Prevention après sa modification sur Server :*

- Trigger :

```
create trigger after_update_server_users after
 update on
server.users for each row execute function update_user_in_prevention();
```

- Fonction :

```
create or replace function update_user_in_prevention()
returns trigger as $$
begin


        -- mettre à jour les données dans prevention.users
    update prevention.users
    set username = new.username,
        email = new.email
    where id = new.id;


    return new;
end;
$$ language plpgsql;
```

3. *Suppression d'un compte utilisateur sur Prevention après sa suppression sur Server :*

- Trigger :

```
create trigger after_delete_server_users after

delete on

server.usersf or each row execute function delete_user_in_prevention();
```

- Fonction :

```
create or replace function delete_user_in_prevention()

returns trigger as $$

begin


    -- supprimer l'utilisateur dans prevention.users

    delete from prevention.users


    where id = old.id;


    return old;

end;

$$ language plpgsql;
```

Centraliser l’authentification de deux projets Laravel avec Keycloak sans interface Keycloak

- Objectif : Permettre aux utilisateurs de se connecter sur les différents applis directement après une seule authentification et sans passer par l'interface Keycloak.

Étape 1 : Installer et Configurer Keycloak

1.1 Lancer Keycloak avec Docker

– Dans un fichier docker-compose.yml :

```yaml
version: '3.8'

services:
  keycloak:
    image: quay.io/keycloak/keycloak:latest
    container_name: keycloak
    command: start-dev
    environment:
      - KEYCLOAK_ADMIN=admin
      - KEYCLOAK_ADMIN_PASSWORD=admin
    ports:
      - "9090:8080"
    volumes:
      - keycloak_data:/opt/keycloak/data

volumes:
  keycloak_data:
```

– Ouvrir un terminal dans le chemin du fichier docker-compose.yml et taper :

`` docker compose up –d ``

– Accès à l’interface Keycloak : http://localhost:9090
– S’Authentifier → [username : admin / mot de passe : admin]

1.2 Créer un Realm

Aller dans Manage → Realm

Créer un Realm → Nom : Lery

1.3 Créer un Client

Aller dans Clients → Créer un Client

- Client ID : laravel
- Client Type : Confidential
- Root URL : http://localhost:8000

Enregistrer, puis aller dans Settings :

- Désactiver Standard Flow
- Activer Direct Access Grants
- Activer Service Accounts

Ajouter les URLs des projets :

- Valid Redirect URIs: http://localhost:8000/* http://localhost:8001/*

1.4 Récupérer le Secret du Client :

- Aller dans l'onglet Credentials du client laravel-backend
- Copier Client Secret (on en aura besoin pour la configuration des projets laravel).

1.5 Créer un Utilisateur Test :

Utilisateurs → Ajouter un utilisateur.

- Nom d'utilisateur : test
- Email : test@example.com

Activer l'utilisateur.

Aller dans Credentials et définir un mot de passe.

- Mot de passe : password
- Activer "Temporary" sur OFF

Étape 2 : Configurer Laravel

2.1 installer Guzzle :

- Dans les deux projets :

  `` composer require guzzlehttp/guzzle ``

2.2 Configurer .env pour Keycloak :

- Dans les deux projets, éditer .env :

KEYCLOAK_BASE_URL=http://localhost:8080

KEYCLOAK_REALM=Lery

KEYCLOAK_CLIENT_ID=laravel

KEYCLOAK_CLIENT_SECRET=TON_SECRET # Remplace par le secret récupéré

2.3 Ajouter keycloak_id à la Table users :

- Dans les deux projets, créer une migration :

  `` php artisan make:migration add_keycloak_id_to_users_table --table=users ``

- Modifier le fichier généré dans database/migrations :

```
public function up() {
        Schema::table('users', function (Blueprint $table) {
                $table->string('keycloak_id')->unique()->nullable();
        });
}
```

- Appliquer la migration :

  `` php artisan migrate ``

2.4 Créer le Service Keycloak :

– Dans les deux projets, créer un fichier app/Services/KeycloakService.php :

```
namespace App\Services;



use GuzzleHttp\Client;
use GuzzleHttp\Exception\RequestException;

class KeycloakService
{
    protected $client;
    protected $baseUrl;
    protected $realm;
    protected $clientId;
    protected $clientSecret;

    public function __construct()
    {
        $this->client = new Client();
        $this->baseUrl = env('KEYCLOAK_BASE_URL');
        $this->realm = env('KEYCLOAK_REALM');
        $this->clientId = env('KEYCLOAK_CLIENT_ID');
        $this->clientSecret = env('KEYCLOAK_CLIENT_SECRET');
    }

    public function authenticate($username, $password)
    {
        try {
            $response = $this->client->post("{$this->baseUrl}/realms/{$this->realm}/protocol/openid-connect/token", [
                'form_params' => [
                    'grant_type' => 'password',
                    'client_id' => $this->clientId,
                    'client_secret' => $this->clientSecret,
                    'username' => $username,
```

```
                'password' => $password,
            ]
        ]);

        return json_decode($response->getBody(), true);

    } catch (RequestException $e) {
        return null;
    }
}

public function getUserInfo($accessToken)
{
    try {
        $response = $this->client->get("{$this->baseUrl}/realms/{$this->realm}/protocol/openid-connect/userinfo", [
            'headers' => [
                'Authorization' => 'Bearer ' . $accessToken,
            ]
        ]);

        return json_decode($response->getBody(), true);
    } catch (RequestException $e) {
        return null;
    }
}
}
```

2.5 Modifier AuthController.php :

– Créer app/Http/Controllers/AuthController.php dans les deux projets :

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use App\Services\KeycloakService;
use App\Models\User;

class AuthController extends Controller
{
    protected $keycloak;

    public function __construct(KeycloakService $keycloak)
    {
        $this->keycloak = $keycloak;
    }

    public function showLoginForm()
    {
        return view('auth.login');
    }

    public function login(Request $request)
    {
        $credentials = $request->validate([
            'email' => 'required|email',
            'password' => 'required'
        ]);

        $response = $this->keycloak->authenticate($credentials['email'], $credentials['password']);
```

```
        if ($response && isset($response['access_token'])) {
            $userInfo = $this->keycloak->getUserInfo($response['access_token']);

            if ($userInfo) {
                $user = User::firstOrCreate(
                    ['keycloak_id' => $userInfo['sub']],
                    ['name' => $userInfo['name'], 'email' => $userInfo['email']]
                );

                Auth::login($user);

                return redirect('/home');
            }
        }

        return back()->withErrors(['email' => 'Identifiants incorrects']);
    }

    public function logout()
    {
        Auth::logout();
        return redirect('/');
    }
}
```

2.6 Ajouter les Routes :

- Dans routes/web.php :

```
use App\Http\Controllers\AuthController;



Route::get('/login', [AuthController::class, 'showLoginForm'])->name('login');

Route::post('/login', [AuthController::class, 'login']);

Route::get('/logout', [AuthController::class, 'logout'])->name('logout');



Route::get('/home', function () {

    return view('home');

})->middleware('auth');
```

Étape 3 : Tester

- Lancer Keycloak
- Lancer les deux projets Laravel :

`` php artisan serve --port=8000 `` # Projet 1

`` php artisan serve --port=8001 `` # Projet 2

- Se connecter sur http://localhost:8000/login
- Vérifier que la session fonctionne sur http://localhost:8001/home