

Universidad ORT Uruguay

Facultad de Ingeniería



Diseño de aplicaciones 2

Obligatorio 1

10 de Octubre de 2018

INTEGRANTES:

Andrés Correa (176076)

GRUPO:

M6A

DOCENTES:

Eduardo Cuñarro

Ramiro Visca

Índice

Descripción general del trabajo	3
Descripción del diseño	4
Diagramas	5
Diagramas de clases	5
Paquete Data.Entities	5
Paquete Data.DataAccess	8
Paquete Logic	8
Paquete Repository	9
Paquete Web Api	10
Controllers	10
Filters	10
Diagrama de paquetes	11
Diagrama de interacción	11
Login - WebApi	11
Diagrama de componentes	12
Diagrama de entrega	13
Justificación de decisiones de diseño	14
Login	14
Filtros	14
BaseFilter	15
AutenticacionFilter	15
Fixtures	15
Modelado de la base de datos	15
Acceso a los Repositorios	15
Borrado físico de elementos	16
Manejo de errores	16
Modelo de datos	18
Instructivo de instalación	19
Resultado de la ejecución de las pruebas	22
Resultado de ejecución de las pruebas	23
Cobertura de las pruebas	25
ANEXOS	26
Justificación de Clean Code	26

Descripción general del trabajo

Para esta entrega se diseñó y desarrolló el backend de una web api rest que permite administrar y publicar información sobre encuentros para diferentes deportes.

Los usuarios de la aplicación deben loguearse para poder realizar cualquier operación ya que existen dos roles actualmente: Administrador y Seguidor.

El sistema le permite a los usuarios seguidores visualizar los equipos para seguirlos, así como poder filtrar y ordenar su búsqueda por nombre, realizar comentarios en los encuentros y mostrar los comentarios que realizaron en los encuentros de los equipos a los que sigue.

Por otra parte los usuarios administradores cuentan con la posibilidad de mantener los usuarios del sistema, los deportes, los equipos y los encuentros. También tienen la posibilidad de generar dos tipos de fixtures: liga o fase de grupos.

La solución fue implementada en lenguaje C# utilizando Visual Studio Code como herramienta de desarrollo junto con una base de datos Sql Server.

Además se utilizaron las tecnologías Entity Framework Core y ASP.Net.Core.

Funcionalidades no implementadas y errores conocidos

Quizás un defecto conocido es que la aplicación está desarrollada en un idioma híbrido español-inglés.

Descripción del diseño

En esta sección pasamos a detallar cómo decidimos construir nuestra aplicación de forma tal de delegar las responsabilidades para lograr minimizar el acoplamiento entre paquetes y facilitar la extensión y mantenimiento de nuestro sistema.

Las capas que se definieron para esto se detallan a continuación:

WebApi -> Es el punto de entrada de nuestra aplicación, contiene los controllers que exponen sus funcionalidades mediante endpoints definidos bajo el formato REST. Esta capa recibe las solicitudes y se comunica con la lógica de negocios solamente. Además cuenta con filtros que interceptan las llamadas a los controllers y validan que sean de un usuario logueado y con los permisos suficientes, así como también controlan los errores que existan con el acceso a la base de datos.

Logic -> Es el nexo entre la web api y los datos, se encarga de procesar las solicitudes recibidas a través de la web api, realizar las validaciones necesarias y en caso de ser válidas las solicitudes invoca al paquete repository que se encarga de realizar las operaciones necesarias en los DbSet.

Luego llama a la clase UnitOfWork cuya única responsabilidad es guardar los cambios luego de realizadas todas las operaciones necesarias.

Expone sus servicios mediante interfaces, la WebApi se comunica mediante ellas.

Repository -> Se encarga del acceso a datos, realiza las operaciones sobre la base de datos obteniendo, agregando, modificando o eliminando datos sin guardar los cambios. Para esto utiliza un contexto que es definido en el paquete DataAccess.

Expone sus servicios mediante interfaces, la BusinessLogic se comunica utilizandolas.

Entities -> Contiene las clases que representan las entidades de la solución

DataAccess -> Se encarga de realizar el mapeo de entidades a base de datos a través de Entity Framework, es decir donde se definen los data set del sistema.

Aquí se encuentra definida la clase UnitOfWork que se encarga de guardar los cambios en la base de datos.

Además contamos con un paquete de prueba que contiene las pruebas automáticas que creamos aplicando TDD para la lógica del sistema.

Esta descripción es a modo de introducir brevemente la división de capas, en las justificaciones del diseño y otras secciones de este documento se entra en detalle sobre cada componente explicando sus responsabilidades y funcionamiento.

La forma en cómo se comunican estas capas y las dependencias entre ellas se pueden apreciar en los diagramas que se presentan en la siguiente sección.

Diagramas

Diagramas de clases

A continuación se pueden ver los diagramas de clases de los diferentes paquetes de nuestra solución de forma tal de detallar cómo se relacionan las clases y como están compuestas.

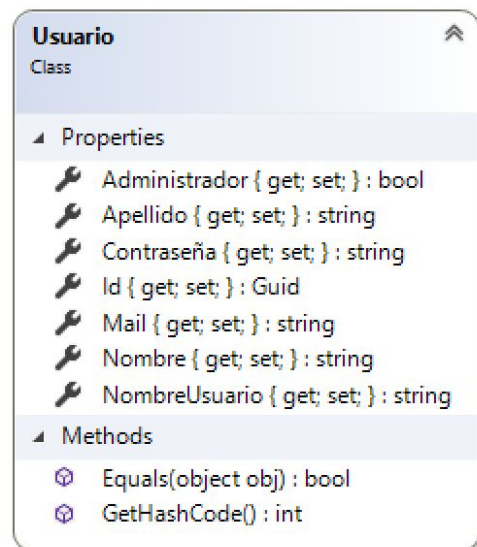
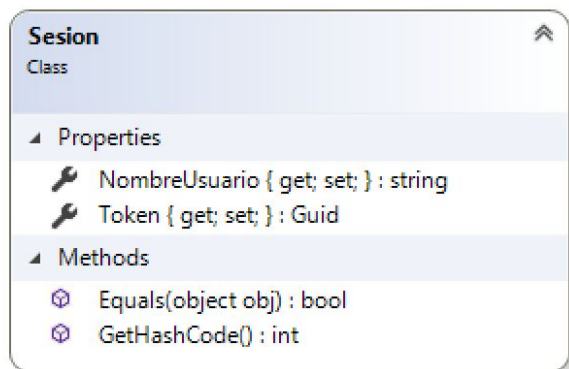
Algunos diagramas se dividieron en secciones para que se vea de forma más clara.

Paquete Data.Entities

Usuario y Sesión

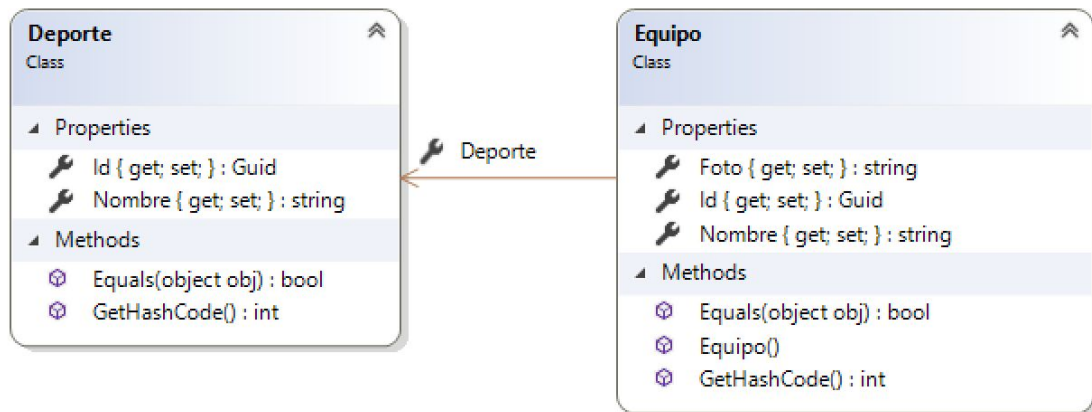
Para el manejo de los roles del usuario optamos por usar una variable booleana Administrador, en caso de que lo sea la variable es true, en caso contrario (seguidor) es false. La decisión fue tomada de acuerdo a que se especifica que no es necesaria una solución compleja de manejo de roles debido a que estos van a ser los únicos roles necesarios en nuestra aplicación. El nombre de usuario debe ser único en la aplicación.

La entidad Sesion almacena nombre de usuario y el token de la sesión,



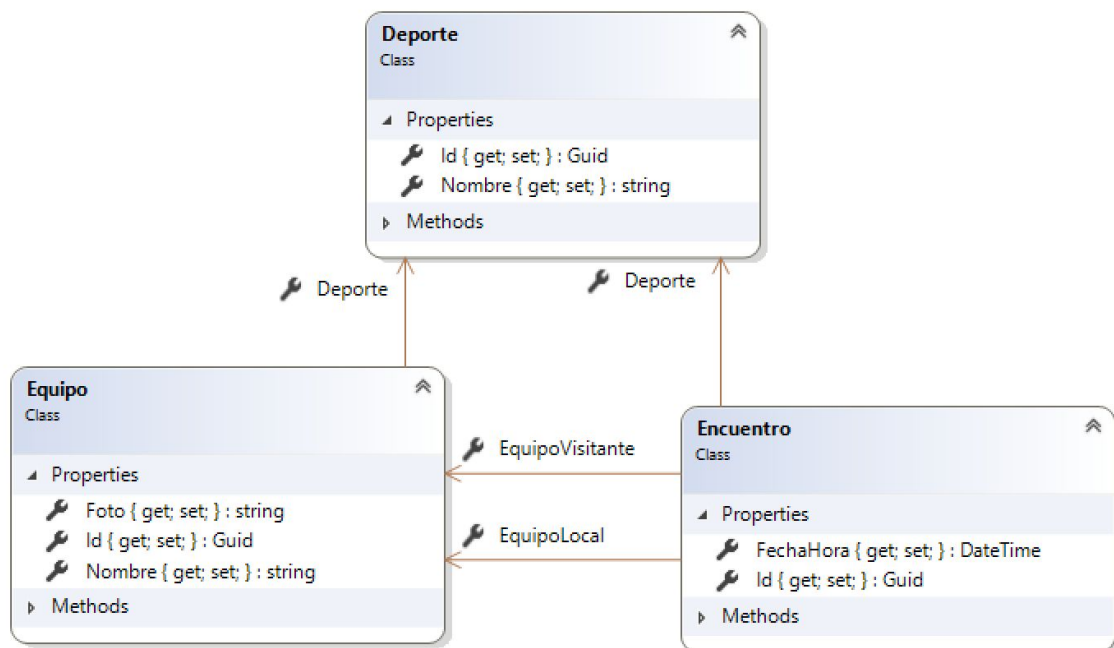
Equipo y Deporte

Definimos que cada Equipo tiene un deporte y el deporte que se le asigna al equipo no puede ser modificado una vez guardado en la base de datos.



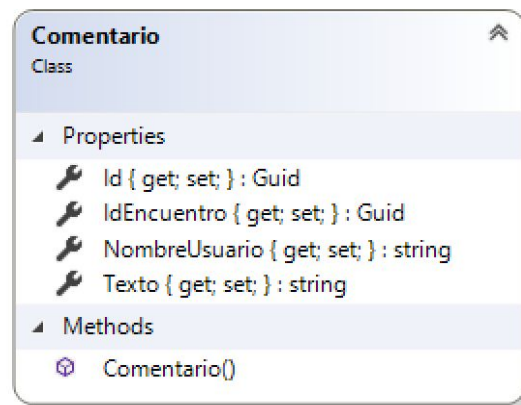
Equipo, Encuentro y Deporte

Definimos que un encuentro tiene fecha, el deporte y dos equipos(local y visitante) distintos.



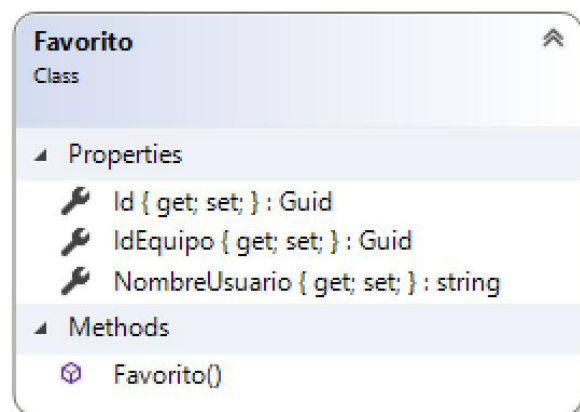
Comentario

La entidad Comentario almacena, el id del encuentro en el cual se realizó dicho comentario, el nombre del usuario que lo realizó y el texto.

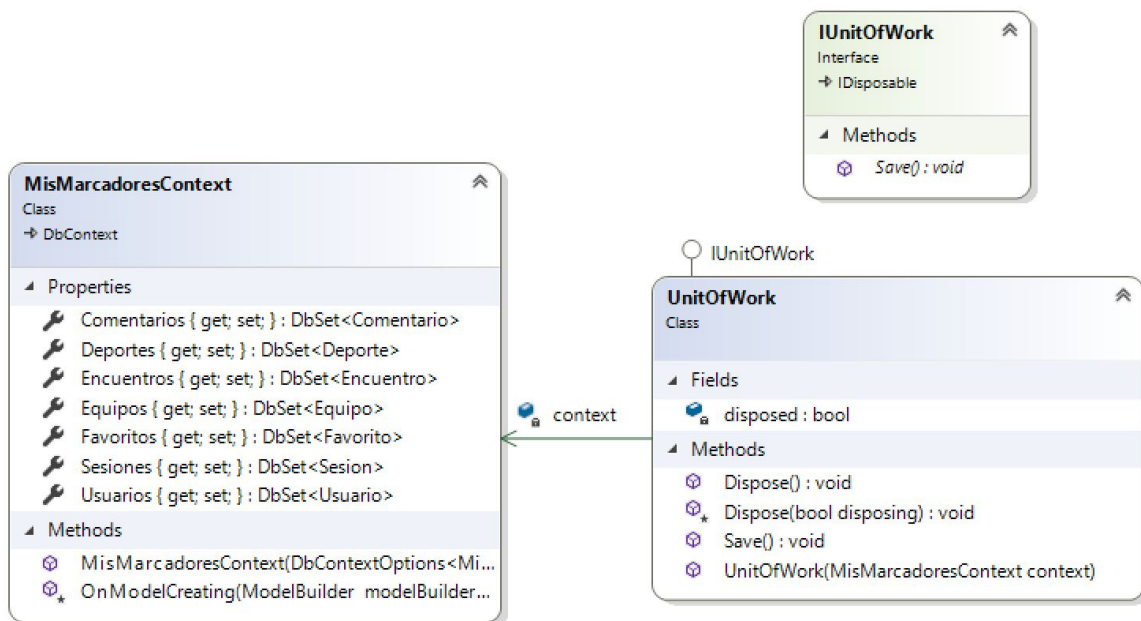


Favorito

Por último la entidad Favorito almacena para cada nombre de usuario los equipos(los id's) a los cuales está siguiendo.

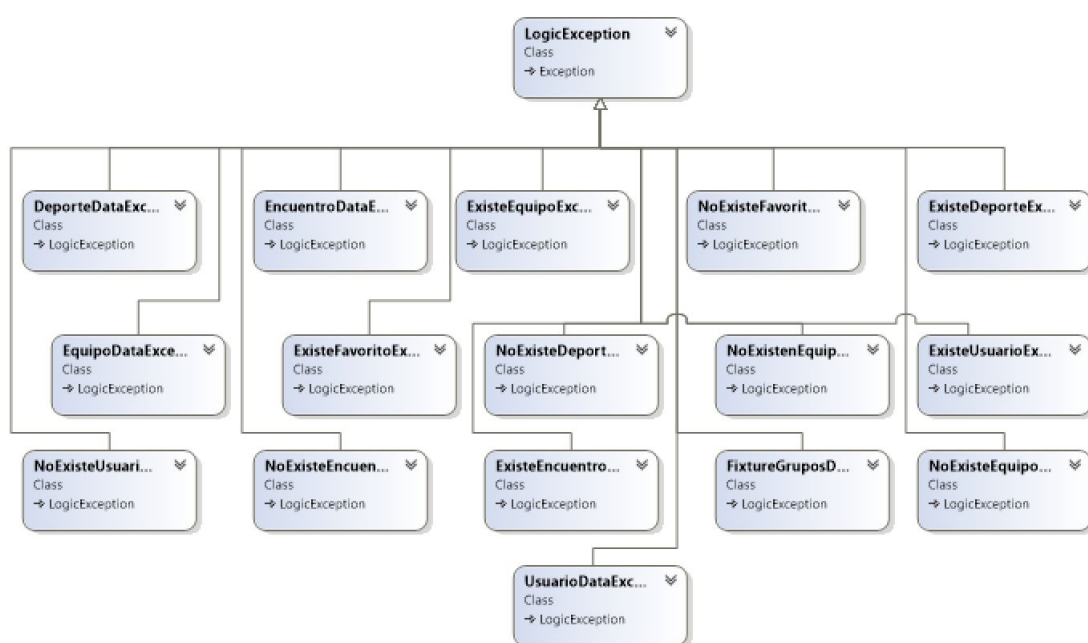


Paquete Data.DataAccess

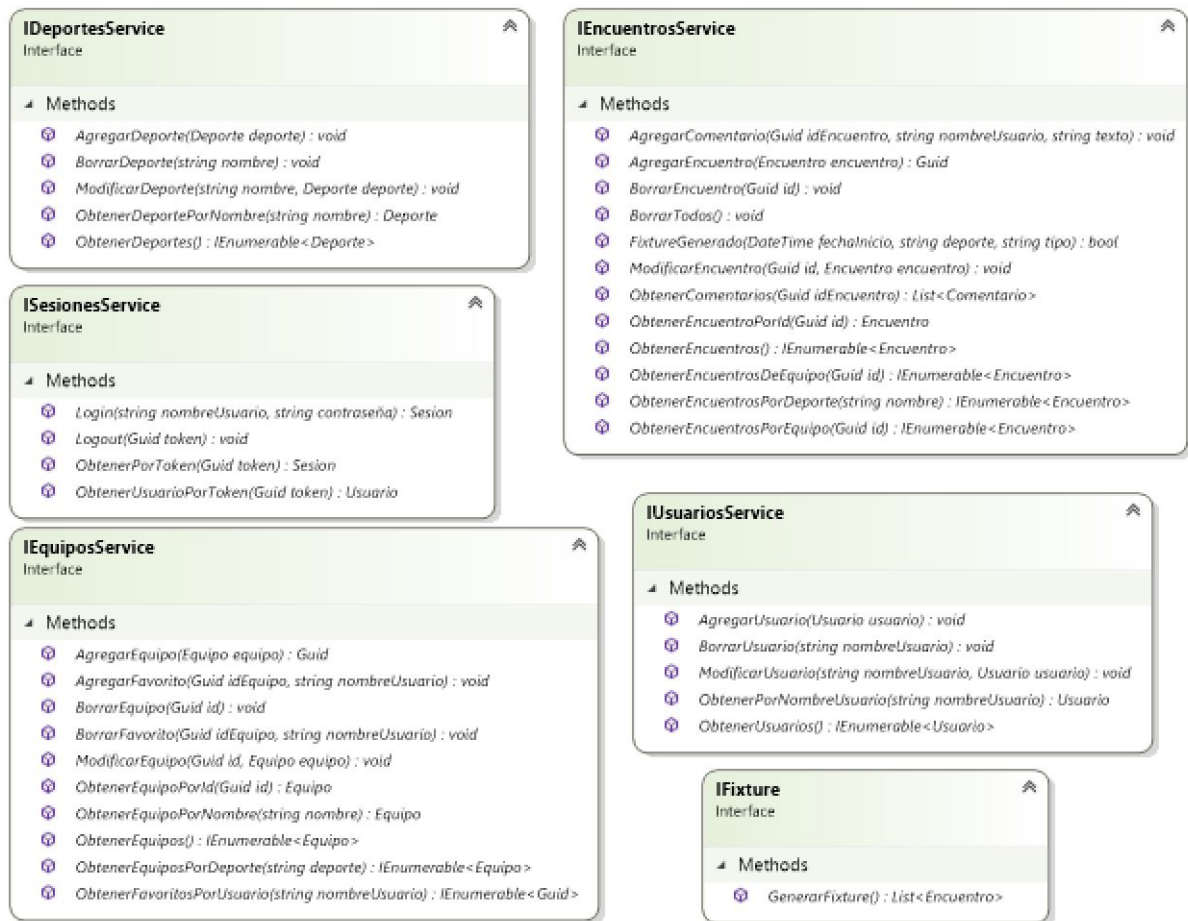


Paquete Logic

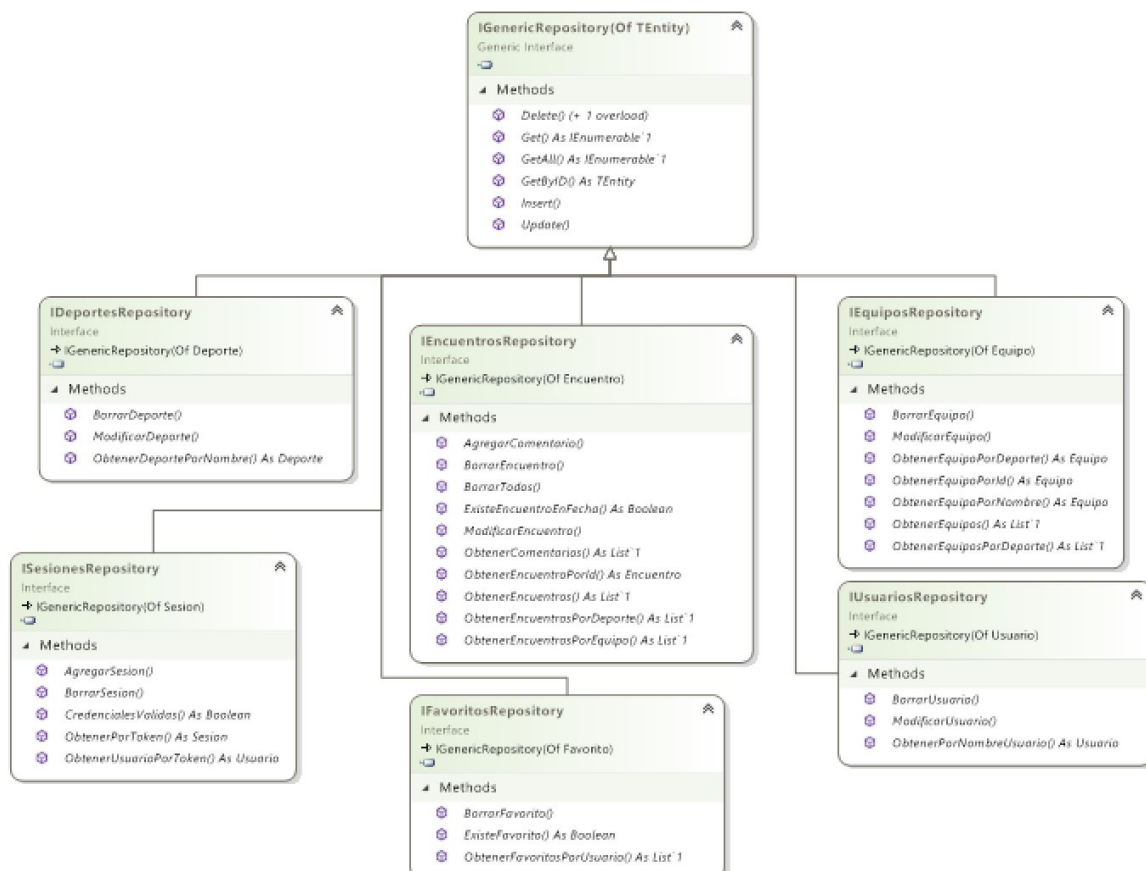
Es en el paquete de la lógica donde controlamos las excepciones, creamos una excepción llamada `LogicException` que sirve de padre al resto de las excepciones de nuestra lógica.



Las interfaces de la lógica quedan definidas de la siguiente forma:

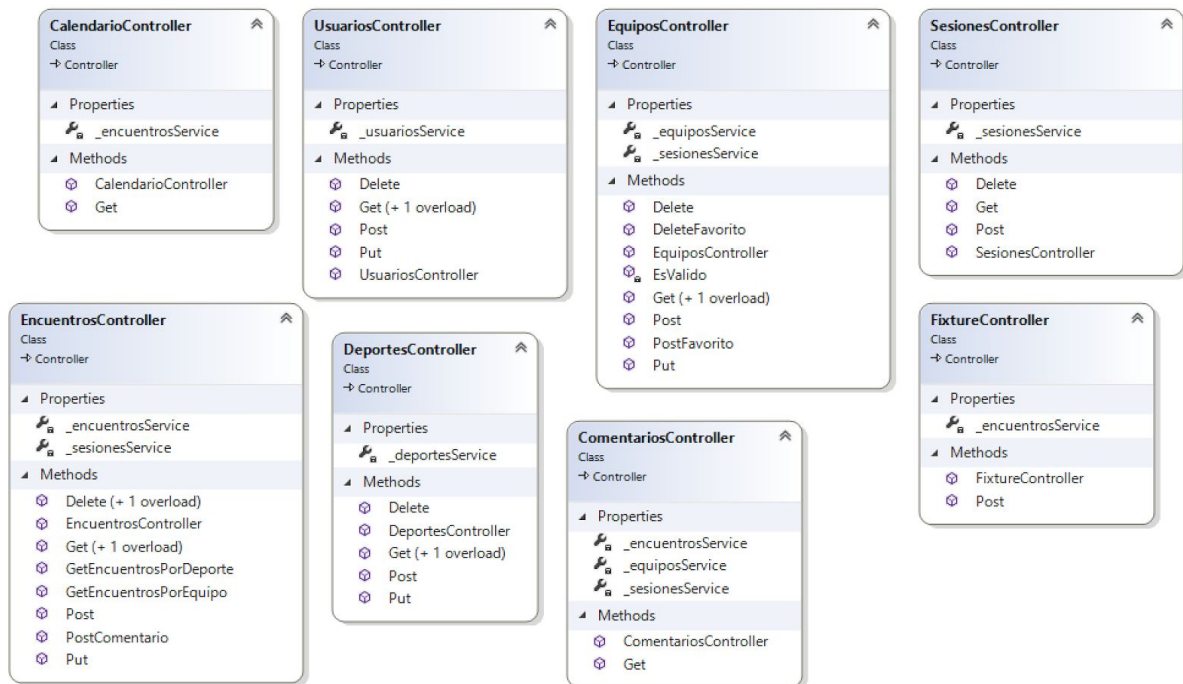


Paquete Repository

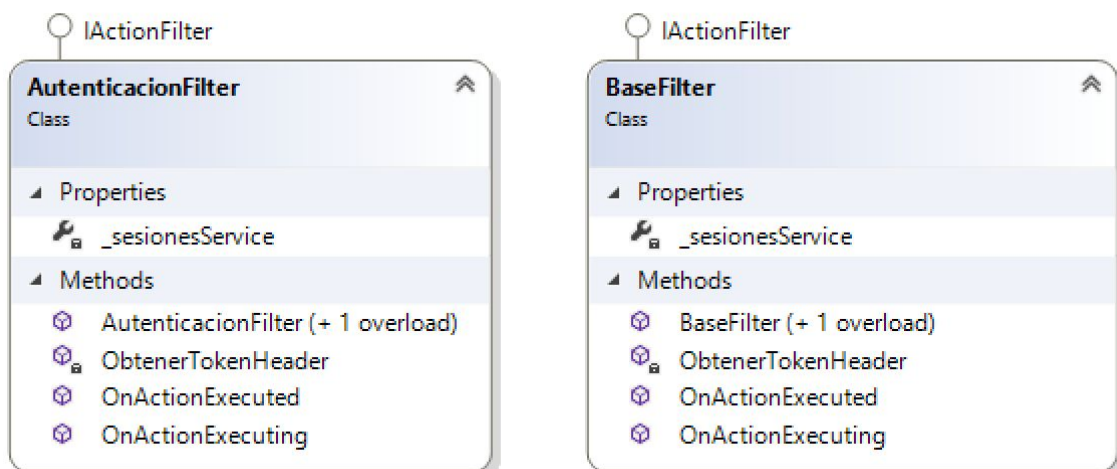


Paquete Web Api

Controllers



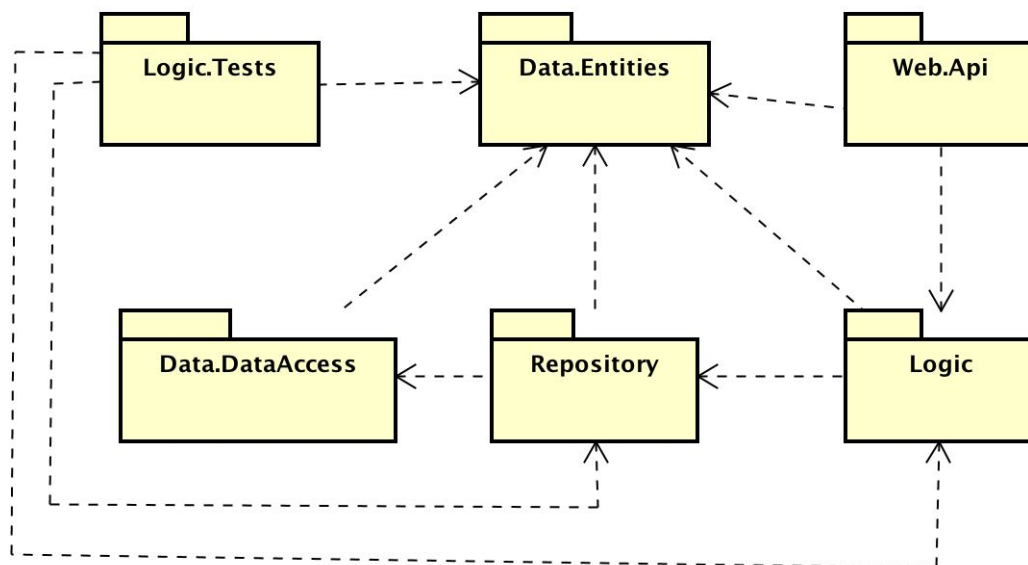
Filters



No consideramos de mucha utilidad agregar el diagrama de clases de los modelos.

Diagrama de paquetes

En este diagrama se pueden apreciar los paquetes que componen la solución de nuestro backend. Nos indica cómo son las dependencias entre las capas que definimos.



Podemos ver que el paquete que contiene las entidades tiene varias dependencias entrantes y ninguna saliente, mientras que por ejemplo en el paquete de la web api se puede ver lo contrario.

Diagrama de interacción

Login - WebApi

El siguiente diagrama muestra la interacción que sucede un usuario quiere loguearse al sistema.

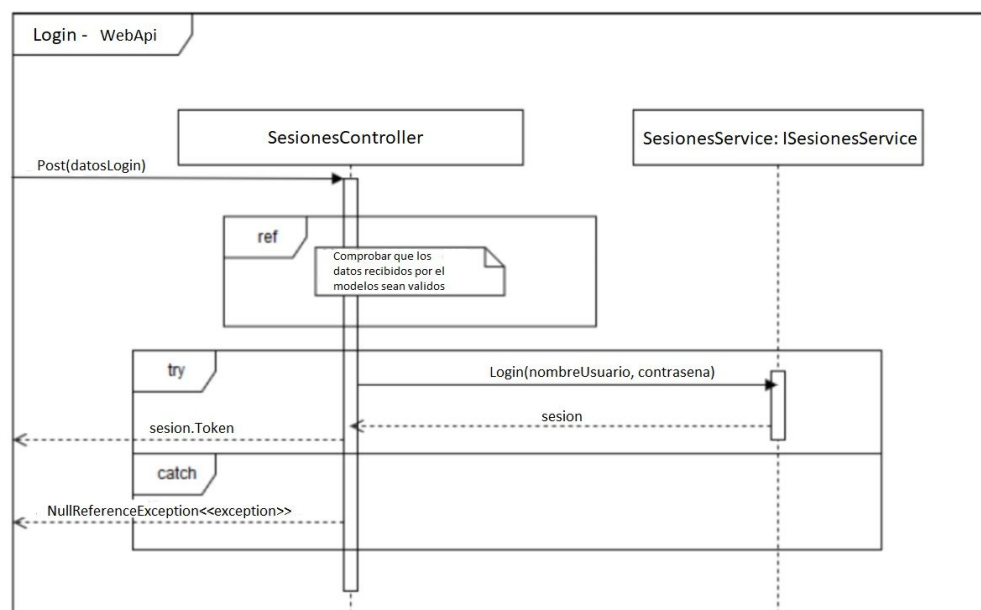
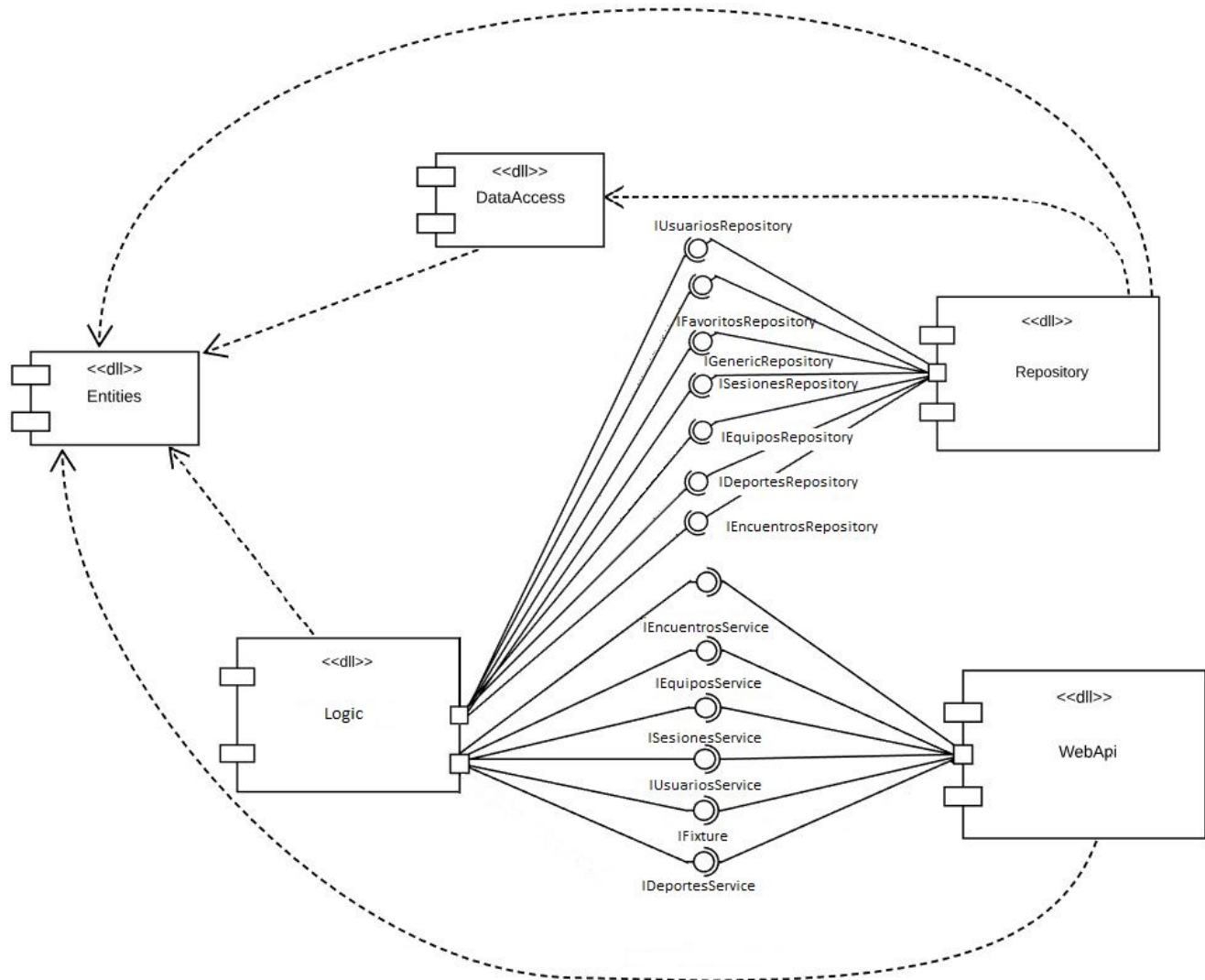


Diagrama de componentes

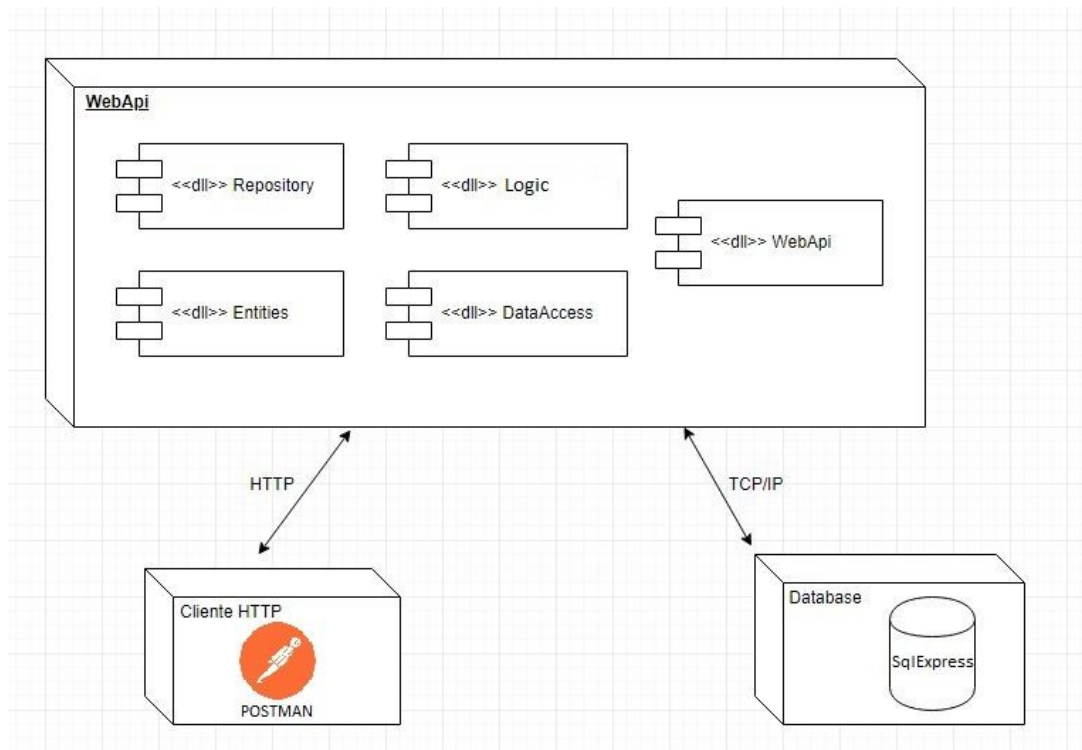
En el siguiente diagrama se puede apreciar cómo se comunican los componentes de nuestro sistema y como son las dependencias entre ellos.



Se puede ver como la comunicación entre la lógica y las capas superiores como son la WebApi y la interfaz de usuario se hace mediante interfaces al igual que la comunicación entre la lógica y el acceso a los datos mediante el componente Repository. Esto nos brinda la posibilidad de modificar las implementaciones dentro de estos componentes sin generar gran impacto sobre los demás componentes que se comunican con el.

Diagrama de entrega

El siguiente diagrama presenta el esquema en el que se comunica la API con quien consume sus funcionalidades. Se asumió que el cliente usa postman.



Justificación de decisiones de diseño

En esta sección pasamos a detallar lo que creemos fueron las decisiones más importantes de diseño tomadas para la construcción de la solución requerida.

Además se brinda una justificación de cada una de estas decisiones, explicando porque fue tomada y porque creemos que era mejor que otras.

Login

Debido a que es requerimiento del sistema que existan usuarios administradores y seguidores es que se decidió implementar un login de usuarios para así diferenciar permisos al utilizar la aplicación. Concretamente se identificaron dos roles para los usuarios del sistema: administrador y seguidor.

Estos roles no son excluyentes ya que todos los usuarios son seguidores, pudiendo ser o no administradores.

Por lo tanto la motivación para contar con login de usuarios es simplemente diferenciar entre administradores y no administradores, al menos para esta primera entrega.

La implementación es bastante simple, cada usuario al loguearse exitosamente con su nombre de usuario y contraseña recibe un token que luego debe enviar en cada request. Para esto tenemos la entidad Sesion que almacena tokens y nombres de usuarios, no creímos necesario mas informacion ya que no decidimos efectuar expiración de sesiones.

Cada vez que un usuario se loguea se crea un registro en la tabla Sesion con el token generado y el nombre del usuario. Si ya existe un registro con ese nombre de usuario solamente se actualiza el token.

Al cerrar sesion lo que se hace es borrar el registro de la tabla para ese usuario y token.

A nivel de credenciales decidimos no encriptar contraseñas ya que no es un requerimiento, por lo que se almacenan como texto plano en la tabla de usuarios.

Filtros

Decidimos aplicar filtros a los controladores, uno que se debe ejecutar para todos los requests (BaseFilter) y otro para saber si el usuario logueado es administrador o no (AutenticacionFilter).

Estos son los responsables de controlar si la request que se quiere realizar proviene de un usuario logueado y en caso de serlo si tiene los permisos para realizar la acción solicitada.

Esto evita que los controladores sean invocados en los casos que el request no sea válido o no tenga autorización.

BaseFilter

Es el filtro base, se invoca para todos los métodos de todos los controladores.

Para pasar exitosamente por este filtro el usuario debe estar logueado en el sistema (en caso contrario devuelve un error 401), debe incluir un token en el header al realizar cada request (en caso contrario devuelve error 400) y por último controla si nuestra base de datos está disponible antes de realizar la request (en caso contrario devuelve un error 500).

Es la forma que tenemos de asegurarnos que nuestra aplicación esté expuesta solamente para usuarios logueados en el sistema, así como controlar que el servidor no este caído antes de llamar a cualquier acción de cualquier controlador.

AutenticacionFilter

Este filtro lo que hace es controlar si el usuario logueado es administrador o no.

Lo invocamos en los métodos de los controladores donde solamente los usuarios administradores deben tener permisos (ABM de usuarios, equipos, deportes entre otros)

Fixtures

Para la generación de los fixtures decidimos implementar el patrón Strategy.

Definimos una clase abstracta Fixture de la cual hereda cada tipo de fixture, en nuestro caso decidimos realizar dos implementaciones: Liga (todos contra todos) y fase de grupos. En ella se define el método abstracto "GenerarFixture()", luego en cada clase concreta se hace override del método para realizar la generación de acuerdo al tipo de fixture.

Modelado de la base de datos

La base de datos se implementó usando Entity Framework ya que era un requisito del proyecto, haciendo uso del paradigma code first.

Todas nuestras entidades (salvo Sesion) tienen un identificador(Id de tipo Guid) en la base de datos, utilizamos fluent api para generarnos un Id automáticamente al guardar cualquier registro en nuestra base de datos.

En el caso de la entidad Sesion, la cual no tenemos identificador en la base de datos, usamos fluent api para asignarle como Key el nombreUsuario del usuario.

Acceso a los Repositorios

Para el acceso a los repositorios se implemento el patron Generic Repository, una interfaz genérica con las cuales se cubre el CRUD de las entidades.

Sí bien usamos nuestra implementación del patrón genérico, hay casos en los cuales cada repositorio debe realizar su propia implementación.

Por ejemplo el borrado de usuario debe también borrar los favoritos de dicho usuario y los comentarios que realizó en los encuentros.

Borrado físico de elementos

Decidimos implementar borrado físico de todos los elementos de nuestra aplicación, por ejemplo al borrar un deporte del sistema, también se borran físicamente todos los encuentros y los equipos en los cuales dicho deporte aparece.

Se optó por esta opción ya que creemos que un borrado lógico hubiera aumentado la complejidad de la solución.

Manejo de errores

Como explicamos anteriormente en el paquete de la lógica controlamos las excepciones, creamos una excepción llamada LogicException que sirve de padre al resto de las excepciones de nuestra lógica.

Luego en la web api las capturamos y devolvemos un mensaje de error con su correspondiente código de error.

Definición de endpoints

En esta sección se detallan los endpoints que disponemos en nuestra WebApi, es decir la forma de interactuar con ella.

Se realizó una división de estos endpoints en diferentes controllers, cada controller está asociado a un recurso o a recursos que están fuertemente ligados.

Usuarios

GET api/usuarios/

GET api/usuarios/nombreUsuario

POST api/usuarios/

PUT api/usuarios/idUsuario

DELETE api/usuarios/idUsuario

Sesiones

GET api/sesiones/nombreUsuario

POST api/sesiones/

DELETE api/sesiones/token

Deportes

GET api/deportes/

GET api/deportes/nombreDeporte

POST api/deportes/

PUT api/deportes/nombreDeporte

DELETE api/deportes/nombreDeporte

Equipos

GET api/equipos/

GET api/equipos/idEquipo

GET api/equipos/idEquipo?orden=""&filtro=""

Al obtener equipos también se pueden realizar dos acciones: ordenar y filtrar

En caso de querer ordenar se debe indicar qué tipo de orden se quiere obtener: ascendente o descendente, para el caso de orden ascendente debe ser de la forma orden="ASC", para el caso descendente orden="DESC".

En caso de querer filtrar se debe poner cualquier letra o palabra que el usuario desee filtrar.

Cada uno de estos parámetros es independiente del otro, es decir que el usuario puede decidir solamente ordenar, solamente filtrar, o ambos.

POST api/equipos/

PUT api/equipos/idEquipo

DELETE api/equipos/idEquipo

Encuentros

GET api/encuentros/

GET api/encuentros/idEncuentro

POST api/encuentros/

PUT api/encuentros/idEncuentro

DELETE api/encuentros/idEncuentro

Fixtures

POST api/fixture/

Dentro del body se debe mandar: la fecha de inicio, el deporte y el tipo de fixture que se desea(actualmente acepta dos valores: Liga y Grupos)

Favoritos

POST api/equipos/{idEquipo}/follow

DELETE api/equipos/{idEquipo}/unfollow

Comentarios

GET api/comentarios/

Lista de todos los comentarios de los encuentros de todos los equipos que el usuario está siguiendo

POST api/encuentros/idEncuentro/comentario

Calendario

GET api/calendario?mes=""&año=""

Obtiene todos los encuentros, pudiendo filtrar por mes o año o ambos.

Reportes

Listar todos los encuentros para un deporte

GET api/encuentros/deporte/nombreDeporte

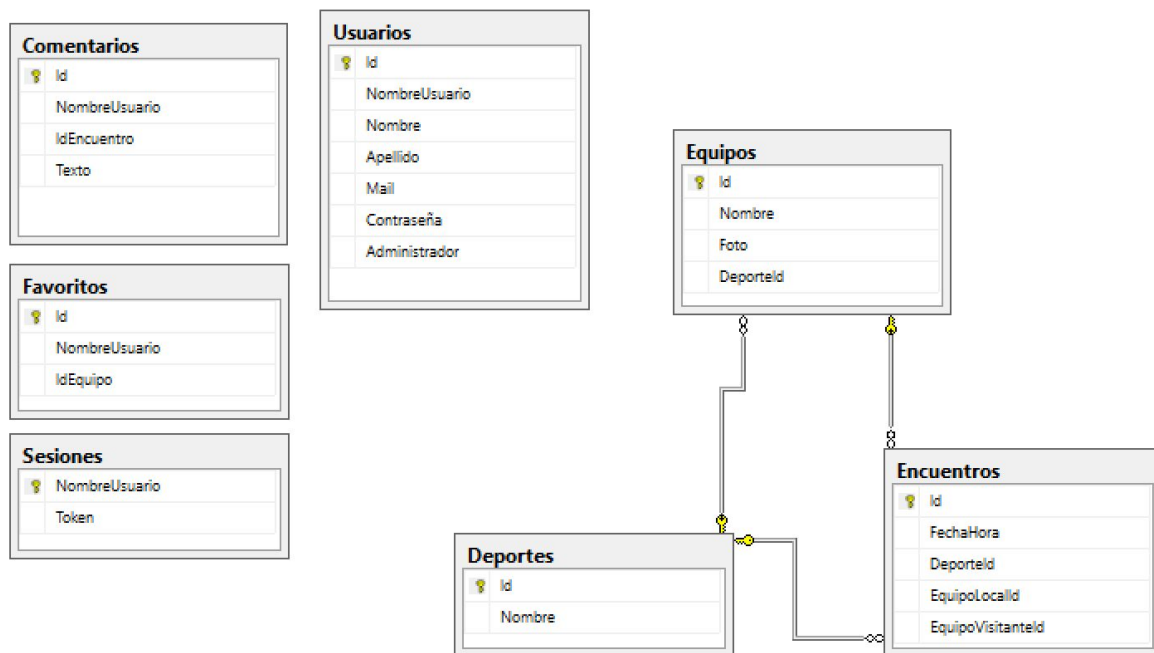
Listar todos los encuentros de un equipo en particular

GET api/encuentros/equipo/idEquipo

Modelo de datos

La solución se desarrolló utilizando EntityFrameworkCore aplicando Code First para luego generar la base de datos.

La estructura generada automáticamente se puede ver en el siguiente modelo:



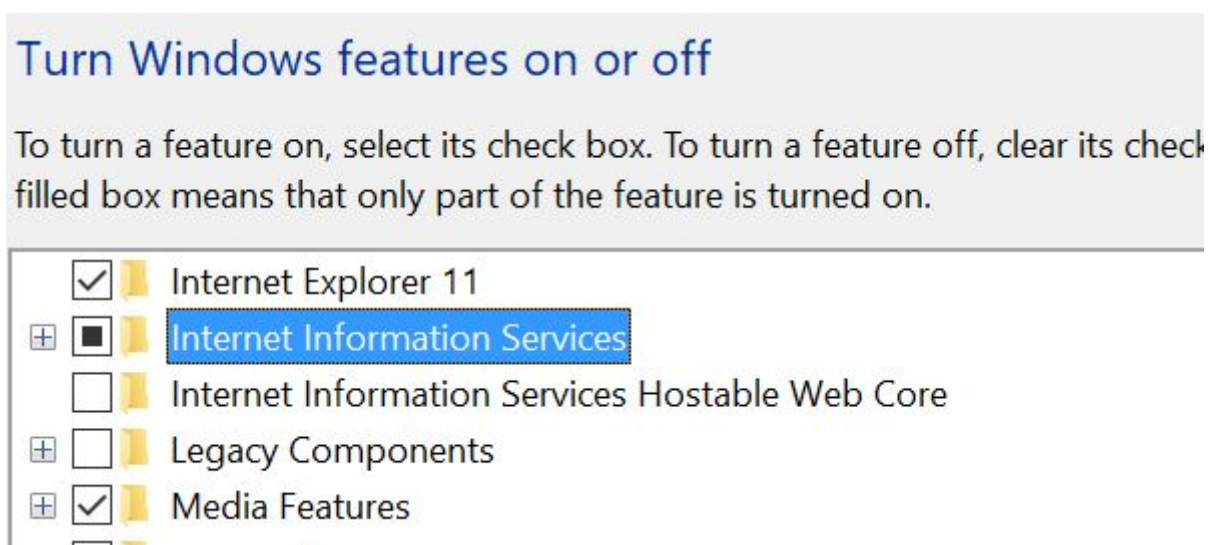
Instructivo de instalación

A continuación se detallan los pasos a seguir para poder realizar la correcta instalación de la Web Api junto con la base de datos en una o más máquinas con sistema operativo Windows y SQLServer.

1. Nos aseguramos que el servicio de SQL Server esté iniciado. Para ello vamos a Servicios de Windows y verificamos que el servicio SQL Server (SQLEXPRESS) esté en ejecución, de no estarlo le damos iniciar. Ver el nombre del servidor de SQL Server y agregarlo al connectionstring del appsettings.json de nuestra web api.
2. Antes de hacer el deployment, debemos asegurarnos de tener instalado IIS.

Vamos a:

Panel de control\Todos los elementos de Panel de control\Programas y características

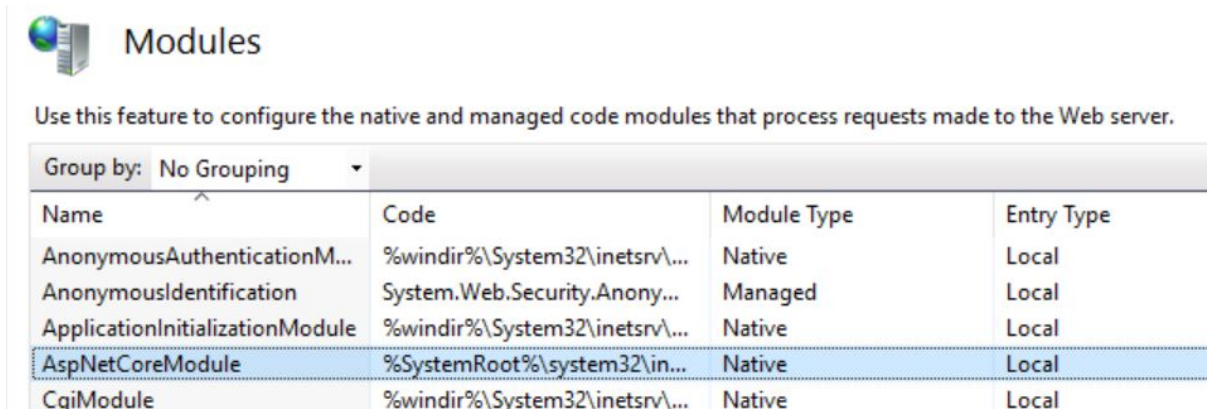


Y allí, elegimos: "Activar o desactivar características de Windows". Allí dentro, tiquear la opción Internet Information Services y todo su interior.

3. Copiamos el proyecto compilado en release (la carpeta publish que está contenida dentro de la carpeta Release en nuestro repositorio de github) y lo pegamos en C:\inetpub\wwwroot.

4. Abrimos el Administrador de Internet Information Services (IIS). Para lograrlo llegar a dicho administrador hacemos un run de "inetmgr" o buscando en Windows.

5. Ahora vamos a Módulos y nos fijamos si tenemos instalado el módulo "AspNetCoreModule" si no es el caso vamos [aquí](#) y descargamos .NET Core Runtime.

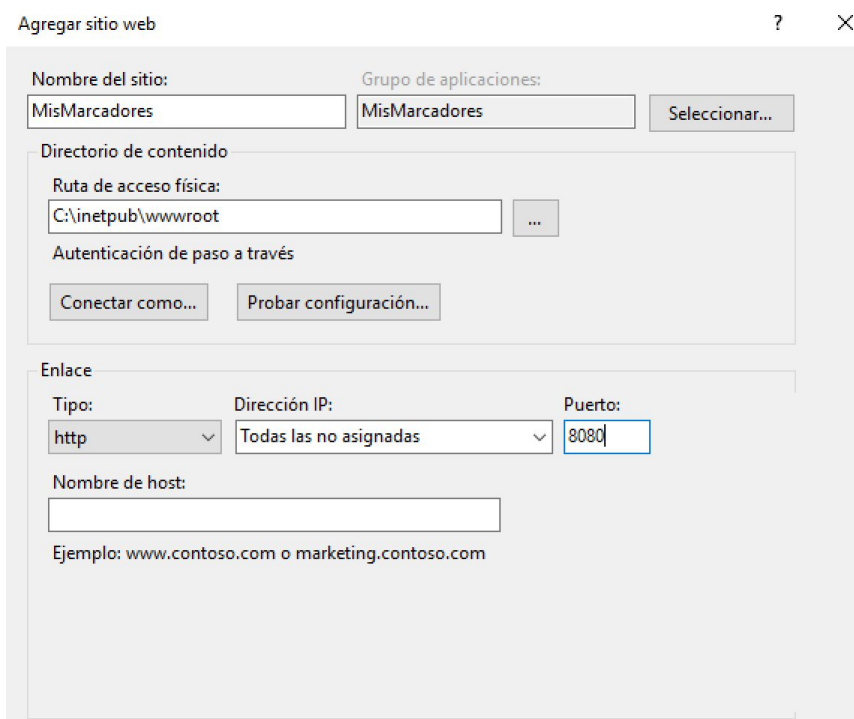


The screenshot shows the 'Modules' page in the IIS Manager. It includes a title bar with a globe icon and the word 'Modules'. Below the title is a description: 'Use this feature to configure the native and managed code modules that process requests made to the Web server.' There is a 'Group by:' dropdown menu set to 'No Grouping'. Below this is a table with four columns: 'Name', 'Code', 'Module Type', and 'Entry Type'. The table lists several modules, with 'AspNetCoreModule' highlighted in blue.

Name	Code	Module Type	Entry Type
AnonymousAuthenticationM...	%windir%\System32\inetsrv\...	Native	Local
AnonymousIdentification	System.Web.Security.Anony...	Managed	Local
ApplicationInitializationModule	%windir%\System32\inetsrv\...	Native	Local
AspNetCoreModule	%SystemRoot%\system32\in...	Native	Local
CgiModule	%windir%\System32\inetsrv\...	Native	Local

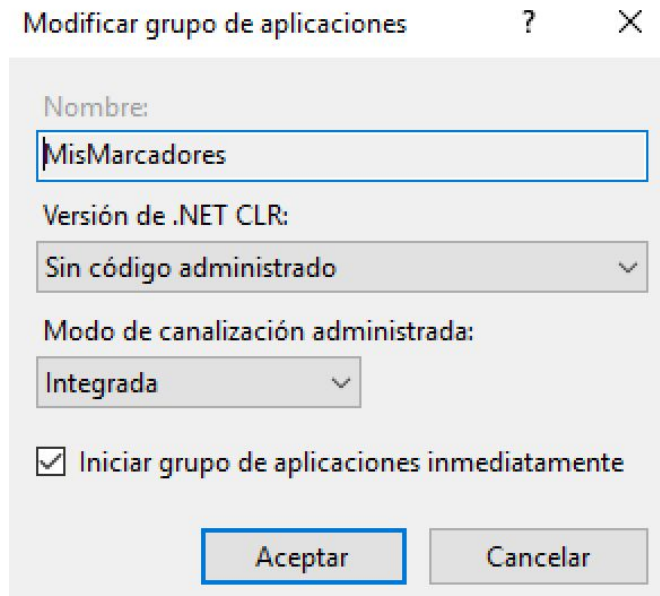
6. Agregamos un nuevo Sitio.

Para ello se hace click derecho sobre "Sitios" y se Agrega un Nuevo sitio. Se le pone un nuevo nombre, se elige la ruta física del proyecto de la Web Api que acabamos de copiar(C:\inetpub\wwwroot) y luego se elige el puerto 8080, 12345, o alguno que no esté en uso. En caso de elegir el puerto 80 (por defecto del iis), se debe detener el Default WebSite y luego darle Iniciar a nuestro sitio.



The screenshot shows the 'Agregar sitio web' (Add website) dialog box. It has a title bar with a question mark and a close button. The dialog is divided into several sections. The first section is 'Nombre del sitio:' (Website name) with a text box containing 'MisMarcadores' and a 'Grupo de aplicaciones:' (Application pool) dropdown also set to 'MisMarcadores', with a 'Seleccionar...' button. The second section is 'Directorio de contenido' (Content directory) with a 'Ruta de acceso física:' (Physical path) text box containing 'C:\inetpub\wwwroot' and a browse button (...). Below this is an 'Autenticación de paso a través' (Pass-through authentication) section with 'Conectar como...' and 'Probar configuración...' buttons. The third section is 'Enlace' (Binding) with 'Tipo:' (Type) set to 'http', 'Dirección IP:' (IP address) set to 'Todas las no asignadas' (All unassigned), and 'Puerto:' (Port) set to '8080'. There is also a 'Nombre de host:' (Host name) text box and an example: 'Ejemplo: www.contoso.com o marketing.contoso.com'.

7. Ahora vamos a Grupos de aplicaciones y buscamos el pool que le asignamos al sitio y hacemos clic en este en .NET CLR Version seleccionamos Sin código administrado y le damos a Aceptar.



Modificar grupo de aplicaciones ? X

Nombre:
MisMarcadores

Versión de .NET CLR:
Sin código administrado

Modo de canalización administrada:
Integrada

☒ Iniciar grupo de aplicaciones inmediatamente

Aceptar Cancelar

Base de datos

Abrir SQL Server 2014 Management Studio, al iniciar copiar el valor del campo "Server name".

Luego de conectarse al servidor, hacer click derecho en Databases y luego en Restore database. Seleccionar la opción Device, luego la ruta hacia uno de los respaldos provistos (vacía o con datos) y por último aceptar para que se cree la base.

Con la base de datos creada vamos a la carpeta "Security" dentro de nuestro servidor de base de datos y luego click derecho a "Logins" -> "New Login".

Como nombre de inicio de sesión agregamos: IIS APPPOOL\MisMarcadores y le damos "Aceptar".

Vemos que se agregó un nuevo Inicio de Sesión, le damos click derecho/propiedades y en roles del servidor, elegimos sysadmin y le damos aceptar.

Resultado de la ejecución de las pruebas

Este software fue desarrollado utilizando TDD, se fue construyendo primero la prueba y luego la implementación de lo probado por la misma.

El procedimiento fue siempre escribir la prueba, escribir la implementación y luego hacer refactor hasta que se pase la prueba.

Para esto se creó un paquete de prueba unitario utilizando pruebas automáticas con MSTests, el paquete sobre el que se realizó TDD fue el paquete Logic.

Esto nos permitió mockear el repository para probar la lógica de negocio.

El beneficio de esto es que se logró aislar el comportamiento de cada prueba para no depender de otros paquetes ni de otro comportamiento y así asegurarnos de que estábamos testeando exactamente lo que necesitábamos.

Para poder hacer mocking es que definimos interfaces en la lógica y en el repository, estas son las que exponen las funcionalidades de cada paquete.

Tanto para el desarrollo de las entidades como para el acceso y manejo de los datos con Entity Framework decidimos no implementar TDD.

Para las entidades consideramos que no era necesario ya que no hay lógica en este paquete y por lo tanto no tenía sentido hacer pruebas unitarias.

En cuanto al manejo de datos con EF no creímos necesario hacerlo ya que de esta manera estaríamos probando la herramienta, lo cual no tiene mucho sentido.


































Tampoco decidimos realizar TDD en la WebApi, más que nada por un tema de tiempo.

De todas formas a nuestra WebApi la testeamos con el postman, ya que siempre al hacer una request hacemos un test asegurándonos que nos devuelve el código correcto, así como el mensaje de error en caso de existir alguno.

Resultado de ejecución de las pruebas

En nuestro caso creamos un total de 65 pruebas, todas estas pruebas son superadas con éxito como se detalla a continuación.

MisMarcadores.Web.Api (65 tests)

▲  Passed Tests (65)	884 ms
 ActualizarDeporteDatosErroneosTest	< 1 ms
 ActualizarDeporteNoExistenteTest	1 ms
 ActualizarEncuentroDatosInvalidosTest	1 ms
 ActualizarEncuentroDeporteNoExistenteErrorTest	4 ms
 ActualizarEncuentroEquipoExisteEncuentroEnFechaErrorTest	13 ms
 ActualizarEncuentroEquipoNoExistenteTest	5 ms
 ActualizarEquipoDatosErroneosTest	1 ms
 ActualizarEquipoNoExistenteTest	3 ms
 ActualizarEquipoNuevoNombreYaExistenteTest	5 ms
 ActualizarUsuarioDatosErroneosTest	< 1 ms
 ActualizarUsuarioExistenteOkTest	5 ms
 ActualizarUsuarioNoExistenteTest	2 ms
 AgregarDeporteExistenteErrorTest	5 ms
 AgregarDeporteNombreVacioTest	9 ms
 AgregarDeporteOkTest	37 ms
 AgregarEncuentroDeporteNoExistenteErrorTest	4 ms
 AgregarEncuentroEquipoLocalExisteEncuentroEnFechaErrorTest	12 ms
 AgregarEncuentroEquipoLocalNoExistenteErrorTest	5 ms
 AgregarEncuentroEquipoNombreVacioTest	2 ms
 AgregarEncuentroEquipoVisitanteExisteEncuentroEnFechaErrorTest	11 ms
 AgregarEncuentroOkTest	46 ms
 AgregarEquipoDeporteNoExistenteErrorTest	3 ms
 AgregarEquipoExistenteErrorTest	5 ms
 AgregarEquipoNombreVacioTest	4 ms
 AgregarEquipoOkTest	5 ms
 AgregarUsuarioApellidoVacioTest	< 1 ms
 AgregarUsuarioContraseñaVaciaTest	< 1 ms
 AgregarUsuarioMailFormatoErroneoTest	2 ms
 AgregarUsuarioNombreUsuarioVacioTest	< 1 ms
 AgregarUsuarioNombreVacioTest	2 ms
 AgregarUsuarioOkTest	9 ms
 BorrarDeporteNoExistenteErrorTest	2 ms

✓ BorrarDeporteOkTest	9 ms
✓ BorrarEncuentroNoExistenteErrorTest	2 ms
✓ BorrarEncuentroOkTest	7 ms
✓ BorrarEquipoNoExistenteErrorTest	2 ms
✓ BorrarEquipoOkTest	4 ms
✓ BorrarUsuarioNoExistenteErrorTest	3 ms
✓ BorrarUsuarioOkTest	5 ms
✓ CerrarSesionTestOK	3 ms
✓ IniciarSesionInvalidaErrorTest	3 ms
✓ IniciarSesionTestOk	11 ms
✓ ObtenerDeportePorNombreErrorNotFoundTest	3 ms
✓ ObtenerDeportePorNombreOkTest	3 ms
✓ ObtenerDeportesNullTest	1 ms
✓ ObtenerDeportesOkTest	360 ms
✓ ObtenerEncuentroPorIdErrorNotFoundTest	3 ms
✓ ObtenerEncuentroPorIdOkTest	8 ms
✓ ObtenerEncuentrosNullTest	< 1 ms
✓ ObtenerEncuentrosOkTest	163 ms
✓ ObtenerEncuentrosPorDeporteOkTest	4 ms
✓ ObtenerEncuentrosPorEquipoOkTest	4 ms
✓ ObtenerEquipoPorIdErrorNotFoundTest	2 ms
✓ ObtenerEquipoPorIdOkTest	3 ms
✓ ObtenerEquipoPorNombreOkTest	3 ms
✓ ObtenerEquiposNullTest	< 1 ms
✓ ObtenerEquiposOkTest	2 ms
✓ ObtenerSesionPorTokenErrorNotFoundTest	3 ms
✓ ObtenerSesionPorTokenTestOk	14 ms
✓ ObtenerUsuarioPorNombreUsuarioErrorNotFoundTest	3 ms
✓ ObtenerUsuarioPorNombreUsuarioOkTest	3 ms
✓ ObtenerUsuarioPorTokenErrorNotFoundTest	2 ms
✓ ObtenerUsuarioPorTokenTestOk	7 ms
✓ ObtenerUsuariosNullTest	< 1 ms
✓ ObtenerUsuariosOkTest	10 ms

Cobertura de las pruebas

Para realizar la cobertura de las pruebas utilizamos coverlet.

```
PS C:\Users\User\Documents\anda_anda\MisMarcadores.Web.Api\MisMarcadores.Logic.Tests> dotnet test ./MisMarcadores.Logic.Tests.csproj /p:CollectCoverage=true /p:CoverletOutputFormat=opencover
Compilación iniciada, espere...
Se completó la compilación.

Serie de pruebas para C:\Users\User\Documents\anda_anda\MisMarcadores.Web.Api\MisMarcadores.Logic.Tests\bin\Debug\netcoreapp2.0\MisMarcadores.Logic.Tests.dll(.NETCoreApp,Version=v2.0)
Herramienta de línea de comandos de ejecución de pruebas de Microsoft(R), versión 15.7.0
Copyright (c) Microsoft Corporation. Todos los derechos reservados.

Iniciando la ejecución de pruebas, espere...

Total de pruebas: 65. Correctas: 65. Con error: 0. Omitidas: 0.
La serie de pruebas se ejecutó correctamente.
Tiempo de ejecución de las pruebas: 3,7217 Segundos

Calculating coverage result...
Generating report 'C:\Users\User\Documents\anda_anda\MisMarcadores.Web.Api\MisMarcadores.Logic.Tests\coverage.opencover.xml'

+-----+-----+-----+-----+
| Module                                | Line | Branch | Method |
+-----+-----+-----+-----+
| MisMarcadores.Data.DataAccess         | 0%    | 0%      | 0%      |
+-----+-----+-----+-----+
| MisMarcadores.Data.Entities           | 33,7% | 0%      | 56,4%   |
+-----+-----+-----+-----+
| MisMarcadores.Logic                   | 52,4% | 49,4%   | 66,1%   |
+-----+-----+-----+-----+
| MisMarcadores.Repository               | 0%    | 0%      | 0%      |
+-----+-----+-----+-----+
```

Vemos que los resultados de la cobertura del código en los métodos de la lógica es de aproximadamente un 66%.

Se debe tener en cuenta que no realizamos pruebas unitarias en el repository ni en el DataAccess porque creímos que era mejor realizar pruebas de integración, otro factor era evitar acceder a la base de datos constantemente, para cambiar datos de prueba.

Las pruebas de integración se realizaron manualmente ejecutando las collections desde Postman y verificando las respuestas obtenidas.

ANEXOS

Justificación de Clean Code

Este software fue desarrollado con la metodología clean code.

Para lograrlo se cumplieron con las diferentes reglas que establece esta metodología, que a continuación se listan y detallan:

Nombres de Clases, Métodos y Variables

La nomenclatura se definió para que al leer el nombre de alguno de estos elementos se pueda comprender fácilmente cuál es su función a cumplir y que no haya lugar a confusiones o ambigüedades. Además que sea fácil de pronunciar.

Métodos

Todos los métodos que componen nuestro sistema fueron hechos con la menor cantidad de parámetros posibles ya que esto facilita la lectura del código y lo hace mas entendible. Además cada método hace lo que dice su nombre y se extiende lo menos posible en cuanto a líneas de código.

Cada vez que un método se extendía demasiado se decidió dividirlo en más métodos y así mejorar la lectura y mantenimiento de estos.

Organización de clases

Las clases fueron codificadas respetando un orden, primero las variables públicas y estáticas y luego los métodos públicos. El tamaño de las clases es correcto.

Cada clase tiene un nombre claro y tiene su comportamiento esperado, no existen clases que contengan funcionalidades que no estén relacionadas a lo que se espera de ellas.

Pruebas

Pruebas claras, nombres de métodos y atributos completos. Se trató de usar un Assert por prueba en la mayor cantidad de pruebas, en algunos casos se utilizaron dos o tres pero fueron casos aislados. No se partieron esos casos en más casos por falta de tiempo. Se cumple con las reglas F.I.R.S.T

Comentarios

El código carece de comentarios ya que el uso de patrones auto-explicativos y una nomenclatura acorde a cada funcionalidad en cuestión determinaron como innecesarios el uso de comentarios..

Excepciones

En cuanto al manejo de errores se optó por usar excepciones en vez de códigos de error y siguiendo los lineamientos de Clean Code creamos una excepción llamada LogicException que sirve de padre al resto de las excepciones de nuestro sistema.