FRBACOMMERCE

Estrategia

La Banda del Chavo

Numero de grupo: 25
Curso: K3014
Profesor: Marcelo Moscuzza
Fecha de entrega: 18-06-14

Integrantes

Nicolas Orchow (responsable) - 1467001
Axel Suvalski - 1472902
Julian Fuks - 1472914
Eric Lifszyc – 1466550

Índice

Estrategia

Para poder realizar el trabajo práctico, obviamente tuvimos que definir por donde íbamos a comenzar. Con lo único que contábamos era con la tabla maestra y la información correspondiente brindada por los ayudantes, y la consigna planteada.

Lo primero que hicimos fue realizar el DER, es decir, un diagrama donde podíamos observar las distintas tablas que intervenían en el negocio planteado. Para esto, como dijimos anteriormente tuvimos que leer varias veces la consigna y ayudarnos con la estructura e información que contábamos en la tabla maestra.

Una vez realizado el DER, comenzamos a trabajar en la resolución misma del sistema realizado en la plataforma .net/C#. Nos planteamos que antes de realizar cualquier funcionalidad, debíamos poder comunicarnos con la base de datos y tratar de obtener datos e insertar datos en la misma. Para esto, realizamos tres clases para poder realizar esto. Dichas clases son: '**DatabaseManager**' (encargada de manejar la conexión con la base de datos, que implementa el patrón Singleton para que exista una única conexión por sistema), '**DatabaseQueries**' (encargada de manejar los nombres de todas las consultas existentes en la base de datos), '**StoreProcedure**' (encargada de impactar a la base de datos, es la que ejecuta los métodos de ExecuteNonQuery, ExecuteScalar y ExecuteReader) y '**SPParameter**' (encargada de representar a los parámetros que espera una determinada consulta).

Una vez que pudimos efectivamente comunicarnos con la base de datos, pudimos comenzar a desarrollar las distintas funcionalidades. La primer funcionalidad que realizamos fue el Login, que creemos que era lo más importante para poder comenzar a entendernos con la plataforma.

Estructura general del proyecto

La solución planteada que encontramos, cuenta con los siguientes proyectos:

- Configuration: es el proyecto encargado de manejar las distintas configuraciones presentes en el archivo de configuración del proyecto, que necesita el sistema para funcionar. Por ejemplo, la connection string de la base de datos, la hora del sistema.
- Filters: es el proyecto que representa los datos introducidos en los filtros de búsqueda para las consultas, tanto para las búsquedas exactas (comparan por '**=**'), como las inexactas (comparan por '**LIKE**').
- FrbaCommerce: es el proyecto que almacena todos los formularios necesarios para poder utilizar todos los casos de usos planteados en la consigna, como así también al archivo de configuración del sistema ('**app.config**').
- Persistance: es el encargado de representar a cada entidad particular que encontramos en la consigna, como así también las clases encargadas de manejar a dichos objetos en la base de datos (tanto obtenerlas, como impactar algún cambio).

- Session: es el proyecto encargado de manejar información del usuario logueado que está ejecutando el sistema. Cuenta con la información del usuario logueado, el rol con el que eligió loguearse (en caso de tener más de uno logueado) y el momento de acceso.
- Tools: se encarga de realizar validaciones de tipos a partir de datos introducidos por el usuario, como así también encriptar la contraseña de un nuevo usuario.

Consideraciones generales

A medida que desarrollábamos la solución para el principal problema planteado, ocurría que nos encontrábamos con situaciones ambiguas, ocasiones donde ocurría que para un mismo enunciado, cada grupo encontraba respuestas distintas, es por esto que realizamos el siguiente listado.

A continuación, presentamos una serie de decisiones concretas que hemos tomado a la hora de desarrollar la solución, para poder solucionar estas distintas situaciones problemáticas:

- En cierta parte de la consigna (donde se habla de las condiciones de entrega de la solución encontrada) se menciona que se debe generar un usuario hardcodeado que posea el rol '**Administrador General**'. Lo que nos llamó la atención de este enunciado, es que no se habla de igual forma que se habla del rol '**Administrativo**' en el resto de la consigna. Es por este motivo, que decidimos crear un rol distinto en la tabla '***TL_Rol***'.
- En nuestro modelo relacional de datos, planteamos una tabla '***TL_Usuario***', que contenía únicamente información de los mismos (Nombre de usuario, contraseña, activo, habilitado, etc.). A la hora de realizar la migración de los datos desde la tabla maestra, no contábamos con los datos del usuario, sino que poseíamos solo la información de los clientes y las empresas. Nuestro primer inconveniente era como generar los distintos usuarios si no contábamos con dicha información. Por un lado, para los usuarios que poseían un registro asociado de tipo '**Cliente**', decidimos que el nombre de usuario sea su DNI, y para los usuarios que poseían un registro asociado de tipo '**Empresa**', el nombre de usuario iba a ser su CUIT.
- Un problema similar ocurrió con la contraseña de los usuarios. Para los que fueron generados a partir de la migración, como no contábamos con esta información, decidimos establecer como contraseña fija "*temporal*". A su vez, como la contraseña debía permanecer guardada en la base de datos encriptada con el algoritmo SHA256, decidimos buscar un encriptador online en internet y le aplicamos el algoritmo de encripción a la cadena "*temporal*". Cuando ya teníamos la cadena encriptada, colocamos dicho valor "hardcodeado" en la consulta que genera la migración de los usuarios desde la tabla maestra. Para los usuarios nuevos, es decir, los que se iban a ir generando mediante nuestro sistema desarrollado, decidimos implementar una clase '**SHA256Helper**' que a partir de una cadena, retorna su valor encriptado. Luego, este nuevo valor calculado se le pasa por parámetro a la consulta que inserta los nuevos usuarios.

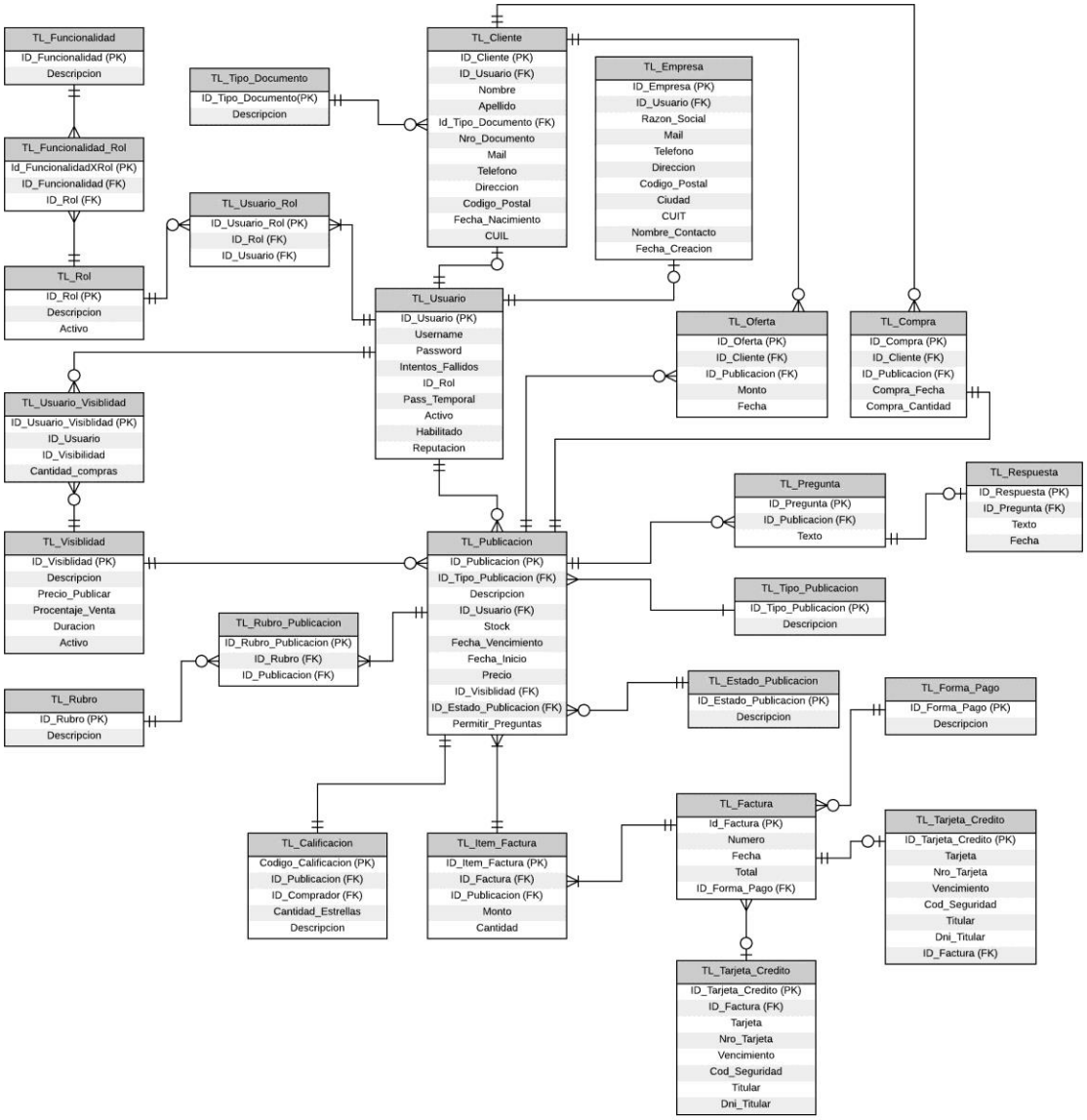
- Durante la etapa de migración, detectamos que los valores almacenados en la tabla maestra correspondiente a las calificaciones otorgadas, podían valer desde 1 a 10. En la consigna planteada, mencionaba que el modelo de datos, las calificaciones podían valer únicamente del 1 al 5, por lo que debimos llevarlo a cabo en nuestro esquema. Para poder realizar esta "transformación" de posibles valores, decidimos tomar los valores existentes en la tabla maestra y dividirlos por dos. Por consiguiente, ocurría que para los valores impares, la división iba a arrojar resultados decimales, por lo que nuestro campo '**Cantidad_Estrellas'** de la tabla '***TL_Calificacion***' dejaría de ser entero (como lo habíamos planteado en un primer momento) para pasar a ser decimal.
- Durante la migración de datos desde la tabla maestra, nos dimos cuenta que en nuestro modelo de datos contábamos con tablas cuya clave principal (PK) ya poseíamos el valor informado en los registros de la tabla maestra. Por lo tanto, debíamos encontrar una forma de insertar los valores que contábamos en la tabla maestra como PK de las tablas de nuestro DER, y a medida que íbamos insertando nuevos registros en dichas tablas a través de nuestro sistema, la clave principal de dichos nuevos registros debían continuar a la última clave principal insertada durante la migración. Para poder realizar esto, tuvimos que primero crear la tabla correspondiente con el campo como clave principal, luego indicarle al motor que nos permita insertar valores en dicho campo (hecho que no se puede realizar si tenemos un campo auto numérico), realizar la migración de los datos, volver a indicarle al motor de base de datos que a partir de la finalización de la migración ya no nos permita insertar nuevos valores en el campo de clave principal y en último lugar volver a setearle la última clave insertada durante la migración como el ultimo valor de la secuencia de campos claves. Esto permite que la clave de los nuevos registros insertados continúen la secuencia de claves insertadas durante la migración.
- Para el manejo del usuario logueado, en vez de instanciar un objeto Usuario en el momento que se loguea el usuario e ir pasando dicho objeto entre las distintas pantallas que lo necesiten, decidimos crear una clase global '**SessionManager**', encargada de contener la información del usuario logueado, el rol con el que se logueo y el momento de login.
- A la hora de realizar los filtros de búsqueda en todas las funcionalidades que los necesitan, decidimos implementar dos tipos de búsquedas. Por un lado, una búsqueda exacta (filtrar por '**=**'), y por otro, una búsqueda inexacta (filtrar por '**LIKE**'). La primera de estas permite realizar filtrar resultados pasándole cualquier tipo de dato (texto libre, selección acotada, selección de fechas), mientras que la última permite únicamente filtrar por tipos de datos de texto, es decir, que no permite realizar una búsqueda inexacta con valores numéricos o fechas. Para poder realizar esto, decidimos inhabilitar los controles que no cumplen con dicha condición y volver a habilitarlos en el caso que el usuario decida realizar una búsqueda exacta.

- Durante el desarrollo mismo de la solución, tuvimos que decidir si la eliminación de los registros (en los distintos ABM's) iban a ser tomadas como bajas lógicas (continuaría existiendo el registro pese a la eliminación) o bajas físicas (el registro a eliminar no existirá mas). Nuestra elección fue tomar el camino de la primera, es decir, la de las bajas lógicas. Para esto, fue necesario agregar una columna '**Activo**' a las distintas tablas donde se debían realizar eliminaciones. Cuando efectivamente se realizaba esta acción sobre alguna de dichas tablas, en vez de ejecutar una consulta '**DELETE**', efectuábamos una consulta '**UPDATE**' que lo único que realizaba era setearle el valor "*False*" al campo '**Activo**' del registro a eliminar.
- En la consigna se planteaba la situación en la que se daba de baja (ver punto anterior) un rol, donde a continuación se le debía eliminar dicho rol a los usuarios que lo posean asignado. Para poder realizar esto, en primer instancia, debimos crear una tabla '***TL_Usuario_Rol***' que efectivamente relacione a los usuarios con sus roles asignados. Para poder realizarle la baja del rol a los usuarios que lo poseían asignado, decidimos crear un trigger (disparador) que se iba a ejecutar siempre luego que se modifique (AFTER UPDATE) la tabla '***TL_Rol***'. Este disparador se fijara si el nuevo valor del campo '**Activo**' del rol modificado fue seteado en "*False*". En dicho caso, realizara la baja física (es la única en todo el sistema) de todos los registros de la tabla '***TL_Usuario_Rol***' cuyo campo '**ID_Rol**' coincida con el del rol modificado. La elección de realizar una baja física de los registros, se debió a que esta tabla va a crecer exponencialmente a medida que se generen nuevos usuarios, por lo que es preferible ir reduciéndole el tamaño a medida que se den de baja los roles. Además, esta tabla solamente se encuentra presente porque necesitamos conocer información particular de una relación que es de muchos a muchos (usuarios con roles), por lo que una vez que los roles dejen de existir, ya no nos resulta significativa dicha información.
- A la hora de realizar la migración de las visibilidades, nos dimos cuenta que en la tabla maestra no contábamos con la información correspondiente a la "prioridad" que cada una de estas debía tener, para poder conocer de qué forma debían aparecer en el listado de publicaciones. Para solucionar esto, decidimos que cuanto mayor sea el precio de la visibilidad de la publicación (información que sí conocíamos), mayor prioridad iba a tener, y por consiguiente, iba a aparecer primero en el listado.
- En el caso de uso de '**Comprar/Ofertar**' publicaciones, se establecía que el listado de las publicaciones debía estar paginado, es decir, se debería recorrer mediante paginas (primera, anterior, siguiente, ultima). Sin embargo, no establecía explícitamente de cuantas publicaciones debía ser cada página. El grupo decidió mostrar diez publicaciones por página.
- A la hora de realizar la modificación de una publicación existente, en un momento ocurrió que le pasábamos todos los valores para modificar todos los campos de la tabla '***TL_Publicacion***', hecho que si es necesario para la inserción de una nueva publicación. Releyendo la consulta que habíamos creado, nos dimos cuenta que había ciertos valores que no eran necesarios

pasarlos por parámetros para la modificación, ya que por el negocio mismo planteado en la consigna sabíamos que eran inmutables una vez que la publicación ya se había insertado en la base de datos. Estos valores que decidimos no pasar por parámetro son: **'ID_Usuario', 'Fecha_Inicio', 'Fecha_Vencimiento**'.

- Como se puede observar en el modelo relacional de datos, la tabla '***TL_Usuario***' cuenta con dos campos '**Activo**' y '**Habilitado**', que a priori, uno podría pensar que representan lo mismo. Sin embargo, en nuestra solución planteada esto no ocurre. El campo '**Activo**' se modifica una vez que el usuario trata de ingresar al sistema cinco veces y no logra hacerlo ya que los datos de acceso son incorrectos. Para poder volver a estar activo, el único que puede realizar esto es un usuario que sea de tipo '**Administrador**', que deberá ingresar al formulario de '**ABM de Usuarios**', seleccionar el usuario que se encuentra inactivo y resetearle dicho valor. El campo '**Habilitado**' se encuentra para poder realizar la baja del usuario, es decir, representa cuando el usuario deja de existir. Esta operación, a diferencia de la operación de re-activar un usuario, es irreversible (no se puede re-habilitar a un usuario dado de baja).

DER (Diagrama Entidad Relacion)



Para verlo ampliado, entrar a: https://www.lucidchart.com/invitations/accept/7a881f7f-7fba-4b89-a99c-883ecc7ecc31

Consultas

A la hora de realizar las consultas, tras investigar sobre el motor de la base de datos y como conectarnos con este desde el C#, averiguamos que hay dos tipos de consultas: las que están almacenadas directamente en la base de datos y se ejecutan desde el código (stored procedures/command type) o las que están escritas directamente en el código y se ejecutan desde el mismo código (command type). Aquí es donde tuvimos que tomar la primer decisión, cuál de los dos tipos elegir. Tras ver que era más cómodo y que agregado a esto, se debía entregar un archivo SQL que se encargue de preparar todo el entorno de la base de datos (tablas, consultas, disparadores), decidimos volcarnos hacia el lado de las consultas almacenadas en la base de datos.

Una vez que ya sabíamos cómo y dónde debíamos almacenarlas, tuvimos que empezar a desarrollarlas. No tomamos ninguna decisión particular para hacer esto (todas al principio, todas al final, etc.), sino que a medida que cada miembro del grupo encontraba que era necesario crear una nueva query, la agregaba al script inicial en su posición adecuada para que nada realizado anteriormente deje de funcionar.

Generalmente, todas las tablas cuentan con las consultas básicas: SELECT (las denominamos con el nombre: '**get…**'), INSERT (denominamos: '**insert…**'), UPDATE (denominamos: '**update…**') y DELETE (denominamos: '**delete…**'). A su vez, las consultas más complejas que obtienen datos que poseen cierta lógica particular, (SELECT), las denominamos: '**getBy…**'.

Realizamos tres tipos de ejecuciones de consultas desde el sistema:

- Las que realizan inserción/modificación/baja de registros de la base de datos y retornan la cantidad de registros afectados, que se ejecutan mediante un '**ExecuteNonQuery**'.
- Las que realizan inserción/modificación/baja de registros de la base de datos y retornan la primera columna de la primera fila del conjunto de resultados devuelto por la consulta, que se ejecutan mediante un '**ExecuteScalar**'.
- Las que únicamente obtienen datos de la base de datos, que se ejecutan mediante un '**ExecuteReader**'.

Triggers

Como dijimos anteriormente en el punto '**Consideraciones generales**', realizamos un trigger (disparador) '***TL_Rol_After_Update***' que se ejecuta luego de la modificación (AFTER UPDATE) de la tabla TL_Rol. Como planteaba la consigna, cuando se realizaba la baja (lógica) de un rol a continuación se le debía eliminar dicho rol a los usuarios que lo posean asignado. Dicha información, la poseemos en una tabla '***TL_Usuario_Rol***' donde se relacionan a los usuarios con los roles asignados (a través de su ID en las tablas correspondientes). Para poder realizarle la baja del rol a los usuarios que lo poseían asignado, decidimos crear un disparador que se fijara si el nuevo valor del campo '**Activo**' del rol modificado fue seteado en false. En dicho caso, realizara la baja física de todos los registros de la tabla '***TL_Usuario_Rol***' cuyo campo '**ID_Rol**' coincida con el del rol modificado.

Por otro lado, en la consigna se planteaba como requisito funcional, que a medida que un usuario decida realizar una compra de una publicación (de cualquier tipo) y califique a dicho vendedor (tanto empresa como cliente), la puntuación que dicho usuario otorgó afecte a la reputación del vendedor. Para esto, en primer instancia decidimos crear un campo '**Reputación**' en la tabla '***TL_Usuario***', cuyo valor por defecto es 0. Y luego, tuvimos que crear un trigger (disparador) que se ejecute luego de la inserción (AFTER INSERT) de un registro en la tabla '**TL_Calificacion**'. Al producirse este evento, se actualiza el campo '**Reputación**' del registro del vendedor en la tabal '***TL_Usuario***', seteandole el promedio (AVG) de las distintas calificaciones recibidas debido a las compras de sus publicaciones.

Índices

A la hora de decidir la creación de nuevos índices, nos dimos cuenta que bastaba con los generados propiamente por el motor de la base de datos cuando decidimos agregar la constraint de IDENTITY (identidad) a un determinado campo durante la creación de las tablas. Es por esto que decidimos no crear ningún índice aparte.