Capítulo 3

Procedimientos y descripción de las actividades realizadas

Contiene los procedimientos y actividades genéricas llevadas a cabo en los proyectos desarrollados por la Unidad Administrativa responsable así como los lineamientos y estándares de programación.

3. PROCEDIMIENTOS Y DESCRIPCIÓN DE LAS ACTIVIDADES REALIZADAS

3.1. DISEÑO ARQUITECTÓNICO

El *diseño arquitectónico* corresponde al proceso de diseño que identifica los subsistemas que conforman un sistema y la infraestructura de control y comunicación. Un *subsistema* es un sistema en sí mismo cuya operación es independiente de los servicios provistos por otros subsistemas. Un *módulo* es un componente del sistema que provee servicios a otros componentes pero no se consideraría normalmente como un sistema separado.

La estructuración del sistema consiste en descomponer el sistema en varios subsistemas principales e identificar la comunicación entre ellos. Se define el modelo de comunicación entre las partes del sistema. Se descomponen los subsistemas en módulos.

Por estándares de desarrollo dentro del Servicio de Protección Federal, la programación de cualquier producto debe basarse en la programación orientada a objetos y seguir una arquitectura en capas. Estas capas originalmente se definen en: *Datos*, *Negocio* y *Presentación* [3]. Adicionalmente se proponen las capas de *Entidades* y *Seguridad*, como a continuación se describen:

Imagen 3-1. Modelo de programación por capas.

CAPA DE DATOS

Encargada de la conexión a la Base de Datos y de las operaciones de inserción, actualización, eliminación y consulta mediante llamadas a procedimientos almacenados. Provee a las capas superiores toda la información necesaria para su funcionamiento.

CAPA DE NEGOCIO

Encargada de hacer valer las reglas de negocio solicitadas por el cliente; cuando se garantiza que la información que recibe es válida es enviada a la capa de datos para su almacenamiento.

CAPA DE PRESENTACIÓN

Aplicación Web encargada de interactuar con el usuario en la captura y presentación de la información. Los datos recibidos son enviados a la capa de negocio para su validación.

CAPA DE ENTIDADES

En esta capa se crean todas las clases que representan los objetos que serán utilizados por las capas de presentación, negocio y datos. Es una capa transversal que puede ser utilizada por todos las clases del proyecto. La idea central es crear un objeto de entidad en la capa de presentación donde se recogen los datos introducidos por el usuario.

Posteriormente, dicho objeto es enviado a la capa de negocio para su validación. Finalmente, cuando la información es válida, el objeto es enviado a la capa de datos para su procesamiento final. Una vez procesada la información, se devuelve un mensaje de confirmación o de error en un formato legible al usuario. Los mensajes son visualizados en la capa de presentación.

CAPA DE SEGURIDAD

Encargada de crear y mantener las sesiones activas en el sistema, llevar el control y la bitácora de accesos, crear y manipular perfiles de acceso y evitar la intromisión al sistema por personas no autorizadas.

Su funcionamiento se basa en objetos específicos de la capa de entidades así como de métodos en la clase de negocio que validen las reglas de autenticación de usuarios y el manejo de sesiones utilizando métodos de la capa de datos.

La capa de seguridad debe garantizar entre otras cosas:

- El manejo de sesiones de tiempo finito.
- Autenticación segura de usuarios, es decir, un mismo usuario no puede iniciar más de una sesión al mismo tiempo.
- Encriptación de cadenas de conexión a Base de Datos.
- Administración de perfiles.
- Administración de usuarios.

IMPLEMENTACIÓN

La programación ya sea en capas o mediante otra arquitectura de programación requiere de elementos que permitan mejorar la programación y el diseño de sistemas de información aunado a la reutilización de código. El acoplamiento mide el grado de interdependencia entre las unidades de software, es decir, el grado en que una unidad puede funcionar sin recurrir a otras. La cohesión hace referencia a la forma en que se agrupan las unidades de software en una unidad mayor.

Para lograr un bajo acoplamiento de tipo normal, es decir, que una unidad de software solo invoca a otra de un nivel inferior y tan solo intercambian parámetros de entrada salida, que dentro del esquema mostrado, serian instancias de la capa de *entidades*, se optó por manejar librerías de clases encapsuladas en archivos *dll* (*dynamic-link library*, biblioteca de enlace dinámico).

Los *archivos dll* son archivos con código ejecutable que se incluyen e invocan bajo demanda de un programa o sistema operativo. Ofrecen entre otras cosas, reducir el tamaño de los archivos ejecutables, ser compartidas entre varias aplicaciones, facilitan la gestión y aprovechamiento de la memoria y brindan flexibilidad ante cambios [4].

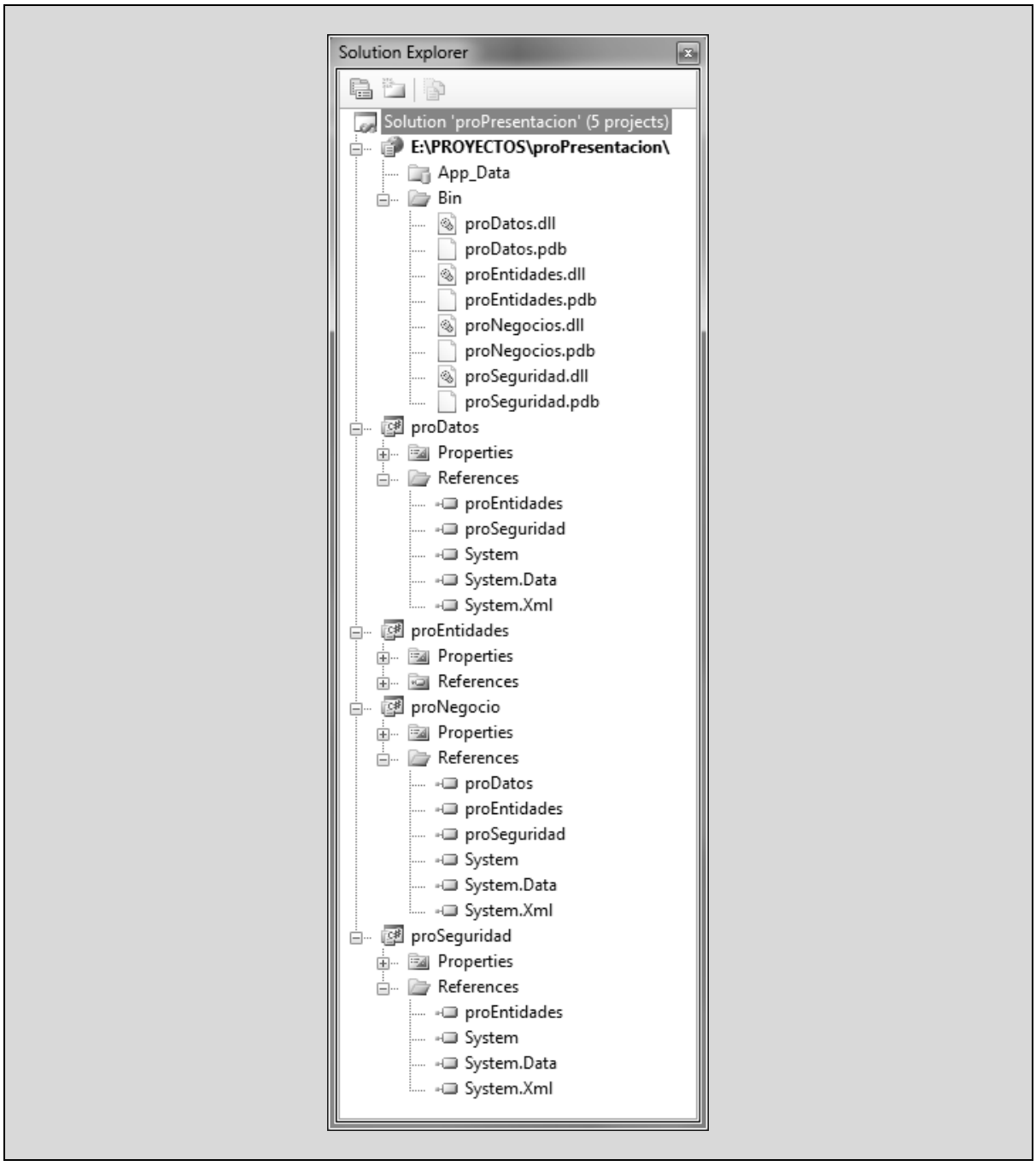


Imagen 3-2. Implementación de un modelo de programación en capas.

El uso de archivos dll permite tener un bajo acoplamiento entre las capas de programación, mejorando su mantenimiento y reutilización. Si se detectan defectos en los archivos dll únicamente se reemplaza el archivo dañado además de que es más sencillo detectar en donde se encuentra el error. Los cambios hechos a cada unidad tienen un impacto menor en las capas a las cuales da soporte.

En Visual Studio encontramos un tipo de proyecto que nos permiten generar estas librerías de clases, compilar su código fuente en conjunto con toda la solución e integrarla a las capas que hagan uso de ella. Gracias a la inclusión de un depurador de código es posible seguir el flujo de información entre las capas, facilitando la detección de fallas.

Para garantizar un bajo acoplamiento en las capas de datos, negocio, entidades y seguridad, estas fueron implementadas como librerías de clases.

La capa de presentación esta implementada como un sitio Web, ya que este tipo de proyecto nos permite montarlo en un ambiente vía Web, y hacer uso de las capacidades de ASP.NET. Así mismo, se observa otra ventaja de manejar librerías de clases, ya que únicamente se requieren los archivos dll para soportar las reglas de negocio, de datos y de seguridad organizada como archivos independientes, así que el mantenimiento de cada una se vuelve más sencillo pues no se requiere la compilación de todo el sitio.

Para tener una alta cohesión, además de la agrupación por capas, dentro de cada una se hace una separación entre aquellos métodos relacionados con alguna entidad específica, con esto se logra que trabajen en un mismo fin. Por ejemplo, se tiene la entidad *empleado*, y todos aquellos métodos relacionados con la capa de negocio serán encapsulados dentro de una clase llamada *clsNegEmpleado*, lo que nos ayuda a identificar que en esta instancia se manejan todas las reglas de negocio relacionadas con el empleado. Así mismo, en la capa de datos se tendrá algo similar, *clsDatEmpleado*, en donde se agrupan los métodos relacionados con la persistencia de empleados [5].

Adicional a la separación física en archivos dll, se hace uso de los *namespaces* o nombres de espacios de .NET, que se utilizan para declarar ámbitos que permiten organizar el código de forma lógica con el propósito de identificarlo como miembro de una capa y hacer uso de ella en ambiente de programación.

La comunicación entre las diferentes capas se hace a través del paso de parámetros, sin embargo, para invocar cada uno de los métodos de la capa inferior estos son creados con la palabra reservada *static* o métodos estáticos, lo que permite que sean utilizados sin crear una instancia de la clase a la que pertenecen, únicamente se hace referencia a la clase, el operador punto y el método. Con esta funcionalidad se pretende disminuir la escritura de código fuente dentro de los módulos donde se haga uso de estos métodos. Además, un método de instancia es aplicado a una instancia como tal y un método estático puede ser aplicado para realizar solo una validación de información o de una instancia no necesariamente a la que pertenece.

3.2. MAPEO OBJETO-RELACIONAL

El concepto de mapeo objeto relacional será aplicado en la capa de entidades; este mapeo es independiente del manejador de base de datos que se utilice o del número de tablas que compongan nuestra base de datos. Un requisito importante es normalizar nuestro modelo entidad relación, ya que además de que se evitan problemas de integridad de datos, el modelado es más sencillo de llevar a cabo así como posibles actualizaciones al mismo.

Para mapear una tabla en su correspondiente clase, identificamos la tabla así como sus atributos y el tipo de dato que le fue asignado así como las relaciones con otras tablas.

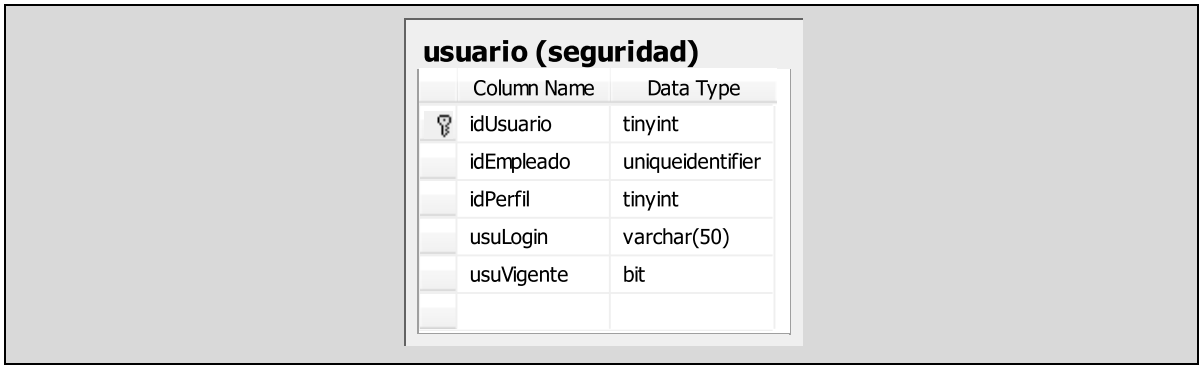


Imagen 3-3. Entidad que representa a un usuario del sistema.

Una vez identificados estos elementos, en la capa de entidades creamos una clase de acceso público a la que nombramos *clsEntNombreTabla*, donde *NombreTabla* corresponde al nombre de la tabla. Se agregan atributos de clase con acceso privado, el tipo de dato y la nomenclatura *_nombreAtributo*. Repetimos este paso para cada uno de los atributos. Finalmente, se generan las propiedades (*get*, *set*) de acceso público.

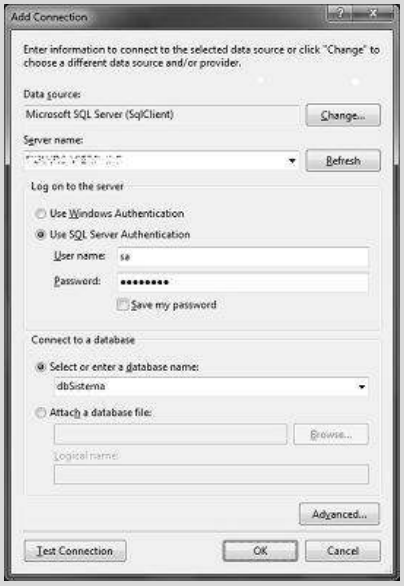
```
public class clsEntUsuario
{
    private short _idUsuario;

    public short IdUsuario
    {
        get { return  idUsuario; }
        set { _idUsuario = value; }
    }
    private Guid _idEmpleado;

    public Guid IdEmpleado
    {
        get { return _idEmpleado; }
        set { _idEmpleado = value; }
    }
    private short _idPerfil;

    public short IdPerfil
    {
        get { return _idPerfil; }
        set { _idPerfil = value; }
    }
    private string _usuLogin;

    public string UsuLogin
    {
        get { return  usuLogin; }
        set { _usuLogin = value; }
    }
    private bool _usuVigente;

    public bool UsuVigente
    {
        get { return _usuVigente; }
        set { _usuVigente = value; }
    }
}
```

Imagen 3-4. Clase que mapea la entidad Usuario.

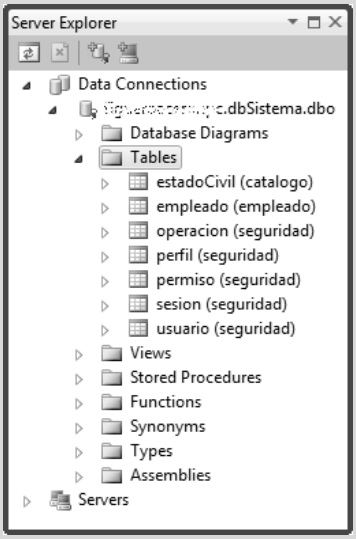
Una vez finalizado el mapeo de tablas, se pueden realizar ajustes como incluir referencias a otras entidades como atributos de otra, con la finalidad de crear una relación entre ambas. Así por ejemplo, en la entidad *clsEntUsuario*, se hace mención del ID de un perfil, el cual podemos sustituir por un atributo de la entidad *clsEntPerfil*, gracias a lo cual incluiríamos los atributos de esa clase pero sin una relación de herencia únicamente de asociación.

Este proceso resultó largo y repetitivo por lo que se optó por utilizar la herramienta *LINQ to SQL* para simplificar el proceso de mapeo [7]. Mediante este método se siguen los siguientes pasos:

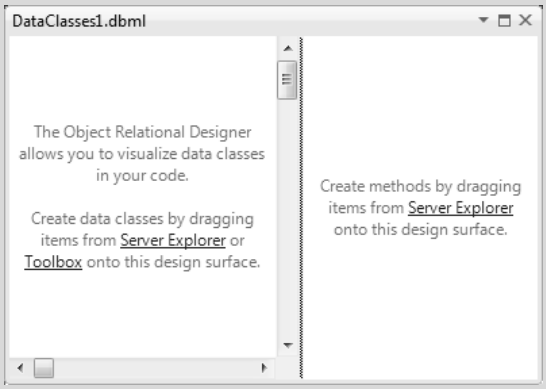
(1) Establecer una conexión de datos con el servidor de base de datos.
(2) Seleccionar la base de datos que nos interesa y desplegar la carpeta de Tablas.
(3) Agregar un elemento de tipo *LINQ to SQL Classes* el cual se divide en dos pantallas: Diseñador Objeto Relacional y Diseñador de Métodos.
(4) Arrastramos las tablas que deseamos mapear hacia el Diseñador Objeto Relacional y se crean los diagramas de clases correspondientes y guardamos los cambios.



(1) Establecer una conexión de datos con el servidor de base de datos.



(2) Seleccionar la base de datos que nos interesa y desplegar la carpeta de Tablas.



(3) Agregar un elemento de tipo *LINQ to SQL Classes* el cual se divide en dos pantallas: Diseñador Objeto Relacional y Diseñador de Métodos.

(4) Arrastramos las tablas que deseamos mapear hacia el Diseñador Objeto Relacional y se crean los diagramas de clases correspondientes y guardamos los cambios.

Imagen 3-5. Proceso para mapear entidades de una Base de Datos en LINQ.

Mediante este procedimiento, podemos actualizar la capa de entidades de una forma rápida y sencilla, podemos actualizar nuestro modelo de datos, eliminar la entidad y nuevamente arrastrarla para aplicar los cambios.

Una vez creada la capa de entidades, se hace uso de ella, agregando la palabra reservada *using* seguida del *namespace* de entidades; una vez indicado, en la porción de código donde se requiere se crea la instancia correspondiente y se asignan las propiedades que se requieran. Es importante mencionar que la nomenclatura de las entidades no corresponde a la descrita con anterioridad, esto se debe a que *LINQ* nombra las entidades con el mismo nombre de la tabla, sin incluir su esquema.

```
using proEntidades;

.
.
.

usuario objUsuario = new usuario();
objUsuario.usuLogin = "usuario";
```

Imagen 3-6. Uso de las clases de LINQ.

3.3. CONEXIÓN A BASE DE DATOS

La comunicación que se establece con el servidor de base de datos debe entre otros aspectos realizarse con una autenticación de usuario, mediante un identificador y una contraseña. Normalmente esta información permanece fija en los archivos de configuración de los sistemas, junto con la dirección IP del servidor, escritas en texto plano [6].

Adicional a esto, las aplicaciones se desarrollar para trabajar con un manejador de base de datos específico, por ejemplo: *SQL Server*, sin detenerse a pensar que posiblemente se requiera en un determinado momento migrar a otro manejador; por esta razón, se optó por utilizar librerías de comunicación a un servidor de base de datos genérico, y agregando en el archivo de configuración una clausula donde de forma sencilla se puede indicar el proveedor de base de datos. Con este mecanismo, la migración a otros manejadores resulta ser más sencilla, únicamente con cambios menores en sintaxis de SQL.

La idea central de estas implementaciones es restringir el acceso a los sistemas de información únicamente a aquellos usuarios que cuenten con un nombre de usuario y una contraseña validas en el servidor de base de datos. Luego entonces, al momento de la autenticación del usuario, se recoge la información necesaria para validar sus credenciales en el servidor. Este mecanismo provee seguridad de la información y permite tener una bitácora de accesos a la información, manejado por el servidor y administrado por el personal correspondiente.

La lógica empleada en este mecanismo consiste en cifrar parte de la cadena de conexión y el tipo de proveedor de datos utilizando el algoritmo *Rijndael* y colocándolos en el archivo de configuración, después en una clase denominada *clsDatConexion* se implementa un método que extrae estas dos variables en texto plano y concatena el nombre de usuario y contraseña de la persona que intenta autenticarse.

```
Public DbConnection getConexion(clsEntSesion objSesion)
{
    string strConnectionString =
clsSegRijndaelSimple.Decrypt(System.Configuration.ConfigurationManager.AppSettings["base"]);

    strConnectionString += "UID=" + objSesion.usuario.UsuLogin + ";PWD=" +
clsSegRijndaelSimple.Decrypt(objSesion.usuario.UsuContrasenia);
    strProvider =
clsSegRijndaelSimple.Decrypt(System.Configuration.ConfigurationManager.AppSettings["provider
"]);

    dbProvider = DbProviderFactories.GetFactory(strProvider);
    DbConnection nuevaConexion = dbProvider.CreateConnection();
    nuevaConexion.ConnectionString = strConnectionString;

    try
    {
        nuevaConexion.Open();
        return nuevaConexion;
    }
    catch (DbException sqlEx)
    {
        throw sqlEx;
    }
    catch (Exception Ex)
    {
        throw Ex;
    }
}
```

Imagen 3-7. Fragmento de código C# para la conexión a Base de Datos.

3.4. PROCEDIMIENTOS ALMACENADOS

La persistencia de datos, producto de la interacción del usuario con el sistema de información, debe ser transparente y no comprometer la integridad de la misma, a la vez de evitar intrusiones o ataques que vulneren la seguridad de la misma.

El manejo de datos tanto de consulta como de inserción, actualización y eliminación, se hace a través de procedimientos almacenados, que son instrucciones ejecutadas en el servidor de base de datos y pueden recibir parámetros de entrada o de salida.

El procedimiento almacenado es nombrado indicando el esquema al que pertenece, el prefijo *sp*, el verbo infinitivo de la acción que realiza y el nombre de la tabla sobre la que opera: *empleado.spInsertarEmpleado*.

```
CREATE PROCEDURE [empleado].[spInsertarEmpleado]
        @empPaterno          VARCHAR(30)
       ,@empMaterno          VARCHAR(30)
       ,@empNombre                  VARCHAR(30)
       ,@empFechaIngreso     DATETIME
       ,@empNoEmp                   INT
AS
BEGIN
       SET NOCOUNT ON;

       INSERT INTO [empleado].[empleado]
                          ([idEmpleado]
                          ,[empFechaCaptura]
                          ,[empPaterno]
                          ,[empMaterno]
                          ,[empNombre]
                          ,[empSexo]
                          ,[empFechaIngreso]
                          ,[empVigente]
                          ,[empFotografia]
                          ,[empNoEmp])
                VALUES
                          (NEWID()
                          ,GETDATE()
                          ,@empPaterno
                          ,@empMaterno
                          ,@empNombre
                          ,NULL
                          ,@empFechaIngreso
                          ,1
                          ,NULL
                          ,@empNoEmp)
END
```

Imagen 3-8. Procedimiento almacenado SQL para la inserción de un registro.

Para comunicar el procedimiento con la aplicación, en la capa de datos, se genera el método que envía los valores a los parámetros del procedimiento y recupera la información devuelta por este, además se manejan las posibles excepciones generadas durante la operación.

```
public static int insertarEmpleado(empleado objEmpleado)
{
    clsDatConexion objConexion = new clsDatConexion();
    DbConnection dbConexion = objConexion.getConexion();

    DbCommand dbComando = objConexion.dbProvider.CreateCommand();
    DbTransaction dbTrans = dbConexion.BeginTransaction();
    dbComando.Connection = dbConexion;
    dbComando.Transaction = dbTrans;

    try
    {
        dbComando.CommandType = CommandType.StoredProcedure;
        dbComando.CommandText = "empleado.spInsertarEmpleado";
        dbComando.Parameters.Clear();

        clsParametro objParametro = new clsParametro();

        objParametro.llenarParametros(ref dbComando, DbType.Int32, "@empNoEmp",
objEmpleado.empNoEmp);
        objParametro.llenarParametros(ref dbComando, DbType.DateTime, "@empFechaIngreso",
objEmpleado.empFechaIngreso);
        objParametro.llenarParametros(ref dbComando, DbType.String, "@empPaterno",
objEmpleado.empPaterno);
        objParametro.llenarParametros(ref dbComando, DbType.String, "@empMaterno",
objEmpleado.empMaterno);
        objParametro.llenarParametros(ref dbComando, DbType.String, "@empNombre",
objEmpleado.empNombre);

        dbComando.ExecuteNonQuery();
        dbTrans.Commit();

        return 1;
    }
    catch (DbException)
    {
        dbTrans.Rollback();
        return 0;
    }
    catch (Exception)
    {
        dbTrans.Rollback();
        return 0;
    }
    finally
    {
        clsDatConexion.cerrarTransaccion(dbConexion);
    }
}
```

Imagen 3-9. Método C# para la inserción de un registro.

Para mantener la separación entre el proveedor de datos y el sistema, se utilizó la clase del *namespace System.Data.Common*: *DBConnection*. Esta clase provee los métodos necesarios para enviar los parámetros de entrada y recibir los parámetros de salida utilizados por los procedimientos almacenados.

4.4.1. TRANSACCIONES

Los sistemas deben ser capaces de mantener la integridad de los datos, para cumplir este requisito es necesario manejar transacciones, que son secuencias de operaciones realizadas como una sola unidad lógica de trabajo. El manejo de transacciones es responsabilidad del programador y el gestor de base de datos debe tener la capacidad de soportar este tipo de actividades, es decir, un manejador transaccional.

Comúnmente, en un proceso de almacenamiento de datos por ejemplo, se invocan dos o más procedimientos almacenados, o incluso uno solo procedimiento es invocado un número repetitivo de veces en un evento de estos. *SQL Server* soporta transacciones a nivel de procedimientos almacenados, pero dada la situación anteriormente descrita es necesario manejar las transacciones a un nivel superior, en este caso, la capa de datos.

.NET Framework provee la capacidad de manejar transacciones a nivel de código fuente mediante clases específicas contenidas en el namespace System.Data.Common: DBTransaction. Esta clase permite iniciar una transacción, si no se genera ningún error o excepción se invoca el método commit, y en caso de existirlo, el método rollback.

```
DbTransaction dbTrans = dbConexion.BeginTransaction();

.
.
.

try
{
    dbTrans.Commit();
}
catch(DbException)
{
    dbTrans.Rollback();
}
```

Imagen 3-10. Uso de transacciones en la capa de datos.

3.5. DISEÑO DE LA INTERFAZ DE USUARIO

El siguiente paso en el diseño de los sistemas de información es el diseño de la interfaz de usuario, cuyo objetivo principal debe ser el de permitir a los usuarios obtener e introducir información al sistema. Las interfaces de usuario deben ser eficaces, al lograr que el usuario satisfaga sus necesidades, y eficiente, al mejorar la velocidad de captura y la reducción de errores.

Durante esta etapa, tenemos que centrarnos en la idea de que el sistema es quien debe adaptarse al usuario y no el usuario al sistema, por lo que las interfaces deberán ayudar al usuario a realizar las tareas que espera realizar con el mínimo de interrupciones.

Para lograr lo anterior, se propone las siguientes reglas básicas:

- La distribución de los menús debe ser consistente en cuanto a presentación y el orden de las instrucciones. No saturar el menú principal con demasiados elementos, en vez de eso, agrupar los que se guarden relación.
- Cualquier acción que implique eliminar información, debe ser verificada por el usuario, así se pueden evitar muchos problemas.
- Permitir deshacer una acción o varias acciones.
- Validar todos los posibles errores del usuario, desde la captura de información incorrecta hasta el hecho de que pulse opciones fuera del ciclo de navegación.
- Utilizar verbos de acción simple, en infinitivo así como frases cortas para nombrar acciones.
- No saturar con información innecesaria minimizando errores.
- Cuando existan errores, mostrar al usuario mensajes significativos que le permitan identificarlos.
- Agrupar el contenido en bloques significativos.
- Utilizar ventanas auxiliares para segmentar los diferentes tipos de información.
- Distribuir el espacio disponible en la pantalla de manera eficiente.
- Los controles o acciones más utilizadas deben ser más grandes y distinguibles fácilmente.
- Evitar mensajes de alerta innecesarios y solo generar aquellos que contienen información útil.

- Minimizar el número de acciones de entrada de datos que hace el usuario.
- Permitir la flexibilidad en la forma de ingreso de datos, ya sea por teclado, ratón, etc.
- Desactivar aquellas acciones en el contexto actual, con el fin de evitar acciones erróneas.
- Se deben manejar aspectos como la combinación de colores utilizados, no dejando de lado los colores institucionales.

Para diseñar la interfaz de usuario, se utilizar los siguientes elementos:

- Página maestra o *master page*.
- Hoja de estilo en cascada o *Cascading Style Sheets CSS*.
- JavaScript.
- Controles ASP.NET.

Las páginas maestras proporcionan el aspecto y el comportamiento estándar que desea para todas las páginas del sitio Web [8]. Así pues, la distribución de los elementos básicos (cabecera, pie de página, menú principal, etc.) quedo se la siguiente manera:

Imagen 3-11. Diseño de la interfaz de usuario de un sistema de información Web.

Para generar el diseño visual, combinación de colores y ubicación de elementos, se utilizaron las hojas de estilo junto con elementos de ASP.NET y HTML:

```
body, html
{
        background-color: #ebeace;
        font-family: Verdana, Geneva, sans-serif;
        font-size: 11px;
        color: #252525;
        margin: 0;
        padding: 0;
        border: 0;
}

.Menu
{
        width: 200px;
        font-weight: bold;
        height: 34px;
        float: left;
        padding-right: auto;
        margin: 118px 0px 0px 0px;
        font-family: Verdana, Geneva, sans-serif;
        font-size: 10px;
        text-align: center;
}
```

Imagen 3-12. Hoja de estilo utilizada en la interfaz gráfica.

```
<table width="100%">
    <tr>
        <td align="left">
            <asp:Label ID="lblUsuario" runat="server" Text=""></asp:Label>
        </td>
        <td align="right">
            <asp:Label ID="lblFecha" runat="server" Text=""></asp:Label>
        </td>
    </tr>
</table>
```

Imagen 3-13. Implementación de una hoja de estilo en etiquetas ASP.

Algunos elementos interactivos de las pantallas como resaltar los cuadros de texto o convertir su contenido en mayúsculas se hizo a través de scripts de JavaScript.

```
function onFocus(objeto)
{
    objeto.style.backgroundColor = "#FFFF99";

}
function onLostFocus(objeto)
{
    objeto.style.backgroundColor = "#FFFFFF";
    objeto.value = objeto.value.toUpperCase();
}
```

Imagen 3-14. Fragmento de script utilizado en la interfaz gráfica.

Estas herramientas son de uso y mantenimiento sencillo, ya que actualmente existen infinidad de manuales de usuario los cuales ilustran a través de ejemplos su sintaxis.

La pantalla está diseñada para adaptarse a la resolución de la pantalla en donde es visualizada. Algunas de las restricciones que se tienen en este diseño en el navegador de Internet donde se debe desplegar, ya que si bien funciona en exploradores diferentes a Internet Explorer, Mozilla Firefox por ejemplo tiene la opción de desactivar la ejecución de scripts lo que impacta en el desempeño de la aplicación, no solo por JavaScript, sino porque algunos controles de ASP.NET basan su funcionamiento en este tipo de tecnología.

Es por esta razón que, se hace la anotación en el pie de página para que el usuario este consciente de que el mejor desempeño de la aplicación lo tenga en Internet Explorer versión 7.0 o superior.

3.6. AUTENTICACIÓN DE USUARIOS

La *autenticación* es el proceso de detectar y comprobar la identidad de una entidad de seguridad mediante la evaluación de las credenciales del usuario y la validación de las mismas consultando a una autoridad determinada. La autenticación dentro de los sistemas de información del SPF, a menos que se especifique lo contrario, consistirá en validar el nombre de usuario y contraseña de aquella persona o entidad que intenta hacer uso de dicho sistema de información.

El siguiente diagrama de flujo describe el proceso de autenticación de un usuario:

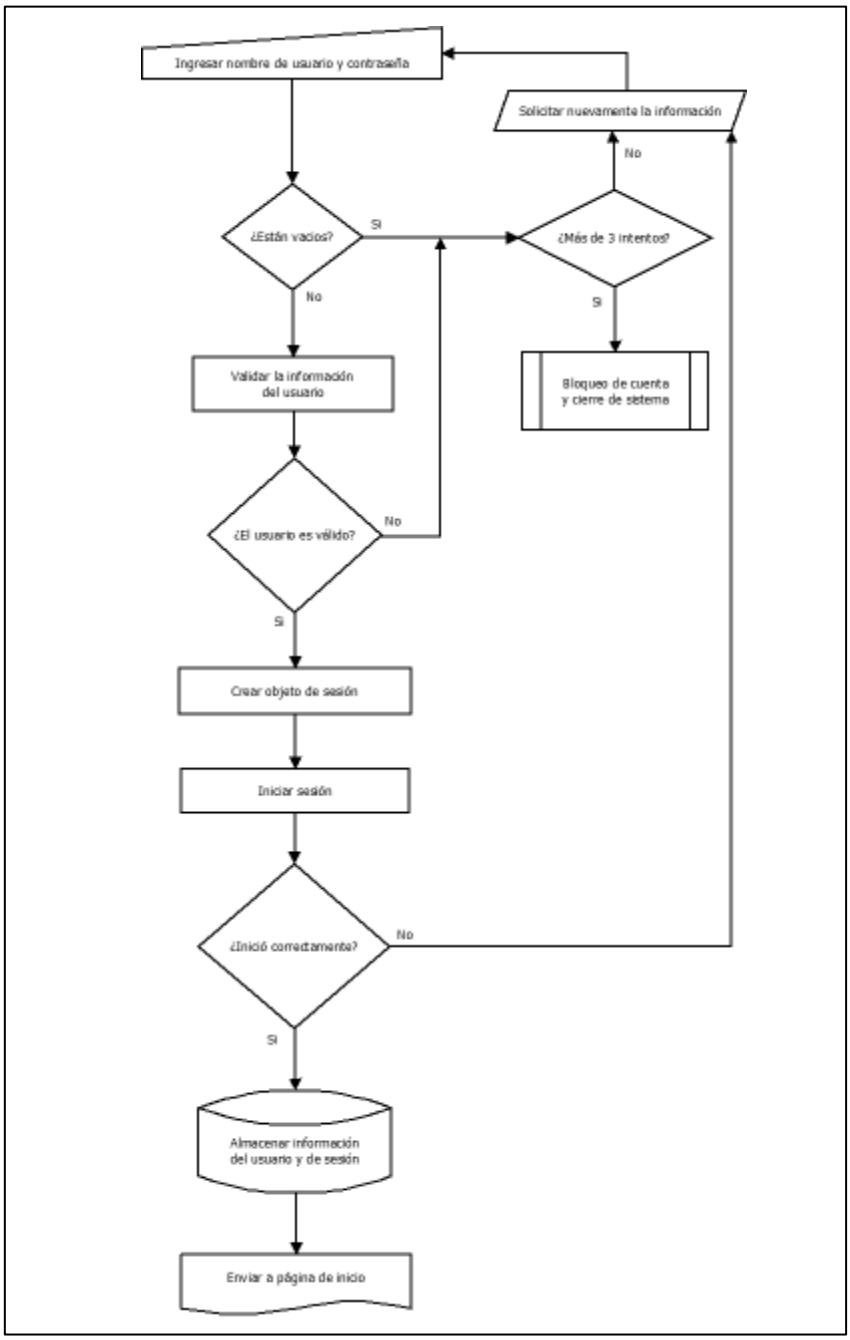


Imagen 3-15. Diagrama de flujo para la autenticación de usuarios.

La implementación de este flujo se muestra a continuación:

- Se tiene una página Aspx en la que se solicita al usuario que ingrese su información.

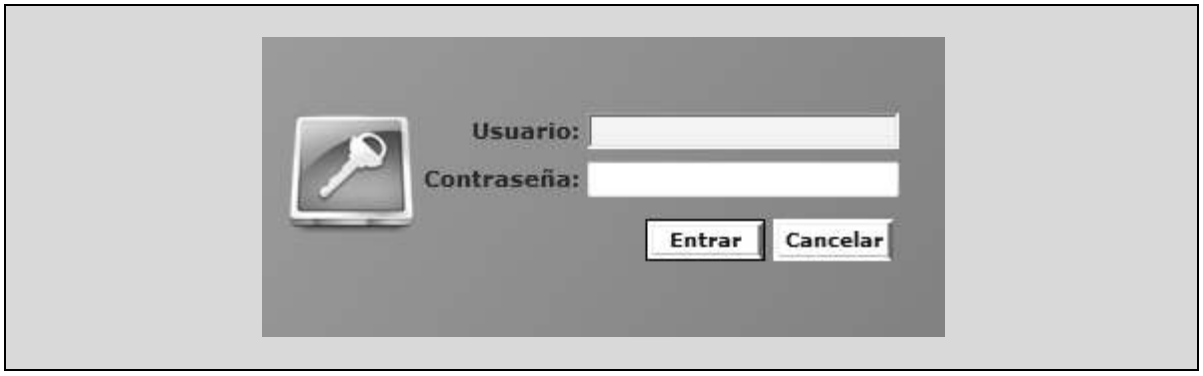


Imagen 3-16. Pantalla de autenticación de usuarios.

- En caso de omitir alguno de los campos, visualizamos la siguiente alerta.

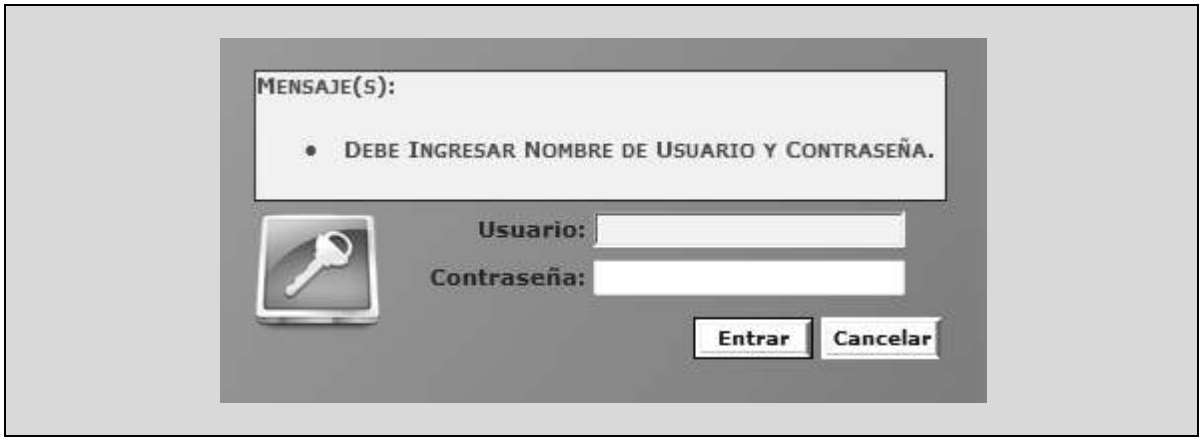


Imagen 3-17. Mensajes de alerta en la autenticación de usuarios.

- Cuando la información fue ingresa, se procede a validar las credenciales del usuario. En caso de que sean incorrectas, se envía una alerta al usuario.

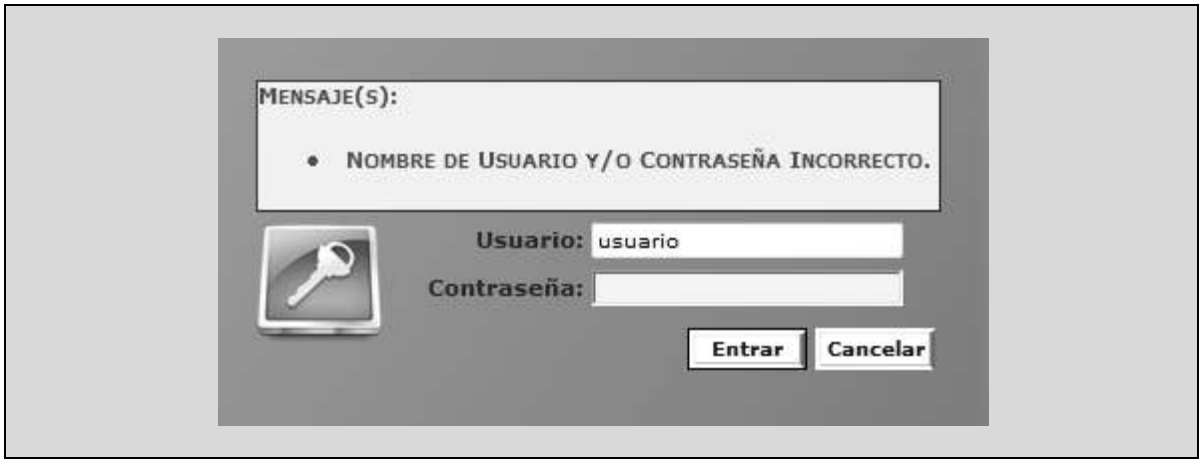


Imagen 3-18. Mensajes de alerta en la autenticación de usuarios.

- Si el usuario ha incurrido más de tres veces en una omisión de datos y/o credenciales erróneas, se envía una alerta al usuario de que su cuenta ha sido bloqueada y la pantalla se cierra.
- Si la información ingresada es correcta, se inicia la sesión del usuario y se le redirige a la pantalla de inicio.

Imagen 3-19. Pantalla de inicio en los sistemas de información Web.

La codificación de algunos puntos se explica a continuación:

- Para crear un objeto de sesión se utiliza una instancia de la entidad *sesión* en la que mediante una llamada a la capa de datos, se extrae la información del usuario (nombre completo, perfil de usuario, etc.) así como también su dirección IP, la fecha de inicio de sesión, el estatus de su sesión, etc. Con esta información se crea un registro en la tabla *sesión* con la información de la sesión que se acaba de crear.

```
// Se crea una instancia de la entidad sesion
hSesion objSession = new hSesion
                            {
                                hUsuario = new hUsuario
                                            {
                                                perfil = new perfil(),
                                                empleado = new empleado()
                                            }
                            };
```

Imagen 3-20. Instanciación de la clase Sesión.

```
SELECT [idUsuario]
         ,[su].[idPerfil]
         ,[sp].[perDescripcion]
         ,[empPaterno]
         ,[empMaterno]
         ,[empNombre]
  FROM [seguridad].[usuario] [su]
  LEFT OUTER JOIN [seguridad].[perfil] [sp]
       ON [sp].[idPerfil] = [su].[idPerfil]
  LEFT OUTER JOIN [empleado].[empleado] [ee]
       ON [ee].[idEmpleado] = [su].[idEmpleado]
 WHERE [usuVigente] = 1
   AND [usuLogin] LIKE @login
```

Imagen 3-21. Consulta SQL para la extracción de información de un usuario.

```
DECLARE @usuarioActivo INT
DECLARE @valido INT

SELECT @usuarioActivo = COUNT([idUsuario])
  FROM [seguridad].[sesion]
 WHERE [seEstatus] = 1
   AND [idUsuario] = @idUsuario

IF @usuarioActivo = 0
       BEGIN
       INSERT INTO [seguridad].[sesion]
                          ([seSessionID]
                          ,[idUsuario]
                          ,[seIP]
                          ,[seInicio]
                          ,[seIntentos]
                          ,[seEstatus])
                VALUES
                          (@seSessionID
                          ,@idUsuario
                          ,@seIP
                          ,GETDATE()
                          ,@seIntentos
                          ,@seEstatus)
              SET @valido = 1
              RETURN @valido
       END
ELSE
       BEGIN
              DECLARE @espera INT
              SELECT @espera = DATEDIFF(mi, [seInicio], GETDATE())
                FROM seguridad.sesion
               WHERE [seEstatus] = 1
                 AND [idUsuario] = @idUsuario
                 AND [seFin] IS NULL

              IF @espera > -1
                     BEGIN
                            UPDATE [seguridad].[sesion]
                               SET [seEstatus] = 2,
                                   [seFin] = GETDATE()
                             WHERE [idUsuario] = @idUsuario
                               AND [seEstatus] = 1

                            INSERT INTO [seguridad].[sesion]
                                               ([seSessionID]
                                               ,[idUsuario]
                                               ,[seIP]
                                               ,[seInicio]
                                               ,[seIntentos]
                                               ,[seEstatus])
                                        VALUES
                                               (@seSessionID
                                               ,@idUsuario
                                               ,@seIP
                                               ,GETDATE()
                                               ,@seIntentos
                                               ,@seEstatus)
                            SET @valido = 1
                            RETURN @valido
                     END
              SET @valido = 0
              RETURN @valido
       END
```

Imagen 3-22. Procedimiento almacenado SQL con el que se inicia sesión del usuario autenticado.

3.7. PERFILES

Cada usuario para el desempeño de sus funciones estará asignado a un *perfil*. El usuario es el personaje principal de la trama informática, es el principio y el fin del ciclo de transferencia de la información: él solicita, analiza, evalúa y recrea la información [2].

Los perfiles de usuario proporcionan información sobre aquellas operaciones sobre las cuales tiene permiso un usuario, ya que estas le permiten alcanzar los objetivos para los cuales ingresa al sistema. Cada usuario debe estar asignado a un único perfil y, en caso de no existir, se debe gestionar la definición del mismo para poderlo incluir dentro de las especificaciones.

La administración de perfiles hará uso de tres tablas en la base de datos: *operación*, *perfil* y *permiso*. *Operación* almacenará todas aquellas acciones que conforman el sistema de información. *Perfil* almacenará todos aquellos perfiles que se definan para el sistema de información. *Permiso* es la relación entre el perfil y las operaciones, es decir, aquellas acciones que determinado perfil puede realizar.

Imagen 3-23. Diagrama entidad - relación del para la administración de perfiles.

Cuando un usuario ingrese al sistema se recupera el perfil al que pertenece, así con esta información se ejecuta un algoritmo que consulta los permisos del perfil y personaliza el menú principal en función de este perfil.

Este algoritmo se describe en el siguiente diagrama de flujo:

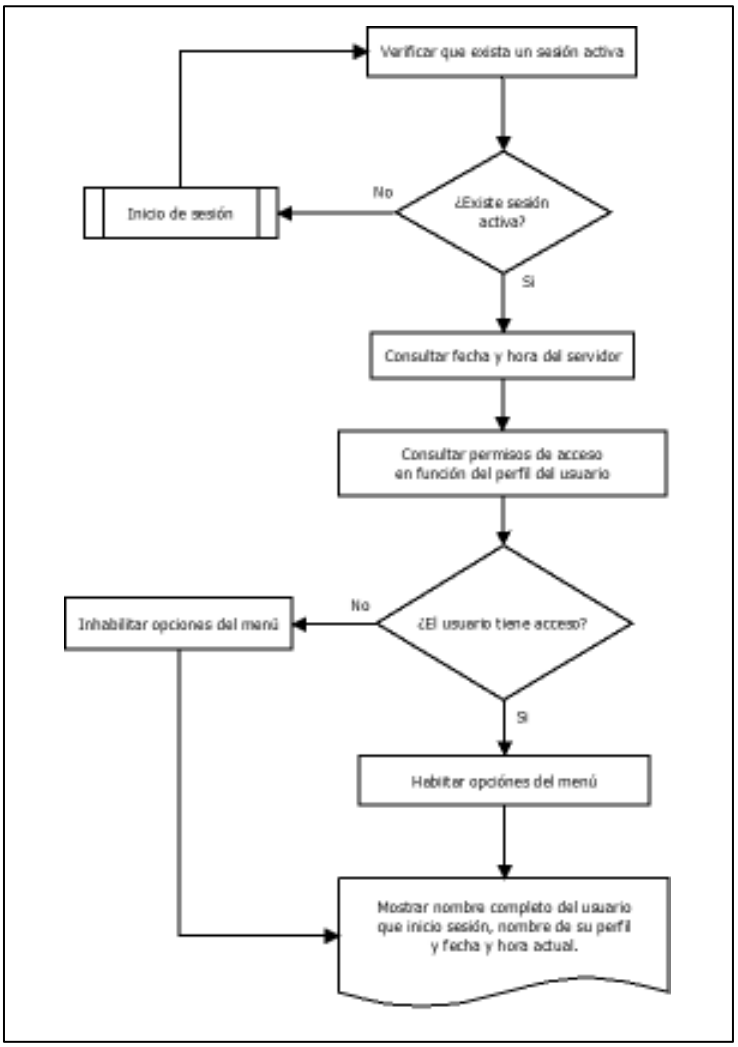


Imagen 3-24. Diagrama de flujo de la administración de perfiles de usuario.

Para verificar que existe una sesión activa, es necesario verificar que la variable de sesión donde se almacena la información del usuario exista. Para evitar la repetición de código, esta validación se hace en la página maestra de nuestra aplicación, con esto, todas aquellas páginas que dependan de ella aplicaran la misma política. En caso de que la sesión no exista, se redirige al usuario a la pantalla de autenticación mostrando la alerta correspondiente.

Imagen 3-25. Mensaje de error cuando no se ha iniciado correctamente una sesión.

Cuando la sesión es correcta, se consulta la hora y fecha del sistema para mostrarla en la pantalla en el área correspondiente así como el nombre del usuario y su perfil.

Posteriormente se ejecuta un método que consulta los permisos a los cuales tiene acceso el perfil del usuario. El proceso de consulta se ejecuta en la capa de datos y procesamiento de los permisos se realiza en la capa de negocio. Esta rutina consiste en métodos recursivos que recorren uno a uno los permisos y deshabilitan aquellas opciones del menú principal con de fin de evitar que se ingrese a las opciones no permitidas.

```
public static void aplicarPermisos(Menu mnuPrincipal, permiso objPermiso, hSesion
objSesion)
{
    foreach (MenuItem item in mnuPrincipal.Items)
    {
        item.Enabled = clsDatPermiso.consultarPermiso(objPermiso.idPerfil,
Convert.ToInt16(item.Value), objSesion);
        permisosRecursivos(item, objPermiso, objSesion);
    }
}

private static void permisosRecursivos(MenuItem miOpcion, permiso objPermiso, hSesion
objSesion)
{
    foreach (MenuItem item in miOpcion.ChildItems)
    {
        item.Enabled = clsDatPermiso.consultarPermiso(objPermiso.idPerfil,
Convert.ToInt16(item.Value), objSesion);
        permisosRecursivos(item, objPermiso, objSesion);
    }
}
```

Imagen 3-26. Método recursivo para la aplicación de permisos.

Este proceso, sin embargo, no evita que el usuario ingrese directamente a través de la dirección URL de cada pantalla, por lo que adicionalmente, cuando se intenta ingresar en una pantalla en particular, se consulta si esta está autorizada al usuario. Cuando esto no es así, se le redirige a la pantalla de inicio.

```
if (!Page.AppRelativeVirtualPath.Equals("~/frmInicio.aspx"))
     if (!clsNegPermiso.permitirPagina(objPermiso, Page.AppRelativeVirtualPath, objSesion))
            Response.Redirect("~/frmInicio.aspx");
```

Imagen 3-27. Fragmento de código donde se invoca el método para aplicar permisos.

3.8. CONSULTA DE INFORMACIÓN

La consulta de información es uno de los procesos más comunes en los sistemas de información, entre otras cosas, porque permite el analizar e interpretar la información almacenada previamente con el fin de apoyar en la toma de decisiones o simplemente en un proceso de consulta de datos [9].

Se plantea el siguiente esquema para la consulta de información de la base de datos:

- En la capa de datos se invoca el procedimiento almacenado con la consulta SQL necesaria para consultar la información de la o las tablas que se requieren, haciendo uso de *joins* para relacionarlas.
- La información arrojada por la consulta se almacena temporalmente en un objeto de la clase *DataTable* que representa una tabla de datos en memoria; este objeto es personalizado a través de un esquema de lenguaje de definición de esquemas XML (XSD) mejor conocidos como *DataSet*. Mediante este mecanismo se puede indicar el tipo de dato que corresponde a cada columna del objeto *DataTable*.
- El objeto *DataTable* es enviado a la capa de negocio donde es interpretado a un esquema orientado a objetos, para lo cual se recurre a la clase *List* que representa una lista de objetos previamente establecido en su instanciación. Los métodos que implementa esta clase permite hacer una manipulación más rápida y segura de los objetos que contiene además de ser económico en cuanto al manejo de memoria, ya que a diferencia de un arreglo, un objeto de la clase *List* solo ocupa porciones de memoria de acuerdo al contenido del mismo.

- Una vez generada la lista que representa la información que se consultó en un inicio, la capa de datos envía a la de presentación una lista con los objetos cargados de información.
- En la capa de presentación se hace uso de un control de ASP.NET llamado *GridView* que entre sus funciones permite desplegar un origen de datos en una tabla donde cada columna representa un campo y cada fila representa un registro. Este control se personaliza para indicar que propiedad de los objetos que contiene la lista se mostrara en cada columna.
- El control *GridView* permite de una forma rápida y sencilla la visualización de información y solo se requiere configurar las columnas asociando la propiedad que se desea mostrar de un objeto y la combinación de colores en columnas y renglones.

Imagen 3-28. Ejemplo de implementación de una consulta de información.