issues
BankArg - APLICACIÓN

Implementar algún tipo de persistencia de datos ya sea para almacenar información de procesos o para servir de sistema de autenticación mediante el almacenamiento de cuentas de usuario/contraseñas.

Para BankArg, una aplicación bancaria que maneja información sensible como cuentas de usuario y transacciones financieras, es fundamental utilizar una implementación segura de persistencia de datos.

Se presentan dos enfoques comunes para la persistencia de datos en una aplicación móvil con requisitos de seguridad, *como el almacenamiento de cuentas de usuario y contraseñas, así como registros de transacciones:*

1. SharedPreferences (para datos sencillos):

SharedPreferences es útil para almacenar datos simples como configuraciones de la aplicación o tokens de usuario. Sin embargo, para contraseñas y datos financieros, no se recomienda su uso, ya que no proporciona un alto nivel de seguridad. Aún así, puede ser útil para guardar información temporal como tokens de autenticación.

Ejemplo de uso de SharedPreferences para almacenar un token de autenticación:

Java

```
// Guardar un token de autenticación en SharedPreferences
SharedPreferences preferences = getSharedPreferences("BankArgPrefs", Context.MODE_PRIVATE);
SharedPreferences.Editor editor = preferences.edit();
editor.putString("AuthToken", "token_de_autenticacion_generado");
editor.apply();

// Recuperar el token de autenticación
String authToken = preferences.getString("AuthToken", "");
```

---

2. SQLite (para datos estructurados y seguros):

Para información crítica como cuentas de usuario y registros de transacciones financieras, se recomienda utilizar SQLite o una base de datos segura similar. SQLite proporciona una estructura de base de datos relacional que es adecuada para almacenar y consultar datos de manera segura.

Ejemplo de uso de SQLite para almacenar cuentas de usuario:

Java

Copiar código

```
// Crear o abrir la base de datos SQLite
SQLiteDatabase db = openOrCreateDatabase("BankArgDB", Context.MODE_PRIVATE, null);

// Crear la tabla de cuentas de usuario si no existe
db.execSQL("CREATE TABLE IF NOT EXISTS CuentasUsuario (id INTEGER PRIMARY KEY AUTOINCREMENT, usuario TEXT, contrasena TEXT)");

// Insertar una nueva cuenta de usuario
String usuario = "nombre_de_usuario";
String contrasena = "contrasena_segura_hash";
db.execSQL("INSERT INTO CuentasUsuario (usuario, contrasena) VALUES (?, ?)", new Object[]{usuario, contrasena});

// Recuperar una cuenta de usuario por nombre de usuario
```

```
Cursor cursor = db.rawQuery("SELECT * FROM CuentasUsuario WHERE usuario = ?",
new String[]{usuario});
if (cursor != null && cursor.moveToFirst()) {
   String storedPassword = cursor.getString(cursor.getColumnIndex("contrasena"));
// Comparar la contraseña almacenada con la contraseña proporcionada durante el
inicio de sesión
}
```

Es fundamental utilizar técnicas de encriptación y hash para proteger las contraseñas almacenadas en la base de datos SQLite y garantizar la seguridad de la información de los usuarios.

Ejemplos de cómo se puede mejorar la seguridad al trabajar con contraseñas:

Hashing de contraseñas: En lugar de almacenar las contraseñas de los usuarios en texto claro, se deben almacenar los valores hash de las contraseñas. *Un hash es una representación irreversible de una cadena de caracteres*. Cuando un usuario se registra o cambia su contraseña, la aplicación debe calcular el hash de la contraseña y almacenarlo en la base de datos en lugar de la contraseña real. Esto hace que sea extremadamente difícil recuperar la contraseña original a partir del hash almacenado.

Java

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public String hashPassword(String password) throws
NoSuchAlgorithmException {
   MessageDigest md = MessageDigest.getInstance("SHA-256");
```

```
    byte[] hashedBytes = md.digest(password.getBytes());
    StringBuilder sb = new StringBuilder();
    for (byte b : hashedBytes) {
        sb.append(String.format("%02x", b));
    }
    return sb.toString();
}
```

Salt (Sal): Para hacer que los hashes sean aún más seguros, se debe utilizar *un valor aleatorio único para cada usuario, conocido como "sal".* La sal se concatena con la contraseña del usuario antes de aplicar la función hash. Esto garantiza que incluso si dos usuarios tienen la misma contraseña, sus hashes serán diferentes debido a las ventas únicas.

```
import java.security.SecureRandom;
import java.util.Base64;

public String generateSalt() {
    SecureRandom random = new SecureRandom();
    byte[] salt = new byte[16];
    random.nextBytes(salt);
    return Base64.getEncoder().encodeToString(salt);
}
```

Almacenamiento seguro de sal y hash: Tanto la sal como el hash deben almacenarse en la base de datos para cada usuario. ***La sal se debe guardar junto con el hash en un campo separado o en una estructura de datos que permita su recuperación cuando sea necesario*** para verificar contraseñas durante el inicio de sesión.

```
// Almacenar la sal y el hash en la base de datos
String salt = generateSalt();
```

```
String hashedPassword = hashPassword(userInputPassword + salt);
```

Verificación de contraseñas: Durante el proceso de inicio de sesión, la aplicación debe recuperar la sal correspondiente del usuario y calcular el hash de la contraseña proporcionada por el usuario con esa sal. Luego, compare el hash calculado con el hash almacenado en la base de datos.

```
// Verificar la contraseña durante el inicio de sesión
String saltFromDatabase = getSaltForUser(username);
String hashedPasswordFromDatabase = getHashedPasswordForUser(username);
String hashedInputPassword = hashPassword(userInputPassword + saltFromDatabase);

if (hashedInputPassword.equals(hashedPasswordFromDatabase)) {
    // Contraseña válida, permitir el acceso
} else {
    // Contraseña incorrecta, denegar el acceso
}
```

En resumen, para una aplicación bancaria como BankArg, es esencial utilizar técnicas de seguridad robustas al almacenar y verificar las contraseñas de los usuarios para proteger la información confidencial y garantizar la integridad de los datos de la aplicación. **El hash de contraseñas con venta es una práctica común y efectiva en la industria para lograr este objetivo.**