Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de aplicaciones 2
Obligatorio 1

Santiago Larralde 217922
Cristhian Maciel 163471

Entregado como requisito de la materia Diseño de Aplicaciones 2
9 de mayo de 2019

Índice

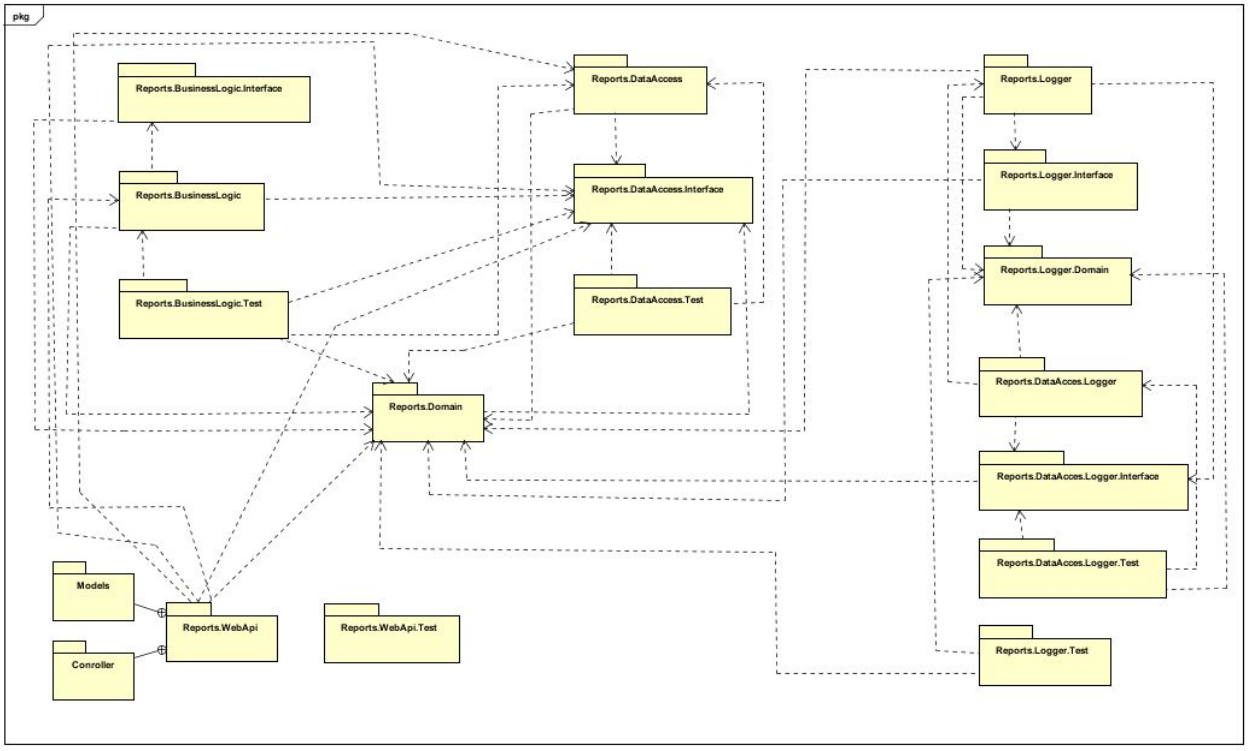
Descripción del trabajo

Construimos un sistema de visualización de reportes para usuarios con ciertos privilegios. El mismo acepta dos tipos de usuarios, usuarios gerentes y usuarios administradores, y según su rol en el sistema es que pueden realizar diferentes acciones. Los usuarios de tipo gerente tendrán acceso a la visualización de indicadores de las áreas a las que pertenecen, pudiendo seleccionar cual o cuales indicadores desea ver y en qué orden.
los usuarios de tipo admin son los encargados de creer los indicadores, las áreas y los demás usuarios.

Arquitectura

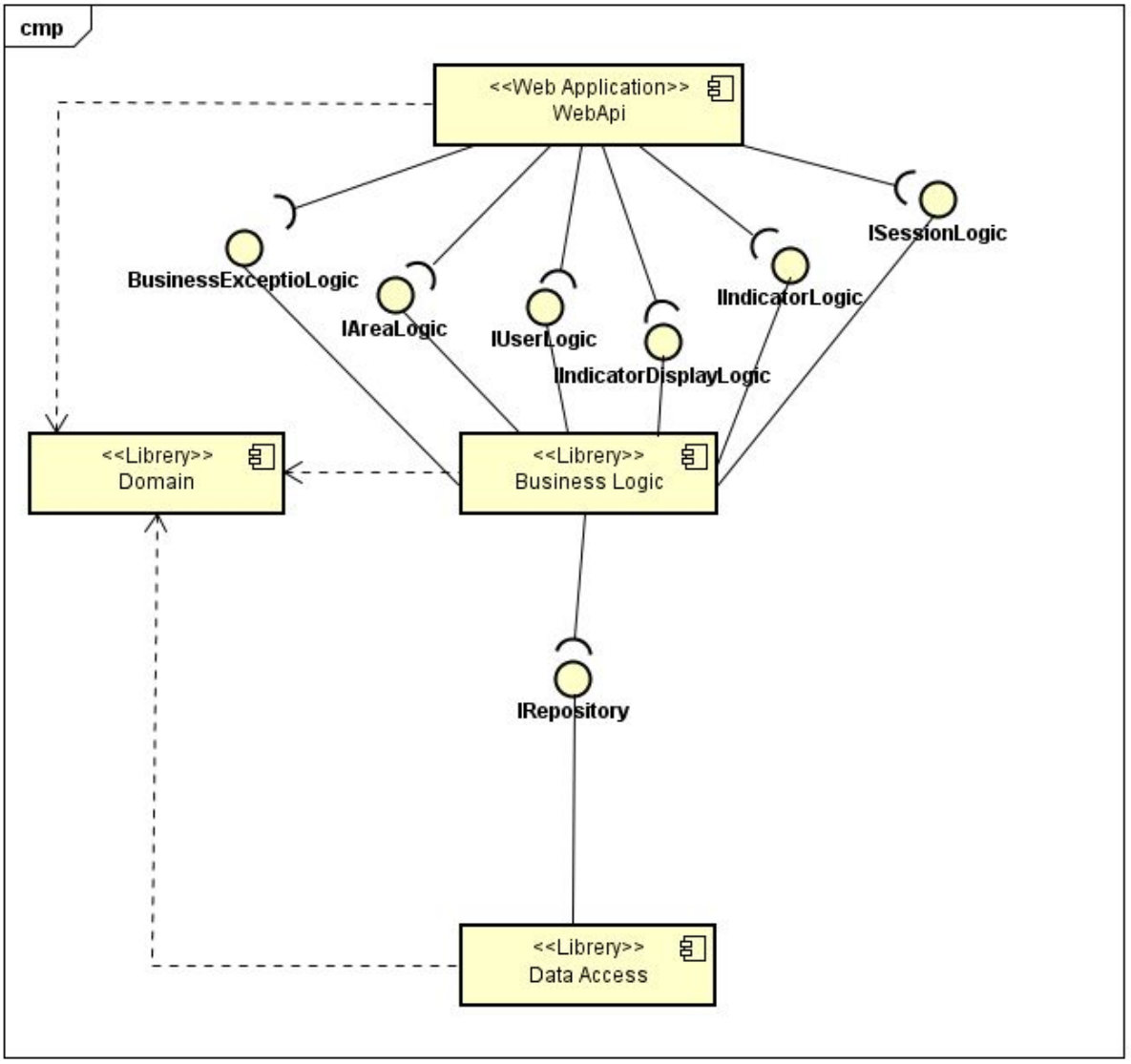
Nuestra solución está compuesta de una arquitectura de 4 capas : capa de WebApi, capa de lógica, capa de Repositorio y capa de Dominio. A continuación, se muestra un diagrama de paquetes y un diagrama de componentes que muestran la interacción de las mismas.

Diagrama de paquetes



En el diagrama se puede visualizar las decisiones tomadas en cuanto a la organización de los paquetes, y sus dependencias en tiempo de compilación.
Si dividimos nuestro diagrama en alto y bajo nivel vemos que el alto nivel, (el dominio y las business logic) no dependen del bajo nivel siendo este la webapi y el acceso a datos, sino que se acoplan por intermedio de interfaces bien definidas, esto nos permite no violar el principio de inversión de independencia (DIP), y además trae consigo los beneficios que esto implica, ya sea mayor nivel de indirección, uso de polimorfismo etc. Si bien cumple DIP, somos conscientes de las posibilidades de mejora existente, por ejemplo nuestra webapi está acoplada ya sea en BusinessLogic o en DataAcces al paquete con las clases concretas y al paquete que contiene las interfaces, lo ideal seri está acoplado solo al de las interfaces, y mediante la utilización de reflexión podríamos haberlo obtenido, ya que la relación de uso se da porque la webapi hace el new de la clase.

Diagrama de componentes



En este diagrama vemos como los componentes de nuestro sistema interactúan entre si, nuestros componentes son la webapi, el dominio, la lógica de negocios y el acceso a datos, en este diagrama también puede verse el cumplimiento de DIP ya que el alto nivel se relaciona con el bajo a través de interfaces bien definidas.

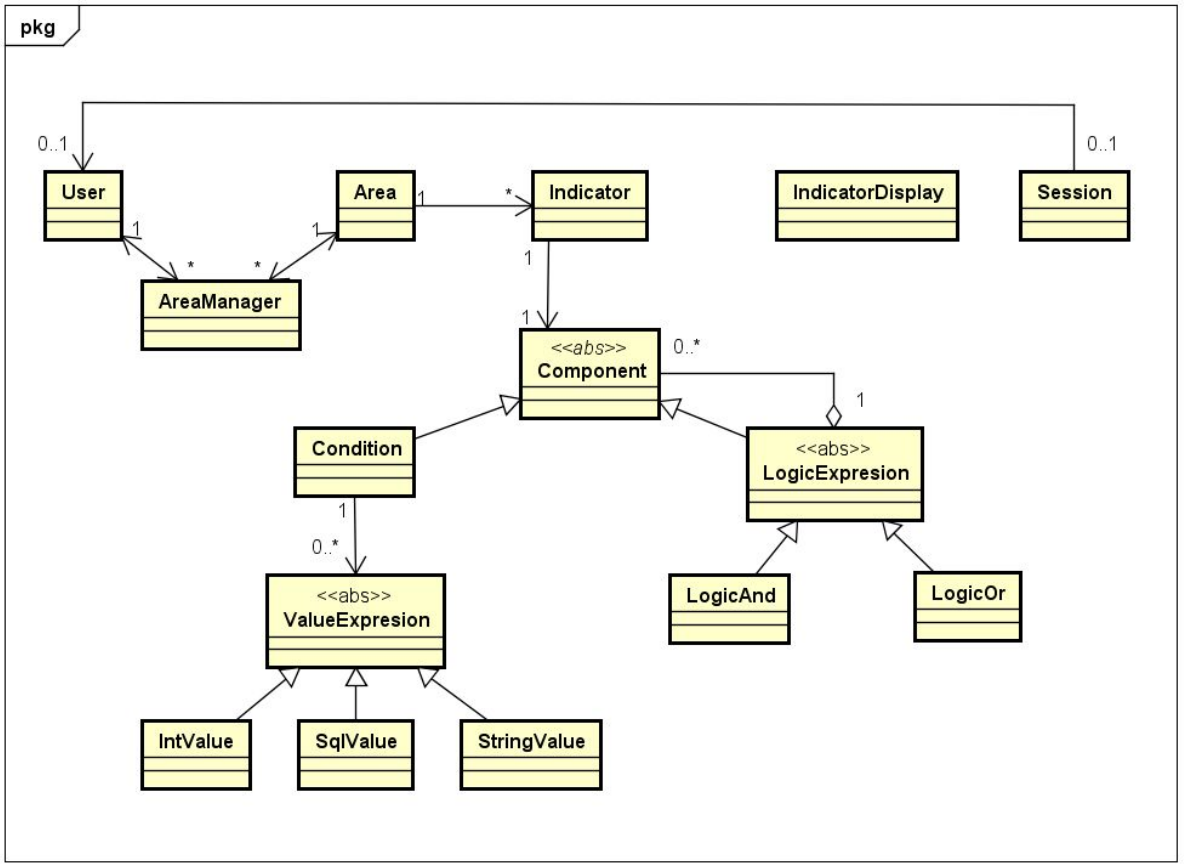
Dominio

Capa de Dominio:
Es la encargada de contener el dominio del problema, la misma no cuenta con lógica y solamente cuenta con la estructura de los datos a manejar en el sistema. De esta forma logramos favorecer el GRASP que nos recomienda tener alta cohesión ya que las responsabilidades individuales son únicas.
Por la forma en la cual se realizaron las dependencias entre los paquetes, se logró

también un bajo acoplamiento, el cual va a ser especificado a continuación.
Si bien en un principio pensamos en utilizar una generalización para usuarios pero optamos por una solución diferente que fue identificar el rol dentro de la clase, ya que la letra garantiza que estos serán los roles a mantener.

Las clases representadas serán luego las permitidas en base de datos.
A continuación se muestra un como queda organizado nuestro dominio a nivel de clases

Diagrama de clases de Dominio

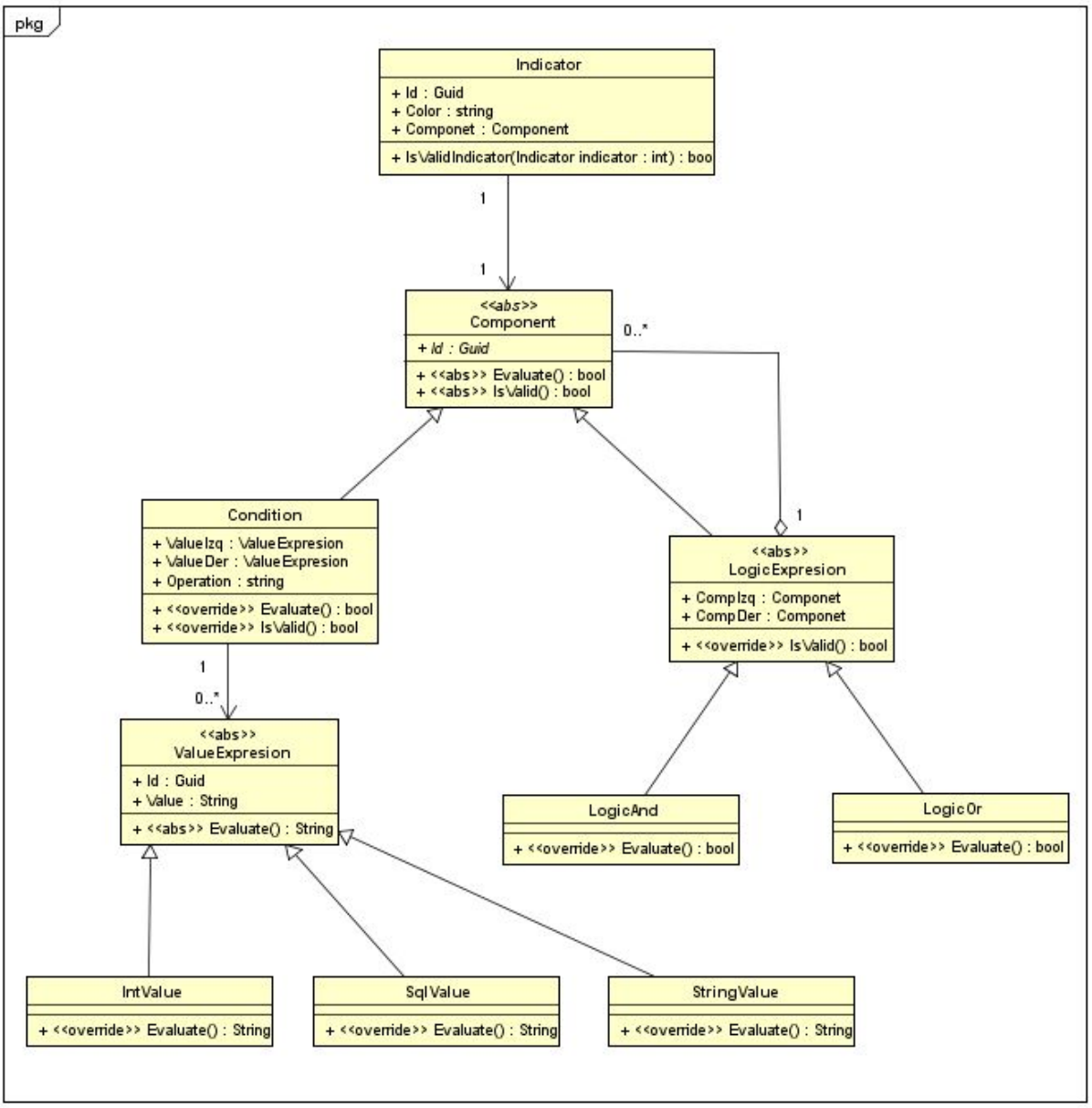


Como puede observarse hay una clase de asociación llamada AreaManager, que no se desprendido de la decodificación de la letra, sino de una limitante que tiene entity framework core, el cual no permite el mapeo automático a base las relaciones n a n. Por este motivo es que fue creada dicha clase como consecuencia del uso de esta tecnología.

Dentro de los problemas a resolver se encontraba el modelado de indicadores, para esto fuimos mutando nuestra solución hasta encontrar una que consideramos óptima.

en primera instancia habíamos guardado una lista de operadores lógicos, una de operadores de comparación, y otra con cada uno de los datos que podíamos recibir, y manejando esas listas podíamos armas los indicadores. Al fin y al cabo esto era funcional, pero era muy compleja su lógica de ejecución, rebuscada y muy poco flexible, luego de ellos pasamos a tratar el problema como una estructura arborescente y en la iteración de esta forma es que llegamos a utilizar el patrón composite, eso sí un poco customizado a nuestra realidad.
La solución final cuenta de una clase Component que será la evaluación final del indicador, tenemos los nodos hoja que serán las comparaciones entre los diferentes tipos de datos con los diferentes operadores y luego tenemos los nodos que no son hoja los cuales serán los comparadores lógicos and y or.
Esta solución es más flexible y mantenible de la que partimos inicialmente, los cual nos deja satisfechos en este aspecto.

Diagrama de modelado de indicadores



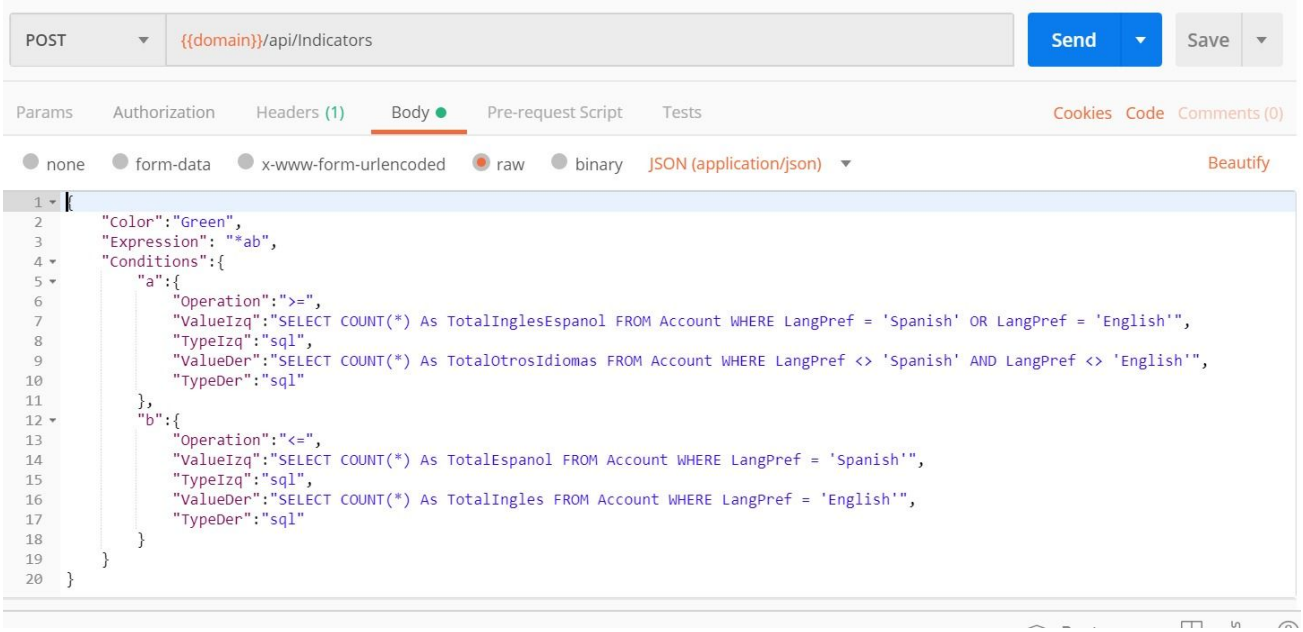
Creación de Indicadores

En esta sección explicaremos que mandamos en Body a la hora de crear un indicador

En el body del request enviamos toda la información necesaria para poder crear un indicador en formato Json.

¿Qué información tenemos que mandar?
En el body debemos mandar el color y la expresión que el indicador debe evaluar.
¿En que formato lo mandamos?
El color lo mandamos como un string en el campo Color, en el campo expresión enviamos lo la expresión a evaluar escrita en forma infija, es decir primero el Operador lógico luego el operando izquierdo y luego el operando derecho, el AND lo mandamos con el caracter “*” y el OR con el caracter “+”.
Dicho esto si quisiéramos mandar la expresión “a AND b” deberíamos mandarla como “*ab”.
y si quisiéramos mandar la expresión “a OR b” deberíamos mandarla como “+ab”.

Ejemplo de construcción de un indicador.



WebApi

En este paquete se definen los entry point de nuestra app.
Provee las clases que se comunican con la UI (posman), la WebApi solamente llama a la capa de la lógica para que se realicen las funciones correspondientes a cada endpoint de
la WebApi, devolviendo de manera correcta los datos al usuario una vez que se haya concretado la petición.
Contiene 2 paquetes fundamentales para el diseño de nuestro proyecto, un paquete de Models, que es el que tiene los modelos de las clases del dominio. Los cuales serán usados para recibir y para devolver los datos al usuario. También fueron utilizados para filtrar datos que no se quieran devolver o que no se quieran recibir.

En la misma se realiza la conversión (serialización y deserialización), es decir se parsea de una clase de dominio a un modelo. Esto lo hacemos para poder elegir qué información de las entidades queremos exponer, el caso típico es el usuario que tiene datos sensibles como lo puede ser su contraseña, para eso cuando devolvemos un usuario se castea a un modelo de usuario que no contenga ese campo de esa forma no estamos exponiendo datos críticos hacia fuera de la app.
Los modelos creados son: AreaManagerModel, AreaModel, ConditionModel, IndicatorDisplayModel, IndicatorModel, LoginModel, SessionModel, UserModel.

El paquete de Controllers, que es el que contiene los endpoints del sistema, los cuales serán llamados por el usuario mediante verbos HTTP y una correcta nomenclatura y estructura.

También es la encargada de interpretar las excepciones provenientes de la lógica y mostrarlas de mejor manera al usuario.
La elección de las uris para hacer las llamadas a cada endpoint, fueron siguiendo el estándar de no más de 3 niveles.

Endpoint expuestos

Controllers	Resource	Post	Get	Put	Delete
UserController	/users	Crea un nuevo usuario	Devuelve todos los usuarios	-	-
	/users/{id}	-	Devuelve el usuarios identificado con id	Modifica el usuario identificado con id	Elimina el usuario identificado con id
AreaController	/areas	Crea una nueva área	Devuelve todas las áreas	-	-
	/areas{id}	-	Devuelve el área identificada con id	Modifica el ares identificada con id	Elimina el área identificada con id
	/areas/{id}/Manager	Agrega un manager al area	Obtiene los manager de un area	-	Elimina los manager de un area

	/areas/{id}/Indicator	Agrega un indicador al area	Obtiene los indicadores de un area	-	Elimina los indicadores de un area
IndicatorController	/indicators	Crea un nuevo indicador	Devuelve todos los indicadores	-	-
	/indicators/{id}				
	/indicators{id}	-	Devuelve el indicador identificado con id	Modifica el indicador identificado con id	Elimina el indicador identificado con id
IndicatorDetailController	/areas/indicationDetail/{id}	-	devuelve del detalle de un indicador	-	-

Diagramas de secuencia:

Diagrama para crear un usuario

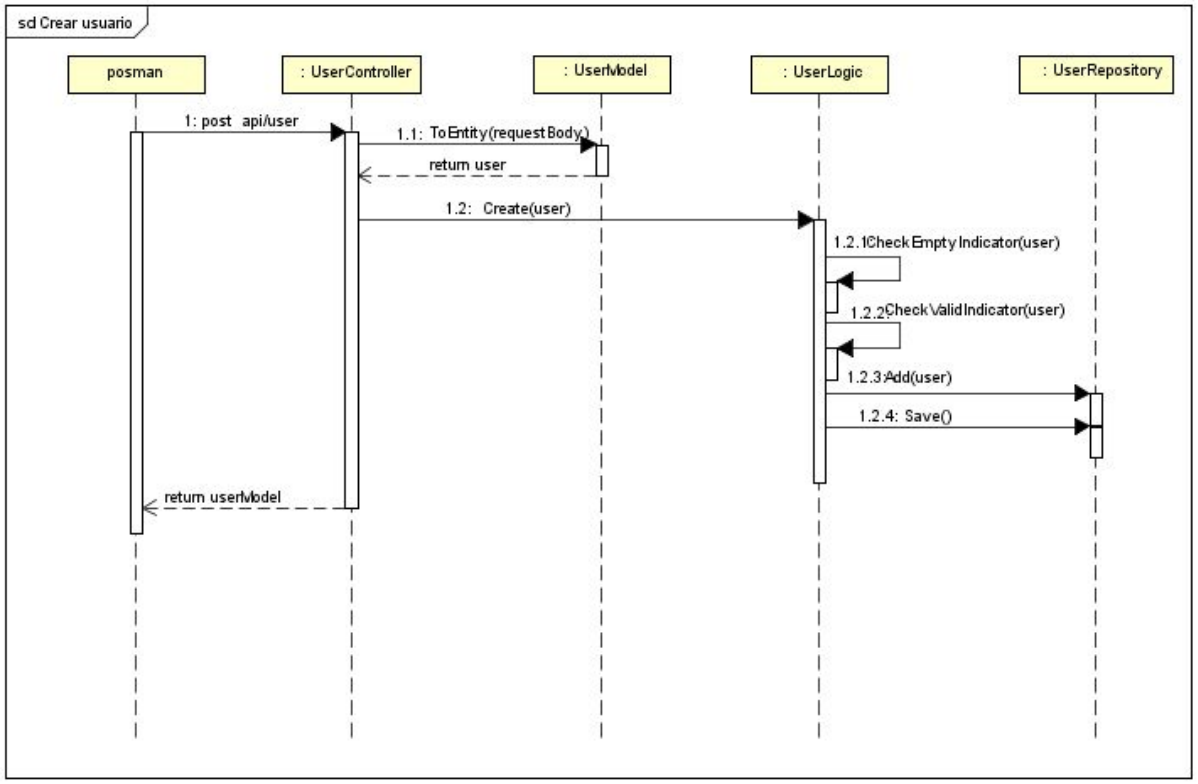


Diagrama para crear un área

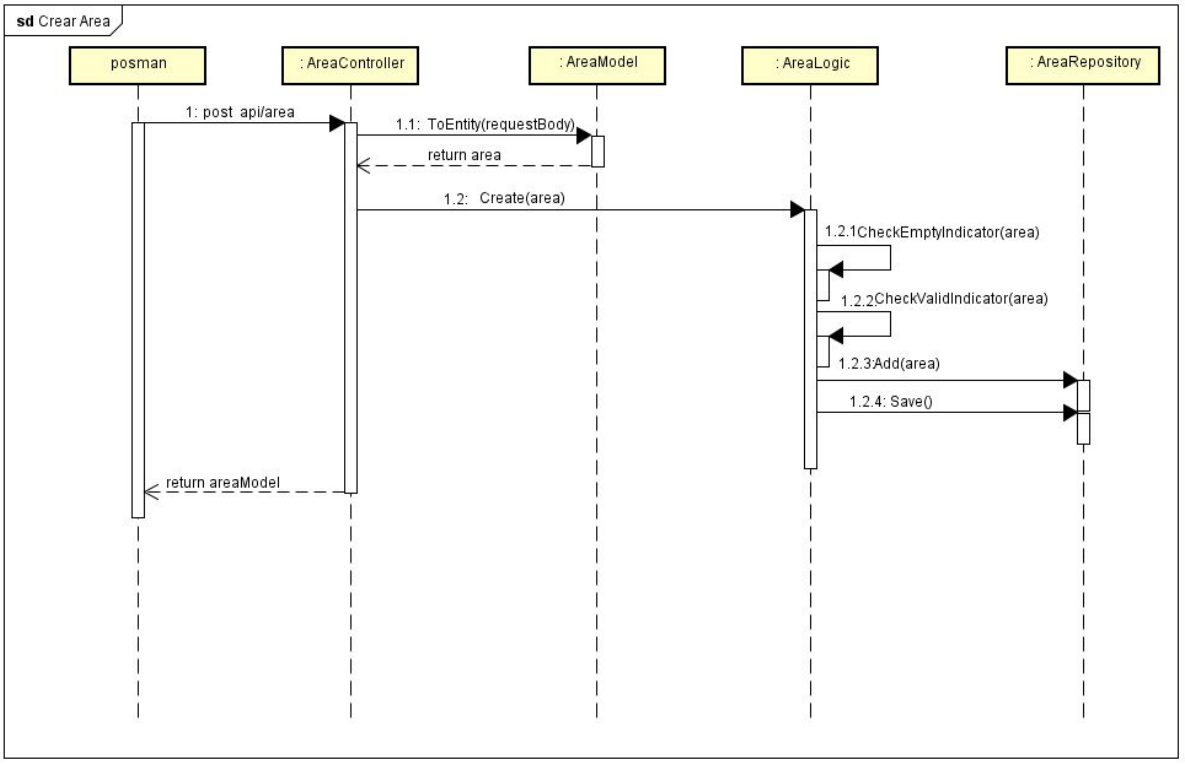
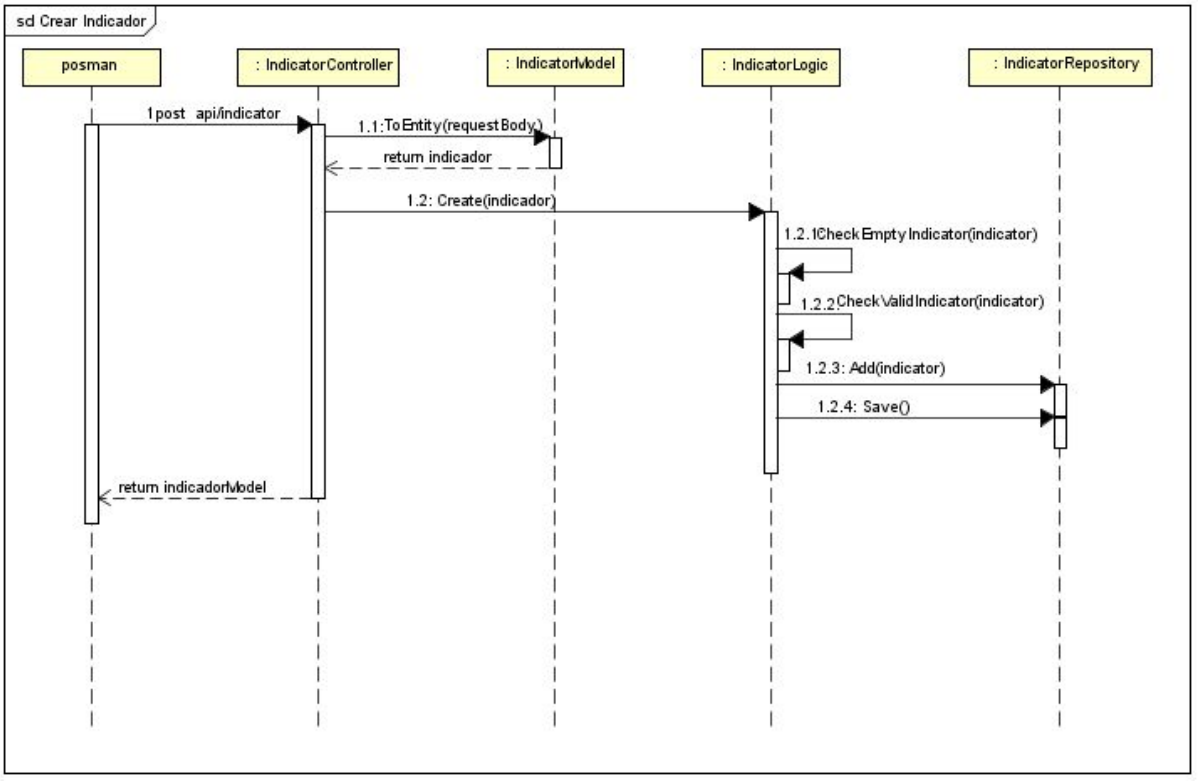
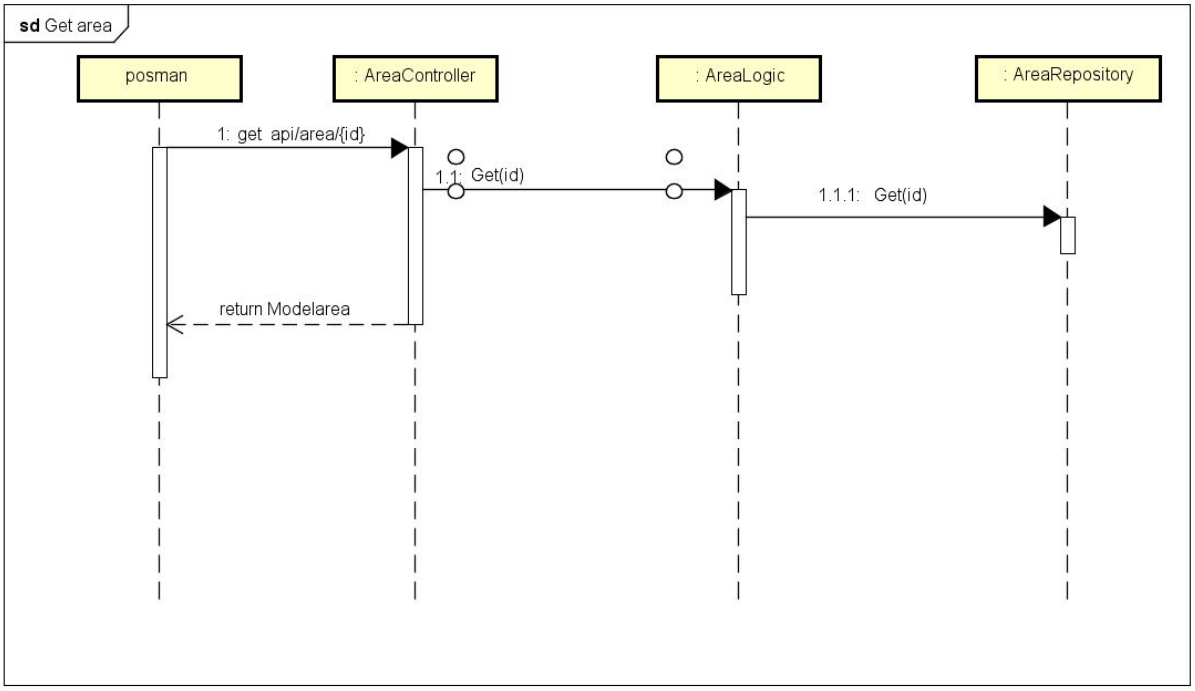


Diagrama para crear un Indicador



Como se puede ver los diagramas son muy similares en su concepto, por ellos para obtener las entidades solo realizaremos el de área ya que los de area e indicador siguen la misma lógica

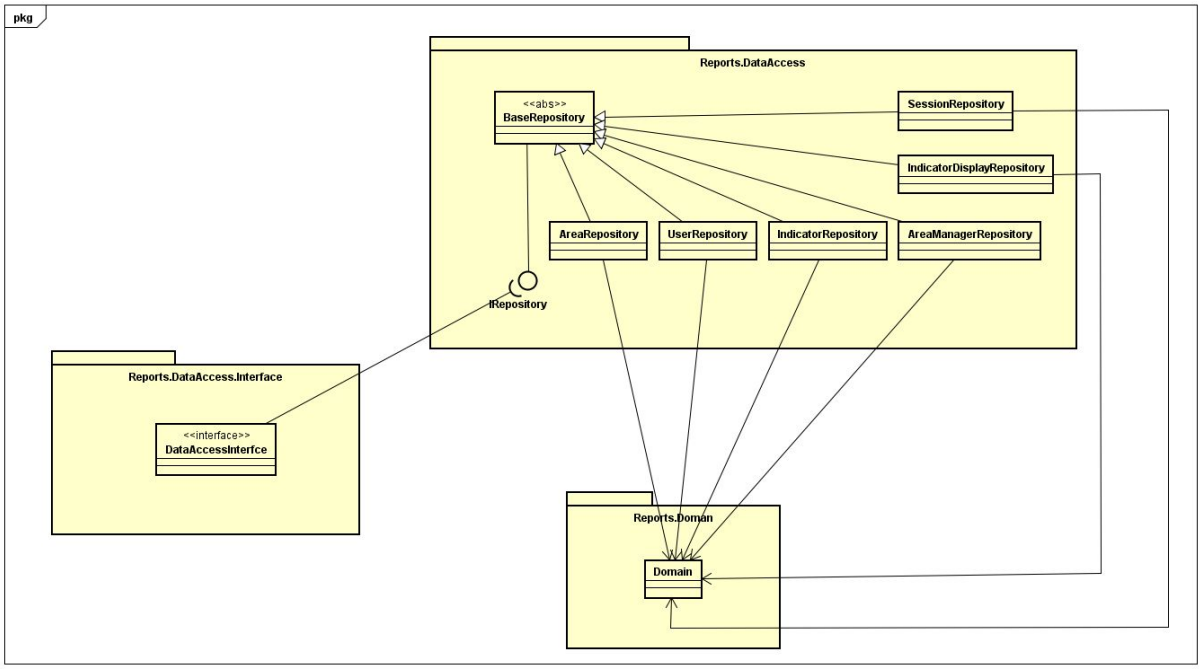
Diagrama para obtener un área



Manejo de acceso a datos.

Esto fue resuelto mediante el uso de dos paquetes un paquete que requiere una interfaz llamada IRepository genérico la cual define los métodos CRUD para acceso a base, y otro paquete que implementa esa interfaz. En el mismo se encuentra la clase BaseRepository que provee dicha interfaz la mismas es abstracta y genérica con lo que será extendida por otras clases que customizar los métodos que necesiten de acuerdo a la realidad del negocio.

Diagrama de acceso a datos

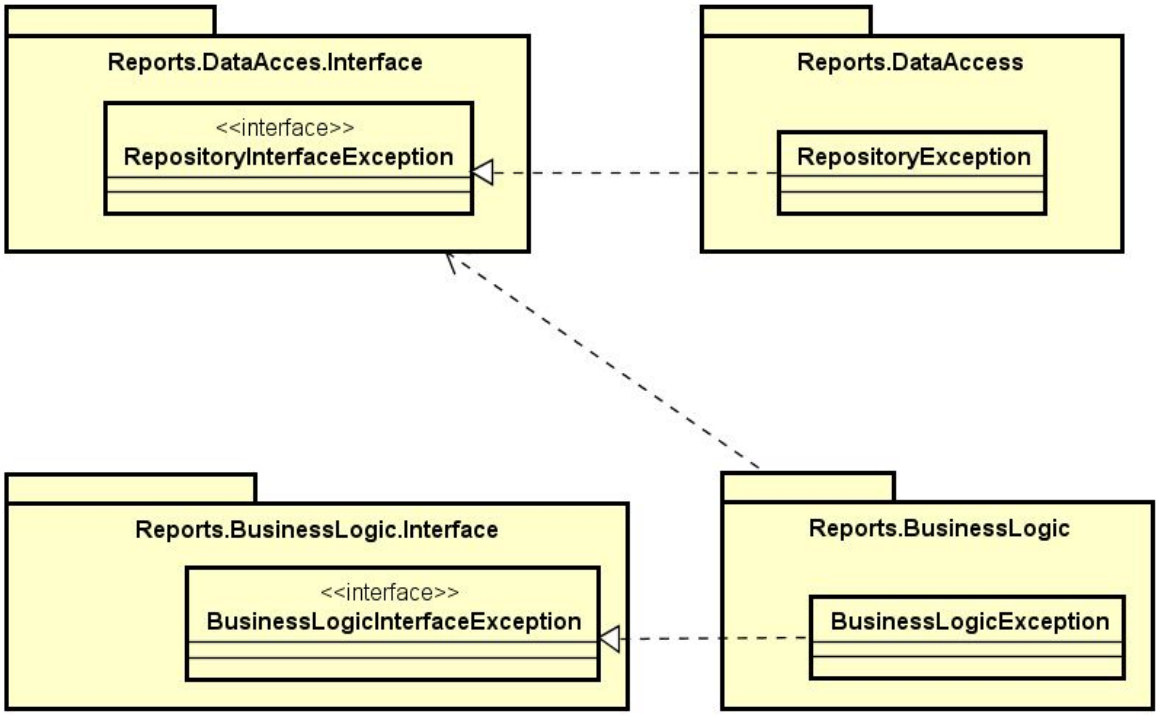


Manejo de excepciones.

En este caso decidimos utilizar un manejo de excepciones por capas, para tener mayor flexibilidad y no estar todas las clases acopladas al mismo tipo de excepción.

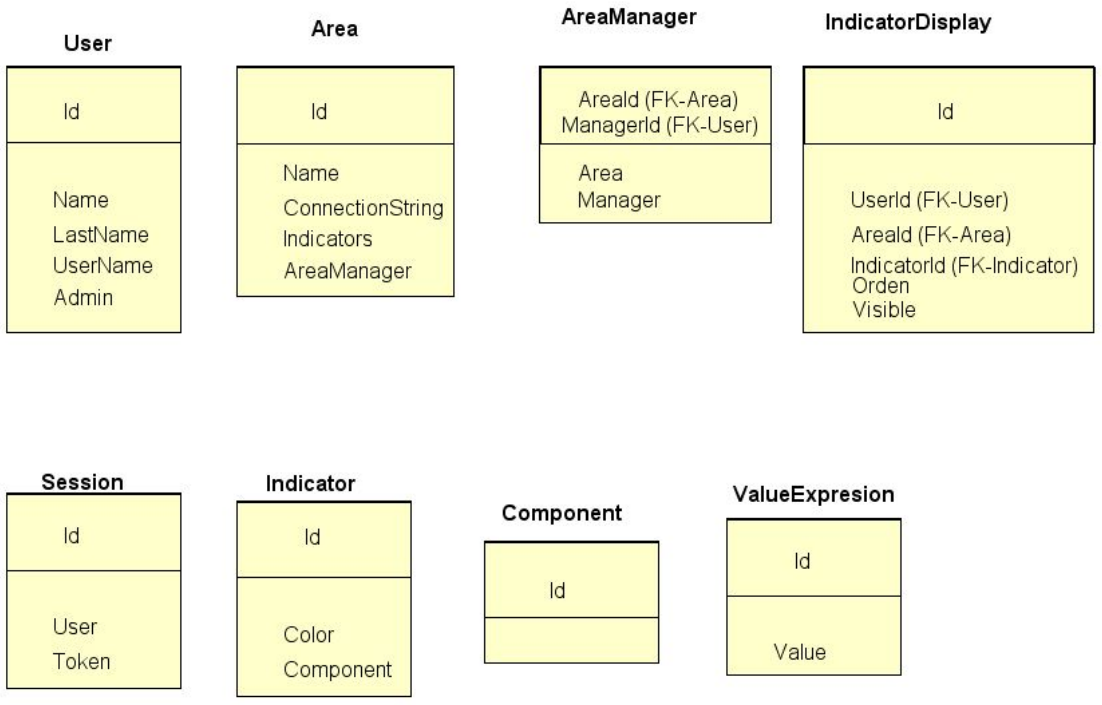
La WebApi, catchea excepciones que arroja la lógica de negocio y muestras un mensaje.
La lógica de negocio catchea las excepciones que arroja el acceso a datos las transforma en excepciones propias y las arroja.
El acceso a datos catchea excepciones del acceso a las tablas en sí y las arroja como excepciones propias.

Diagrama de manejo de excepciones



Modelo de base de datos

Para la creación de la base se utilizó Entity framework core, se creó la base y se generó la migración mediante la forma code first, es decir primero construimos nuestras entidades a persistir y luego se generó la migración.



Clean Code y estándares.

A lo largo de toda la solucion mantuvimos los estandares de codificacion c# como por ejemplo que los métodos comienzan todos con mayúscula y utilizan CamelCase, la variables con minúscula y también con CamelCase.

Una de las prácticas más comunes de Clean Code son los nombres mnemotécnicos para las variables, clases y métodos, algo que utilizamos extensivamente en el desarrollo de nuestro programa.
El programa está escrito en inglés, por lo que todos los métodos y variables también lo están. Además, cada variable y nombre de método fueron escritos pensando que fueran lo más nemotécnico posible.

Los métodos construidos tienen responsabilidad única.
Otra práctica de Clean Code es pasar pocos parámetros en un método.
En la mayoría de los casos respetamos esto, pasando a lo sumo 2 parámetros por método. Las únicas excepciones son los constructores que en algún caso recibe más de 2 parámetros.

Tratamos de que nuestros métodos tengan un máximo de dos niveles de abstracción y eso lo logramos delegando responsabilidades en otros métodos.
En general los métodos no superan las 15 líneas y las clases no superan las 200 líneas salvo las clases de test.

El para la mantención del repositorio en github definimos un estándar interno de commit, el mismo cuenta en que para cada funcionalidad a desarrollar deberíamos hacer una rama nueva y su nombre sería feature/*funcionalidad,* una vez terminada la funcionalidad se hacia un merge con la rama develop.
Hay ramas que no comienzan con feature/ y esas son las ramas que no hacer referencia a una funcionalidad en sí, o son refactor de otra rama.
Para crear una rama nueva, debíamos salir siempre desde develop estando esta actualizada para evitar conflictos a posteriori.
Realizar un solo merge master con el obligatorio terminado.

Cobertura de las pruebas.

Se realizaron también pruebas de postman, las cuales sirvieron como pruebas de integración y pruebas funcionales del software. Las cuales fueron corridas a mano,

verificando en la base de datos y comprobando que estaban los resultados correctamente.

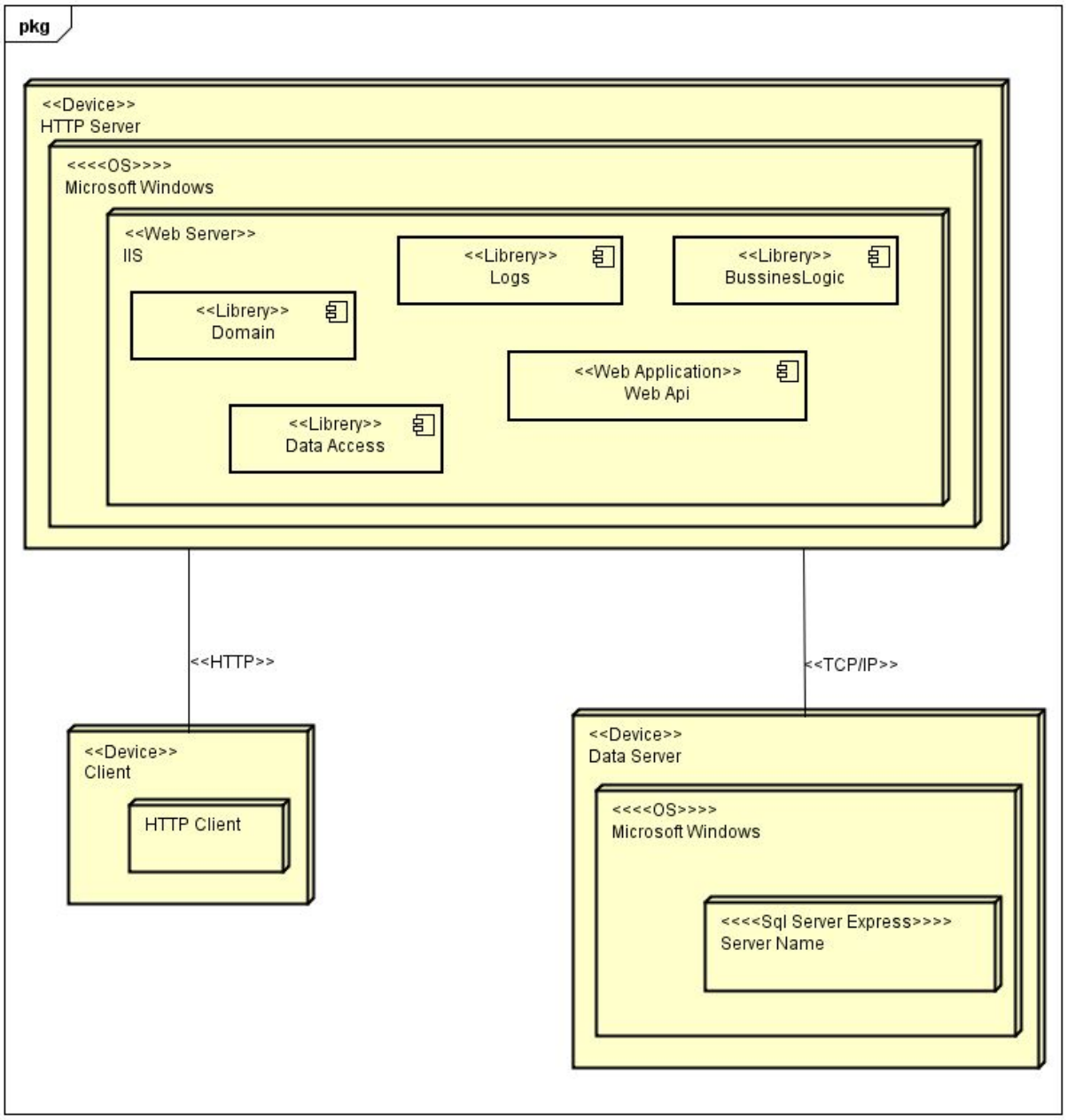
A pesar de no tener una cobertura excelente en la lógica, reforzamos esto con pruebas de
integración en el Postman, que verifican la interacción de la lógica con el repositorio y la WebApi.

La principal razón por la ausencia de pruebas unitarias exhaustivas es haber terminado de desarrollar los controladores y el repositorio cerca de la fecha de entrega, lo cual dejó poco tiempo para realizar una cobertura completa con pruebas unitarias.
Por otro para las clases que sí fueron testeadas en forma exhaustiva las mismas se testean con test unitario con la utilización de mocks y además con test de integración, somos conscientes que las pruebas son una pieza fundamental del software pero como mencionamos anteriormente por temas de tiempo.

Deploy

En este diagrama podemos ver la arquitectura en tiempo de ejecución de nuestro sistema.
Para que el sistema inicie debemos iniciar el servicio de Windows de Microsoft SQL Server (MSSQLSERVER)

Diagrama de deploy


pkg
<<Device>>
HTTP Server
<<<<OS>>>>
Microsoft Windows
<<Web Server>>
IIS
<<Librery>>
Logs
<<Librery>>
BussinesLogic
<<Librery>>
Domain
<<Web Application>>
Web Api
<<Librery>>
Data Access
<<HTTP>>
<<TCP/IP>>
<<Device>>
Client
HTTP Client
<<Device>>
Data Server
<<<<OS>>>>
Microsoft Windows
<<<<Sql Server Express>>>>
Server Name