Estrategia
Grupo: “GEDDES”
Número: 38

Curso: K3021

Integrantes:

- Beraha, Ariel - 144.290-9
- Campassi, Rodrigo - 154.073-7
- Esposito, Lucas - 152.679-0
- Tolaba, Emiliano - 152.413-6

Índice / Etapas del desarrollo:

1. Comprensión del enunciado

El trabajo práctico lo comenzamos reuniéndonos a comprender el enunciado y despejar dudas. En más de una ocasión, debatimos acerca de que había que hacer y cuál era la mejor forma de hacerlo, teniendo en cuenta las buenas prácticas vistas en la materia y a la vez la performance.
En esta etapa logramos comprender el enunciado y modelar una solución, la cual modelamos en un diagrama entidad-relación y tomamos como referencia para la construcción de la aplicación.
En esta etapa fue muy útil conversar acerca de las distintas alternativas y leer los mails del grupo del trabajo práctico, para poder conocer las dudas más comunes y evitar que algunas de ellas surjan luego de haber construido gran parte de la aplicación, cosa que podría forzarnos a cambiar el modelo o la implementación debido a no haber considerado algo en su debido momento.
En este punto, creemos que la metodología empleada nos evitó problemas de último momento, cumpliendo con su objetivo.

2. Diseño de la capa de datos

Luego hicimos un diseño preliminar del modelo de datos.

Una de las decisiones más importantes que tomamos se relaciona al modelado de los Usuarios, Afiliados y Profesionales. Como estos últimos dos presentaban información en común como el nombre, apellido, mail, dirección, etc. decidimos agrupar estos datos, para no tener información repetida, en la entidad Usuarios, y que las otras dos entidades referencien a ésta mediante una foreign key.

En cuanto lo referente al modelado de los Usuarios, además de los datos referente al nombre, apellido y la información básica, contienen la información para acceder al sistema desde la aplicación, como ser la contraseña, el nombre de usuario y la cantidad de intentos. Este último es utilizado para que, al intentar acceder, se reduzca la cantidad de intentos según el usuario haya ingresado la contraseña correctamente o no.

Ademas, decidimos que la Primary key de esta entidad sea el número de documento, debido a que nos facilitó la migración de la base de datos al ser este numero unívoco en el contexto del problema.

Algunos usuarios de prueba que agregamos:

username	password	rol	Número Afiliado	Plan	Número Profesional	usua_id
admin	w23e	Todos	555101	555558	9999	0
afiliado	afi	Afiliado	101	555556	-	756429041
profesional	prof	Profesional	-	-	1465925	146592501

La relación entre el Usuario y sus roles la establecimos mediante una tabla intermedia, *'RolXUsuario'*, debido a que un Usuario podía tener muchos roles y , a su vez, un rol podría estar disponible para todos los usuarios

Del mismo modo, establecimos la misma relación entre los Roles y las Funcionalidades: mediante una tabla intermedia *'FuncionalidadXRol'*

El modelado de la entidad Profesionales contiene, como explicamos anteriormente, una referencia a su respectivo usuario, además de datos como la matrícula.

Para relacionar una Especialidad con un Profesional, establecimos una relación EspecialidadXProf, debido a que un Profesional puede tener más de una Especialidad, y viceversa

La agenda de un Profesional se modeló mediante la entidad Horarios, la cual contenía la fecha y el horario de inicio de la atención. Esta, a su vez, contiene referencias a la entidad EspecialidadXProf ya que un horario en particular puede variar respecto al profesional y a la especialidad.

El modelado de la entidad de Afiliados, contiene, al igual que Profesionales, la referencia a su código de usuario correspondiente, los datos del Plan, estado civil y cantidad de hijos.

En cuanto a los Bonos, decidimos tener dos entidades distintas: una denominada Bonos, con el código de afiliado que lo compró, el plan sobre el cual se compró, el número de consulta y el afiliado que lo usó (estos dos últimos campos se establecen al momento de llegar a una consulta) y otra denominada ComprasBonos que contiene información el afiliado que la compró, la cantidad y el precio final.

Separamos la información del bono en dos entidades porque consideramos que, a efectos de realizar una consulta sobre el precio de compra de un bono por parte de un afiliado en un momento dado, debe permanecer fijo sabiendo que el precio del de compra de un plan en particular puede variar.

Al finalizar volvimos a revisar la consigna para no pasar nada por alto antes de la primer entrega.

3. Desarrollo del script de inicialización

Luego de finalizar por completo el modelo de datos, nos reunimos nuevamente para desarrollar el script de migración. En un principio pusimos foco en realizar todo el esqueleto del script más que en su correcto o completo funcionamiento ya que al tener todo codificado nos permitió ver el dominio entero y visualizar las dependencias causadas por las foreign keys.

Comenzamos creando los drop table de las tablas que ya sabiamos que ibamos a utilizar por 2 motivos: el primero es que sabiamos que tenia que estar en la primer sección del script ya que si se volvía a ejecutar el script las tablas ya iban a estar creadas y también nos facilitaba ya tenerlo codificado para testear de forma más rápida los create. Aunque a medida que creábamos las tablas también debíamos cambiar el orden de los drop, dejando en primer lugar la tabla “sin hijas”.

El siguiente paso fue codificar los create table y decidiendo los tipos de dato en forma más específica. A medida que probábamos que funcionara, íbamos cambiando el orden de las sentencias para que las tablas que llevaban una FK de otra tabla existiese antes de crearla.

Una vez que estaban todas las tablas creadas continuamos por codificar los insert de migración y asi ya podiamos testear su correcto funcionamiento. En este paso no tuvimos que tomar muchas decisiones ya que dependía directamente de la estructura de las tablas que ya habíamos decidido.

Una vez completo este paso si nos dedicamos a corregir todos los errores que habíamos cometido. Prestar atención no sólo a como corregir el error sino a que es lo que lo originó nos fue de suma utilidad para no repetir las mismas equivocaciones durante el resto del desarrollo del trabajo.

4. Desarrollo de estructura general y común (login, elección roles y funcionalidades)

Al terminarlo, consideramos que debíamos desarrollar entre todos la estructura general del proyecto en C#, ya que atravesaba todo el proyecto. Esto incluye los directorios, el login, la elección de roles y de funcionalidades.

Teníamos en claro teníamos que tener 3 formularios. El primero que se ejecutaría sería el Login, y según si el usuario ingresado tenía roles o no, continuar hacia el formulario de Elección Roles o Menu Principal (donde se elige la funcionalidad deseada). Esta decisión es para ahorrar un click al usuario si no tiene más de un rol.

Creamos los formularios e hicimos un rapido analisis de los elementos que debía que tener cada uno, si luego surgian otras necesidades los agregabamos.

En el desarrollo de los .cs nos dimos cuenta que para no repetir lógica en la conexión a la base de datos la mejor opción era delegar y crear una clase abstracta tenga la lógica de la conexión a la base de datos. La llamamos “DBConnection” y tiene su método abstracto: “getConnection”. Otra decisión importante es que optamos por que los parametros de conexion no esten en esa clase si no en el ConfigurationManager del proyecto, declarando 3 parametros: server, user y password.

Cuando empezamos a hacer las consultas a la base de datos para consultar el usuario y los roles nos pareció importante crear una clase abstracta que tenga varios métodos que sabíamos que los íbamos a necesitar en varios lugares para evitar la repetición de lógica. A esta la llamamos “Utils.cs” y colocamos métodos que servían por ejemplo para llenar un listBox o un comboBox, crear un stored procedure, o algunas consultas generales por ejemplo: “getNumeroAfiliadoDesdeUsuario”, entre otros.

Para traer la informacion de los roles disponibles para un usuario desde la base de datos utilizamos un procedure ‘getRolesUsuario’ y al seleccionar uno de los roles se llama a otro procedure llamado ‘’getFuncionalidadXRol’

5. Division de tareas

Una vez completa la estructura general, nos vimos en la necesidad de dividir tareas, debido a la dificultad de coordinar horarios para realizar en equipo la totalidad del trabajo práctico. En este punto, todos comprendíamos lo que había que hacer y lo habíamos charlado en reiteradas ocasiones.
Decidimos entonces dividir las tareas en forma equitativa, pero a la vez tener comunicación frecuente para conversar todos los temas que sean necesarios con detalle.
Por otra parte, tanto durante el desarrollo de los distinto módulos como luego de su finalización, todos los integrantes hacíamos pruebas sobre los módulos del resto,

encontrando errores, validaciones no realizadas, y sugiriendo cambios no funcionales como la disposición de los botones en las distintas ventanas.
Dadas las charlas explicadas previamente para comprender el enunciado y desarrollar el trabajo, todos teníamos el conocimiento suficiente para probar los módulos de otros sin tener dudas respecto a sus funcionalidades, lo que permitió que las pruebas sean rápidas de realizar y correctas a la vez, logrando así muy buenos resultados.

6. Desarrollo de los puntos del enunciado

Ahora procederemos a tratar cada módulo en particular, indicando las decisiones de diseño tomadas y los detalles de implementación asociados a estas.

1) ABM Roles:

Observamos que dentro de esta funcionalidad se incluyen 2 aspectos:
primero el de administrar los distintos roles de un usuario y por otro lado
el abm de roles en sí, por eso implementamos un menú donde se puede acceder tanto a un usuario particular (ingresando su nombre de usuario), como a los diferentes roles del sistema.

Otras consideraciones fueron validar en el Alta y Modificación de roles que los nombres de estos fueran válidos y no existan en los roles actuales del sistema, además en la modificación de roles se lista la totalidad de roles, diferenciado los inhabilitados con la correspondiente leyenda “(inhabilitado)” y en cuyo menú se da la opción de habilitarlo a través de un checkbox.

2) Login y seguridad:

La decisión más importante que tomamos es que la cantidad de intentos realizados sea un atributo de la tabla Usuarios. La otra opción era crear una tabla que contenga una FK a usuarios y un campo Cantidad de intentos, pero consideramos que para tener un campo no valía la pena agregar otra tabla ya que implica más espacio, mayor dificultad de la consulta.

También decidimos que un usuario deshabilitado era aquel que su cantidad de intentos restantes era igual a 0, ya que crear un campo que sea “Habilitado” no tenía sentido si podíamos reutilizar la cantidad de intentos. Ademas que se podria considerar un campo calculado y como vimos en la cursada no seria correcto.

Utilizamos un procedure llamado Login_procedure para validar al usuario y en ese mismo setear los cambios de los intentos realizados.

Parte de esta etapa fue descrita en el punto 4 de la Estrategia.

3) Cancelar atención médica:

En este punto optamos por armar una única pantalla tanto para afiliados como profesionales en cuanto al diseño interno. Desde el punto de vista del usuario, la pantalla se ve diferente según se trate de un afiliado o de un profesional.

El afiliado ve una lista de turnos con los datos necesarios, sobre la que selecciona el que desea cancelar. El profesional, en cambio, ve los turnos que tiene asignados y los datos necesarios, pero cancela un día completo o un periodo de tiempo determinado. En este punto, según que opción se elija (día o periodo) se habilitan o deshabilitan los datepickers correspondientes, evitando que el usuario deba ir a otra ventana.

Los administradores, dado que no se encontraba explícitamente especificado, decidimos considerarlos como profesionales. Es decir que si un usuario es administrador, al entrar como administrador ve lo que vería como profesional.

Para cancelar los turnos fueron necesarios:

- `Cancelar_turno_afiliado`: Stored procedure que dado un turno, se encarga de eliminarlo.

- Cancelar_dia_agenda: Stored procedure que se encarga de cancelar todos los turnos asociados a un profesional para un cierto día de su agenda.

4) ABM Afiliados

Para la realización de ésta funcionalidad, fue necesaria crear la abstracción de una clase Afiliado en el modelado de objetos de la Aplicación, a fines de facilitar la administración y verificación de los datos dentro de ésta y no agrupar y pasar muchos datos por parámetro. Por este motivo, dicha clase no posee comportamiento.

Una vez que se selecciona la funcionalidad ABM Afiliados en el menú principal, se muestra una ventana que ofrece al usuario la selección de cada una de las tres opciones (- Alta, -Baja, -Modificación) y según lo elegido se dispone de la pantalla que resuelve las funcionalidades requeridas, a saber:

a) <u>Alta</u>

Se dispone de un formulario que contiene cuadros de texto donde el usuario escribe sus datos principales, como el Nombre, Apellido, Tipo y número de Documento, Dirección, Teléfono, Su Estado Civil, el Plan y la cantidad de hijos que tiene a cargo.

Cuando el usuario confirma la operación, se validan los datos. En caso de ser inválidos se le notifica pidiendo que los ingrese correctamente. Luego de haber superado la validación de los datos se procede al registro del Afiliado en el sistema.

Si bien no se requería que exista un registro del Usuario en el sistema por parte de la aplicación, fue necesaria realizarla debido a que el modelado de datos agrupaba la

información relacionada con el Nombre, Apellido, DNI, Dirección y Mail en la entidad Usuarios.

En esta instancia se verifica que el usuario exista en el sistema, y que las contraseñas sean correctas (se pide que la confirme). De ser correctos los datos, se procede a registrar primero al usuario y luego al correspondiente afiliado.

Si éste posee familiares a cargo o cónyuges, se le ofrece la posibilidad de registrar a los familiares mostrando un nuevo formulario de alta, con la generación de su respectivo usuario. El formulario de agregado es idéntico al de alta principal, por dicho motivo hereda de éste último, cambiando únicamente su comportamiento en ciertas secciones como por ejemplo, no agregar otro familiar al familiar que se está agregando, ya que solo se deberían agregar familiares afiliados del principal.

Para el registro de ambas entidades, fue necesaria la creación de tres Store Procedure para el registro del Usuario, Afiliado y Familiar respectivamente:

- `ingresarUsuario` : realiza un insert en la tabla usuario con los datos de username, contraseña. Nombre, apellido, tipo y número de documento, teléfono, fecha de nacimiento, sexo y mail .
- `ingresarAfiliado :` con la clave generada por el procedimiento anterior, inserta los valores de plan, estado civil y cantidad de hijos a cargo.
- `agregarFamiliar :` dado el número de afiliado raíz, calcula el próximo número de afiliado, lo suma al raíz e inserta los valores(previa inserción de datos en la tabla Usuarios).
- `verificarUsuario`: ante el evento de inserción de una nueva fila en la tabla Usuarios,se procede a consultar si el nombre de usuario existe en el sistema. De ser así, tira una excepción para que pueda ser notificada en la aplicación.

b) <u>Modificación</u>

Previo al acceso a esta funcionalidad, se muestra al usuario una pantalla donde debe seleccionar el afiliado que desea modificar.
A efectos de poder realizar una consulta dinámica según los filtros que se desean aplicar, se utiliza una clase Parser que se encarga de armar según los parámetros provistos por los TextBox, CheckBox y su respectivo. Sin embargo, debido a la complejidad que conlleva su realización, solo fue efectuada para consultas que relacionen a los Afiliados con los Usuarios.

Una vez seleccionado el afiliado, se muestra el formulario de Modificación correspondiente.
Dicho formulario *se hereda del formulario principal de alta* por motivos de reutilización de código, ya que poseen comportamiento similar en ciertas ocasiones (es por dicho motivo que no se puede ver la vista previa del formulario).
El formulario de Modificación tiene los mismos campos de texto que el formulario anterior, con la diferencia que se muestran los datos principales como el Nombre, el Apellido y el DNI, sin posibilidad de modificarlos.

Los únicos campos disponibles para ser completados son, la Dirección, el Número de Teléfono, el el Plan, la cantidad de hijos a cargo, y el estado civil.

Una vez ingresados los datos (no todos deben estar completos a diferencia del formulario de Alta) , se procede a la modificación de los datos en el sistema previa verificación de datos.

Antes de proceder, si el usuario ha decidido cambiar el plan, otro formulario pide que se ingrese el motivo por el cual se realiza dicha modificación, para luego registrarla en el sistema, en la tabla HistorialAfiliado.

En éste caso, como no se especificó en la consigna sobre ofrecer la posibilidad de agregar al cónyuge o a las personas a cargo, asumimos que ésta operación sólo se da en el alta de afiliado.

Al igual que en la funcionalidad descrita anteriormente, se utilizó un Store Procedure para proceder a la modificación de los datos del Afiliado y su Usuario correspondiente:

- `modificarAfiliado` : dado un número de usuario, calcula su respectivo afiliado asociado e inserta los valores de dirección y teléfono en la tabla Usuarios y los valores de plan y cantidad de hijos en la tabla Afiliados.

c) Baja

Para el desarrollo de esta funcionalidad, no se requirió de un formulario ni clase adicional, debido a que su comportamiento podría agruparse en dos botones.
Tanto la baja lógica como la física son operaciones que se ofrecen en el formulario que lista a los afiliados del sistema. La opción Deshabilitar da de baja al afiliado estableciendo el valor 0 en el campo de los intentos del Usuario asociado, mientras que la opción de Eliminar borra los registros asociados al Afiliado y su respectivo Usuario del sistema.
Al no aclararse el caso de la eliminación de un afiliado raíz, decidimos que si se elimina al mismo, el resto no será eliminado del sistema.

En el caso de la baja lógica utilizamos el siguiente store procedure:

- `darDeBaja` que dado un id de usuario, actualiza la cantidad de intentos a 0

Para que la eliminación física se pueda llevar a cabo, se utilizó un Trigger que, ante dicho evento en la Tabla afilados, realice las eliminaciones en las otras tablas que tienen referencias a ella con el fin de evitar la inconsistencia de los datos.

- `eliminarAfiliado` : dado el user_id del afiliado, calcula su clave de afilado correspondiente, y elimina los registros correspondientes de las Tablas que tienen referencias a la fila que se quiere eliminar en el siguiente orden:
  HistorialAfiliado -> ComprasBonos ->Bonos -> Turnos -> Afiliados -> Usuarios

- `triggerElimTurnos`: llegado el punto de querer eliminar un registro de un turno, primero borra los registros de las tablas CancelacionesTurnos y Consultas para luego borrar los registros de la tabla de Turnos correspondientes

- `triggerEliminarUser`: Ante un evento de eliminación del registro de la tabla de Usuarios, este trigger se encarga de eliminar los registros asociados a el en la tabla RolXUsuario

8) Registrar agenda del profesional:

En este punto consideramos que los profesionales sólo podían cargarse horarios a sí mismos, pese a que el enunciado no lo decía explícitamente. Los administradores, en cambio, como sería esperable, pueden cargarle turnos a cualquier profesional.

También consideramos que si habían menos de 48 horas cargadas para un profesional en una semana, y se intentaban agregar más horas, el sistema debía agregar horarios a la agenda hasta que se alcancen las 48 horas, y recién entonces dejar de insertar horas, informando al usuario en forma clara y concisa lo que acababa de suceder.

En este punto del trabajo práctico notamos que el input del usuario difería en gran manera respecto a los datos que debían insertarse en la base de datos. Tomamos la decisión de mantener la pantalla lo más simple posible, priorizando la facilidad y comodidad en el uso de la aplicación. Para esto, según el día de la semana, los horarios de inicio y finalización de la atención, y el profesional y especialidad seleccionados, se generan turnos consecutivos con una duración de media hora cada uno entre las fechas de inicio y fin de validez de dichos horarios.

Para hacer las validaciones fue necesario crear un trigger:

- LimiteHoras: Trigger que se activa si el límite de horas del profesional en alguna semana superaría las 48 de cargarse los horarios pedidos. Si esto sucede, da error, en caso contrario, inserta normalmente. Este tirgger también chequea que no se quieran agregar horarios a la agenda de un médico si ya los tiene.

El resto de las validaciones no requerían información de la base de datos, por lo que se manejan directamente en la aplicación.

9) Compra de bonos:

El primer punto a considerar fue que el enunciado detalla que a esta funcionalidad pueden acceder tanto los afiliados como un administrador. Como medida de seguridad nos pareció necesario que si entra un afiliado el campo que describe el número afiliado sea calculado directamente y el atributo ReadOnly seteado en true (para que no pueda comprar un bono para otro afiliado). En cambio un administrador si tiene que poder escribir el número de afiliado con el que va a operar. Por cuestión de diseño, se agregó la funcionalidad de que se autocomplete el número de afiliado.

Al confirmar una compra de los bonos se realiza primero un for desde 0 hasta la cantidad de bonos seleccionados para comprar, donde se hace un insert en la tabla Bonos. A continuación se hace un insert en la tabla ComprasBonos registrando la compra que hizo el usuario con fines estadísticos o de auditoría.

Finalmente, las medidas de seguridad que había que tomar en cuenta era que al momento de confirmar, el afiliado sea válido y no este deshabilitado, y verificar que este bien seteado el plan correspondiente

10) Pedido de turnos:

Lo primero que observamos fue la necesidad de 2 formularios. El primero que tenia la responsabilidad de mostrar todos los profesionales y dar la opción de filtrar por especialidad.

Al seleccionar un profesional se abre el otro formulario, donde se busca la agenda profesional del profesional. Como medida de seguridad, al confirmar un turno se actualiza la lista de horarios para que no pueda ser reservado múltiples veces si es que no sale de esa pantalla.

11) Registro de llegada para atención médica

Dicha funcionalidad solo puede ser utilizada por el administrativo. Al ser iniciada, se muestra una lista de los profesionales que existen en el sistema. Una vez seleccionado el profesional, se procede a la selección del turno mostrando una lista desplegables con los turnos que el profesional tenía para ese dia y los respectivos afiliados.

Luego de seleccionar el afiliado, se abre una nueva ventana que permite escoger el bono ocupado. Finalmente, se procede a generar la consulta y registrar el bono como ocupado insertando el número de afiliado en la fila ingresada de Consulta correspondiente.

Para realizar la persistencia de los datos de la llegada a la atención médica, se utilizó un store procedure

- `registrarConsulta` : dados un número de turno, bono, y afilado, prodece a insertar los valores del turno y la fecha actual del sistema en la tabla Consultas, luego se introduce en la tabla Bonos el número de consulta generado y el afiliado que hizo uso del bono.

12) Registro de resultado para atención médica

Una cosa a tener en cuenta es el hecho de que el enunciado especificaba que el administrador pueda acceder a todas las funcionalidades. Por ende, debería poder acceder a esta funcionalidad. Sin embargo, la misma debería ser usable solo con el profesional, ya que el es encargado de atender a los pacientes y registrar el resultado.

A pesar de esto consideramos y resolvimos este requerimiento como orientado al profesional.

Una vez que se ingresa, se muestra una pantalla donde se muestra información sobre las consultas que tenia en el mismo día, junto con el nombre del afiliado. Una vez seleccionado, se habilitan los campos en donde el profesional ingresa los síntomas, el diagnóstico del paciente y si la consulta fue concretada o no. Una vez llenados todos los campos, se procede al registro en el sistema.

Por cuestiones de estética y simplicidad, se decidió poner toda esta información en un solo formulario, además de que no requería mucho comportamiento.

Para persistir los datos del resultado, se utilizó un Store Procedure:

- `registrarResultadoConsulta` que dada un número de consulta, los síntomas , el diagnostico y un valor que indica si fue concretada, actualiza los valores en la tabla Consulta, que fueron insertados en el procedure `registrarConsulta` pero en el que no se llenaron los campos mencionados anteriormente

13) Listado Estadístico:

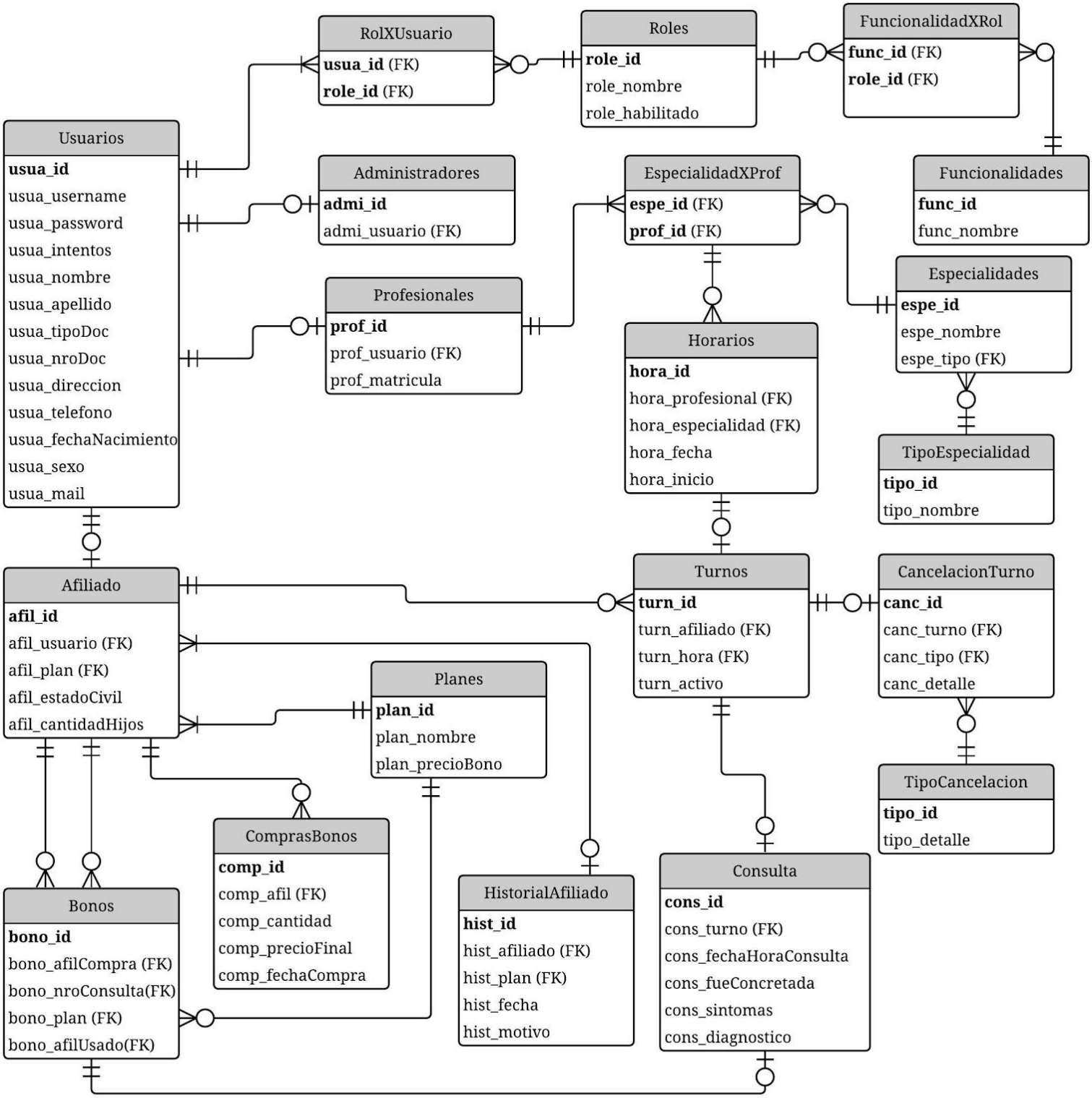
Decidimos agregar una opción general en los filtros (opción todos) para poder apreciar los totales globales de los listados.

También decidimos colocar un botón de actualización de los listados con respecto a la fecha seleccionada para no provocar una consulta por cada cambio en alguno de los parámetros, ya que pueden ser necesarios varios ajustes para completarla y evitar así tiempos de carga innecesarios.

En el Listado 4 se utilizó una función (`tieneFamilia`) para obtener si el usuario era parte de un grupo familiar y representarlo directamente como un string “SI”/”NO”.

7. Anexo: DER

Es importante aclarar que la agenda la modelamos con la entidad ‘Horarios’, ya que el nombre puede no ser lo suficientemente expresivo.



* La descripción de los triggers y procedures fueronexplicados en las funcionalidades que los utilizan, en el inciso 6.

Respecto a los índices, buscamos usarlos para aquellos atributos no indexados por los que frecuentemente hacemos más búsquedas, y las claves elegidas fueron usua_username, hora_fecha y turn_hora.