**1) HashPassword(string password) — qué hace y por qué**

**Qué hace (resumen):**

* Recibe la contraseña en texto plano.
* Calcula un hash SHA-256 de esa contraseña.
* Devuelve el hash codificado en Base64.

**Paso a paso:**

1. using (SHA256 sha256 = SHA256.Create())
   * Crea un objeto que implementa SHA-256.
2. byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(password));
   * Convierte la contraseña a bytes UTF-8 y calcula el resumen (hash) SHA-256.
3. return Convert.ToBase64String(bytes);
   * Devuelve el hash como string Base64 para almacenar en BD.

**Por qué se hace:**

* Hashear evita almacenar contraseñas en texto plano. Si alguien accede a la BD no tendrá la contraseña literal.

**Limitación / Riesgo:**

* **SHA-256 solo es un hash rápido**: vulnerable a ataques de diccionario y GPU/brute force si no se usa salt y un algoritmo de derivación lento (PBKDF2, bcrypt, Argon2).
* **No hay salt**: dos usuarios con la misma contraseña tendrán el mismo hash.
* **No hay versión ni parámetros** (imposible actualizar esquema fácilmente).

**2) Login(string usuario, string contrasena, out Usuario user) — qué hace y por qué**

**Qué hace (resumen):**

* Intenta leer el usuario por nombre desde la BD.
* Si existe, calcula el hash de la contraseña ingresada con HashPassword.
* Compara el hash calculado con el hash guardado en BD.
* Si coinciden, rellena user con los datos (Id, NombreUsuario, Rol) y devuelve true; si no, devuelve false.

**Paso a paso:**

1. user = null; — inicializa salida.
2. AccesoDatos datos = new AccesoDatos(); — instancia helper de acceso a BD.
3. datos.setearConsulta("SELECT Id, NombreUsuario, ContrasenaHash, Rol FROM Usuarios WHERE NombreUsuario = @user");
   * Prepara SQL con parámetro @user.
4. datos.setearParametros("@user", usuario);
   * Asigna el valor; **esto evita SQL injection**.
5. datos.ejecutarLectura(); — ejecuta la consulta y posiciona Lector.
6. if (datos.Lector.Read()) { ... } — si hay fila:
   * string hashGuardado = (string)datos.Lector["ContrasenaHash"]; — lee hash BD.
   * string hashIngresado = HashPassword(contrasena); — calcula hash de la contraseña suministrada.
   * if (hashGuardado == hashIngresado) { /\* autentica y rellena user \*/ }
     + Si igual, crea Usuario con Id, NombreUsuario, Rol y retorna true.
7. Si no se encontraron coincidencias o la comparación falla, retorna false.
8. finally { datos.cerrarConexion(); } — siempre cierra conexión.

**Por qué se hace así:**

* Se compara hashes en lugar de texto plano.
* Uso de parámetros evita inyección SQL.
* out Usuario user permite devolver datos del usuario autentificado para guardarlos en sesión.

**Limitaciones / Riesgos:**

* **Comparación simple de strings**: suceptible a *timing attacks* (pequeño riesgo práctico aquí).
* **No hay control de intentos** (lockout), no hay registro de fallos.
* **No hay salt**: facilita ataques con tablas rainbow / GPUs.
* **No hay protección adicional** (2FA, cuenta bloqueada, verificación por email).

**3) Registrar(string usuario, string contrasena, string rol) — qué hace y por qué**

**Qué hace (resumen):**

* Calcula el hash de la contraseña.
* Inserta un nuevo registro en Usuarios con NombreUsuario, ContrasenaHash, Rol.

**Paso a paso:**

1. string hash = HashPassword(contrasena); — obtiene hash Base64.
2. AccesoDatos datos = new AccesoDatos(); — helper BD.
3. datos.setearConsulta("INSERT INTO Usuarios (NombreUsuario, ContrasenaHash, Rol) VALUES (@u, @c, @r)");
   * Inserción parametrizada.
4. datos.setearParametros("@u", usuario); etc.
5. datos.ejecutarAccion(); — ejecuta el INSERT.
6. finally { datos.cerrarConexion(); }

**Por qué se hace así:**

* Guarda solo el hash (no contraseña).
* Uso de parámetros evita inyección SQL.

**Limitaciones / Riesgos:**

* **No comprueba si NombreUsuario ya existe** → puede fallar por unique constraint o insertar duplicados si la columna no es unique.
* **Sin salt**: vulnerabilidad ya comentada.
* **Sin validación de complejidad / fuerza** de la contraseña.
* **No maneja errores específicos** (por ejemplo, usuario existente).

**Recomendaciones prácticas e inmediatas (para mejorar seguridad sin reescribir todo)**

1. **Usá PBKDF2 (Rfc2898DeriveBytes) o bcrypt/Argon2** en lugar de SHA-256 simple.
   * PBKDF2 + salt + muchas iteraciones: mucho más resistente a ataques por fuerza bruta.
2. **Generá y guardá un salt por usuario**. Opciones de almacenamiento:
   * Columnas separadas: PasswordSalt, PasswordHash, o
   * Cadena combinada: salt:hash (Base64).
3. **Verificación**: al autenticar, lee salt y hash, recalcula PBKDF2(password, salt) y compara hashes con comparación segura (constant time).
4. **Comprobá existencia antes de insertar** en Registrar:
   * SELECT COUNT(\*) FROM Usuarios WHERE NombreUsuario = @u y lanzar error ameno si ya existe.
5. **Agregá políticas**: longitud mínima, mayúsculas/números/símbolos, etc.
6. **Protecciones operativas**:
   * Limitar intentos fallidos por IP/usuario y bloquear temporalmente (account lockout).
   * Forzar HTTPS siempre.
   * Cookies de sesión HttpOnly, Secure, SameSite=Strict/Lax.
   * Registrar intentos y alertas de seguridad.
7. **2FA**: en e-commerce recomienda 2FA (OTP/Email/SMS) para cuentas sensibles.
8. **Migración a ASP.NET Identity**: si el proyecto crece, migrá a Identity — te da hashing seguro, salting, gestión de roles, lockout, tokens, 2FA, etc.