Universidad ORT Uruguay

Facultad de Ingeniería

Trabajo Obligatorio

Diseño de Aplicaciones 2

Ramiro Gonzalez - 167011

Brahian Calo - 170540

Grupos M6A y N6A - Junio 2019

Indice

Descripción general del trabajo

Para esta entrega se completó el desarrollo de la arquitectura cliente-servidor para la aplicación IndicatorsManagerApp.

En primer lugar se terminó de implementar la aplicación Web Api mejorando aspectos de arquitectura, alineandonos a las buenas prácticas de APIs REST. La API continuó siendo desarrollada con las tecnologías como .NET Core, Entity Framework Core, Sql Server.

Por otro parte, se desarrolló en paralelo la aplicación cliente la cual fue desarrollada utilizando las tecnologías Angular 7 como framework javascript para realizar SPAs (single page applications), el paquete node Angular Materials 4, para manejar eventos complejos mejorando la usabilidad de la interfaz gráfica y paquete node Boostrap 4 para aplicar estilos a la interfaz gráfica de forma simple.

Nuestra Web Api sigue un patrón de arquitectura layer el cual está conformado por las siguientes capas:

- Acceso a Datos: esta capa se encarga de manipular la base de datos usando Entity Framework Core como ORM (Object-Relational Mapping).
- Lógica del Negocio: esta capa se encarga de las validaciones para las diferentes entidades del negocio y a su vez de una de las funcionalidades más interesantes del trabajo que es la Evaluación de Condiciones.
- Web Api: esta capa se encarga simplemente de capturar las Request Http que son enviadas al servidor y realizar la llamada correspondiente a un método la Lógica del Negocio, a su vez se encarga de capturar alguna excepción que levante la Lógica del Negocio y retornar el error correspondiente en la Response. A su vez se encarga de convertir las entidades del Dominio en Modelos que son utilizados como retorno o entrada en formato JSON.

La interfaz gráfica presenta una gran usabilidad y accesibilidad a la hora de usar la misma. Cuenta con diversos mecanismos sumamente intuitivos de interacción y es placentera de usar. Se sigue una a misma línea de diseños a lo largo de toda la aplicación convirtiéndola en un excelente servicio provisto por los desarrolladores.

Bugs Conocidos:

- Para importar indicadores, se debe tener archivos en la ruta especificada.
- En el momento de importacion, si la ruta especificada no tiene permisos, retorna un error de servidor.
- No se implementaron mensajes de confirmación en la UI, solamente mensajes de error.
- Las condiciones deben ser creadas lentamente dado un delay no controlado de eventos

Descripción diseño propuesto

Diagrama de Paquetes

En términos generales la solución está dividida en varios paquetes para poder separar la responsabilidades de cada uno. Nuestra solución está basada en el Patrón de Arquitectura Layer, donde se pueden distinguir las siguientes capas:

Capa de Acceso a Datos

- IndicatorsManager.DataAccess.Interface
- IndicatorsManagerDataAccess.Interface.Exceptions
- IndicatorsManager.DataAccess
- IndicatorsManager.DataAccess.Visitors

Capa de Logica de Negocio

- IndicatorsManager.BusinessLogic.Interface
- IndicatorsManager.BusinessLogic.Interface.Exceptions
- IndicatorsManager.BusinessLogic
- IndicatorsManager.BusinessLogic.Visitors

Capa de Web Api

- IndicatorsManager.WebApi
- IndicatorsManager.WebApi.Controllers
- IndicatorsManager.WebApi.Models
- IndicatorsManager.WebApi.Exceptions
- IndicatorsManager.WebApi.Filters
- IndicatorsManager.WebApi.Parsers
- IndicatorsManager.WebApi.Visitors

Las capas de Acceso a Datos y Lógica de Negocio son muy similares, tienen un paquete que expone las interfaces de la capa y las excepciones que que tira dicha capa, así como otro paquete que implementa las interfaces. En ambos casos el paquete de los visitors se agrupan las clases Visitor que las implementaciones utilizan y cualquier otra clase utilizada exclusivamente por los visitors.

Web Api por otro lado es un paquete sumamente concreto que se encarga de manejar la request que llegan al servidor.

También existen otros paquetes que no forman parte de las capas mencionadas arriba como son:

- IndicatorsManager.Domain: tiene las clases del Dominio.
- IndicatorsManager.Domain.Visitors: tiene la Interfaz del Visitor
- IndicatorsManager.Logger.Interface: tiene la interfaz para loggear acciones del sistema
- IndicatorsManager.Logger.Interface.Exceptions: tiene las excepciones que tiran las implementaciones del Logger.
- .IndicatorsManager.Logger.Database: tiene la implementación del Logger para Base de datos
- IndicatorManager.IndicatorImporter.Interface: tiene tanto la interfaz para importar importadores, así como la estructura que el importador concreto debe generar para que el sistema pueda crear los importadores.
- IndicatorManager.IndicatorImporter.Interface.Exception: tiene las excepciones que tiran los Importadores concretos
- IndicatorManager.IndicatorImporter.Json: tiene la implementación de la interfaz de Importadores específica para Json.
- IndicatorManager.IndicatorImporter.Xml: tiene la implementación de la interfaz de Importadores específica para Xml.

Algunos paquetes tienen subpaquetes, como por ejemplo IndicatorsManager.BusinessLogic y su subpaquete IndicatorsManager.BusinessLogic.Visitors, con esto se logró separar las clases que implementan las interfaces de la lógica de las clases utilizadas en el visitor.

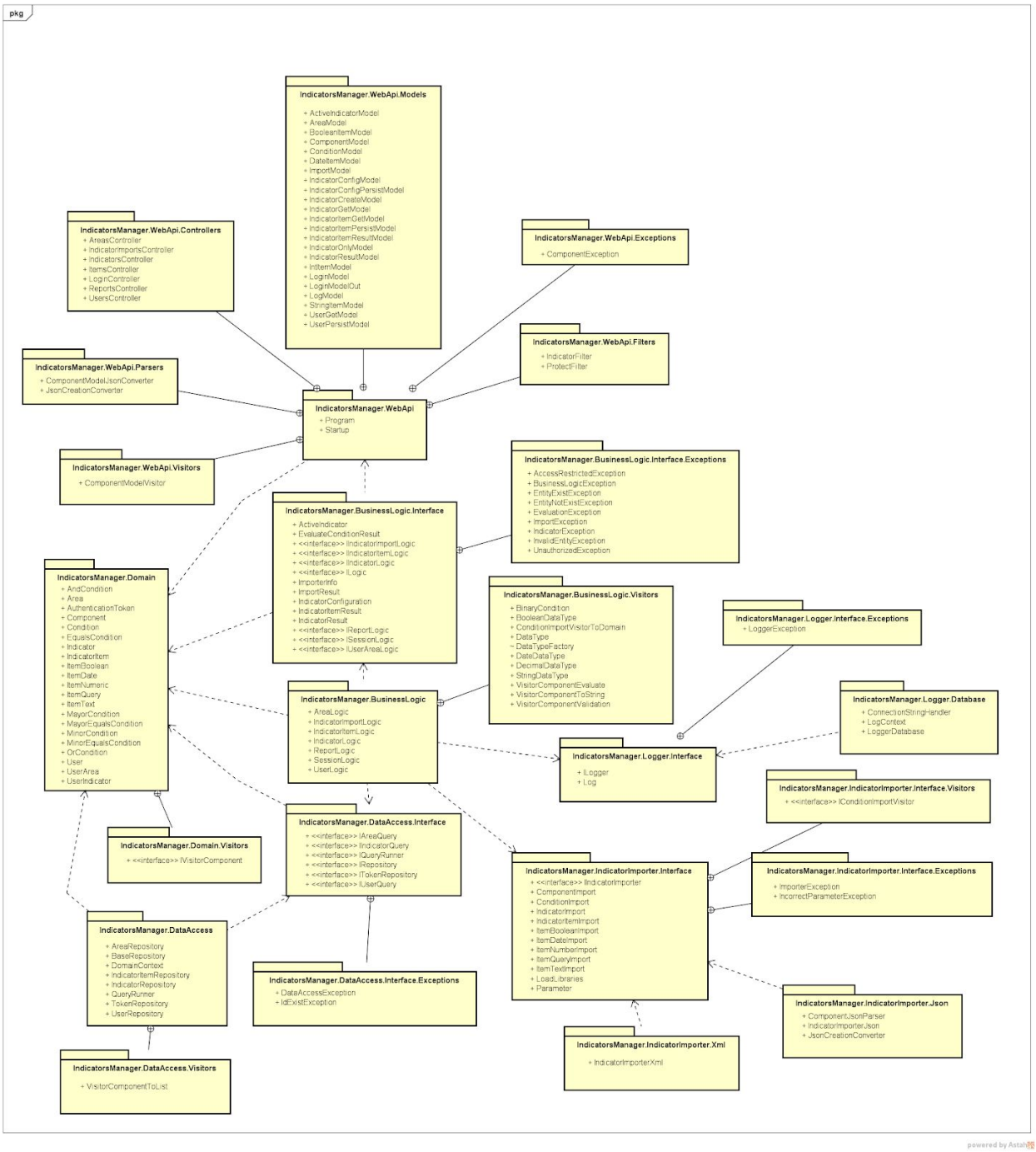
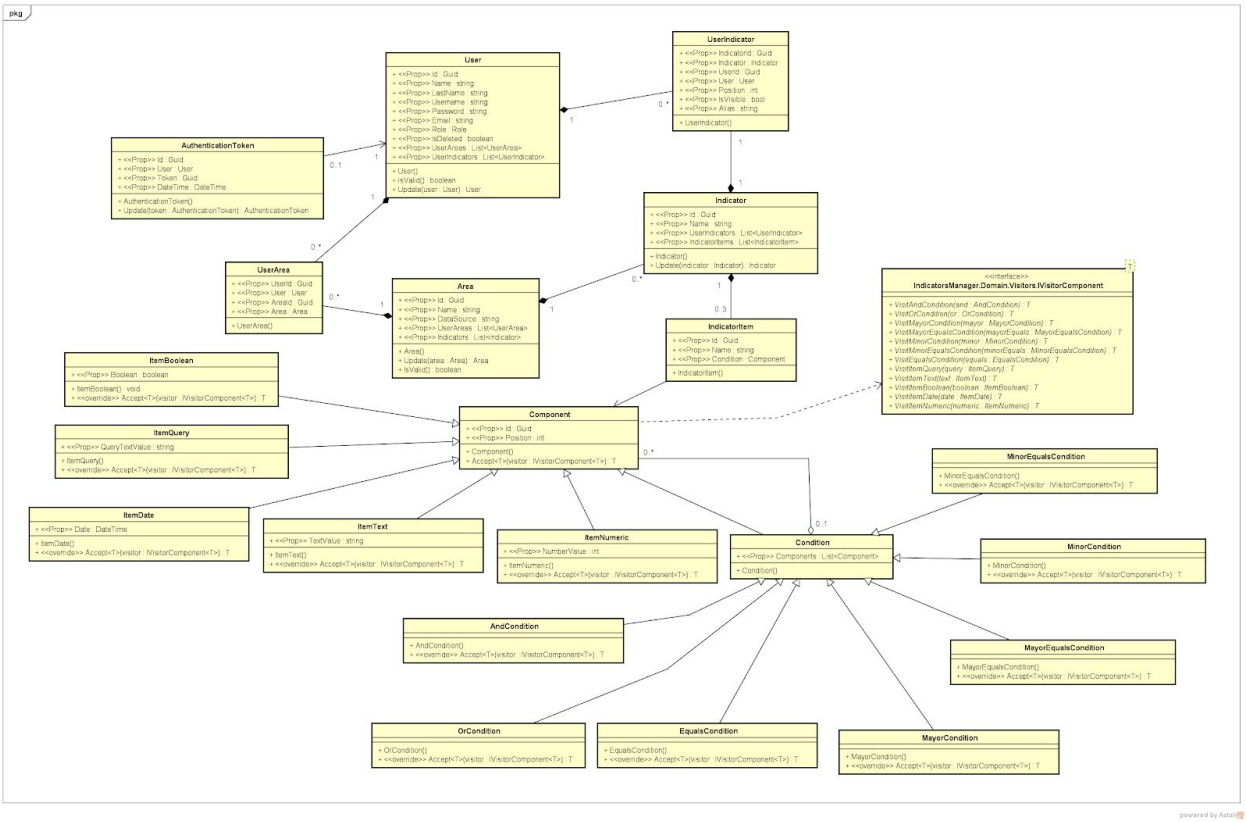
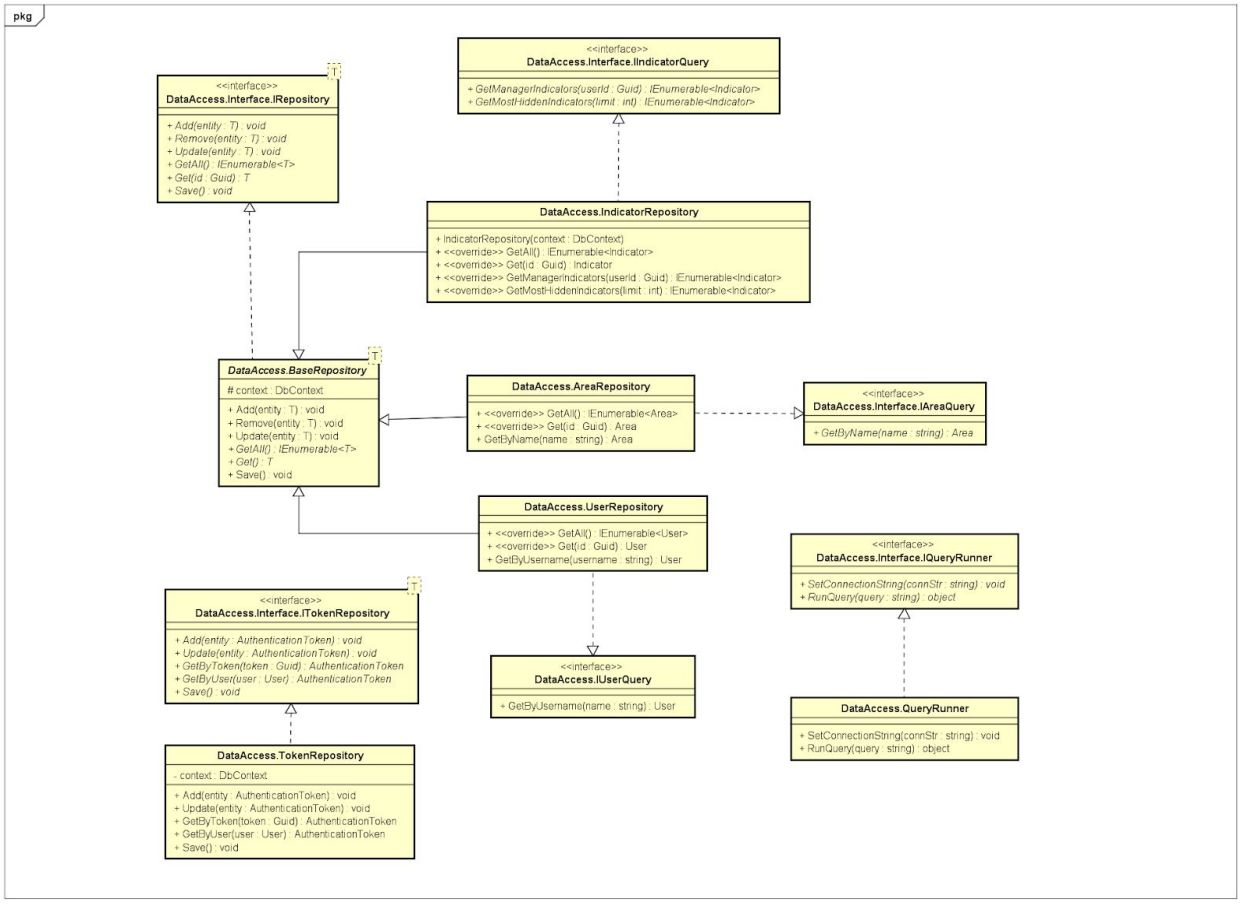

pkg
IndicatorsManager.WebApi.Models
+ ActiveIndicatorModel
+ AreaModel
+ BooleanItemModel
+ ComponentModel
+ ConditionModel
+ DateItemModel
+ ImportModel
+ IndicatorConfigModel
+ IndicatorConfigPersistModel
+ IndicatorCreateModel
+ IndicatorGetModel
+ IndicatorItemGetModel
+ IndicatorItemPersistModel
+ IndicatorItemResultModel
+ IndicatorOnlyModel
+ IndicatorResultModel
+ IntItemModel
+ LoginModel
+ LoginModelOut
+ LogModel
+ StringItemModel
+ UserGetModel
+ UserPersistModel
IndicatorsManager.WebApi.Controllers
+ AreasController
+ IndicatorImportsController
+ IndicatorsController
+ ItemsController
+ LoginController
+ ReportsController
+ UsersController
IndicatorsManager.WebApi.Exceptions
+ ComponentException
IndicatorsManager.WebApi.Filters
+ IndicatorFilter
+ ProtectFilter
IndicatorsManager.WebApi.Parsers
+ ComponentModelJsonConverter
+ JsonCreationConverter
IndicatorsManager.WebApi
+ Program
+ Startup
IndicatorsManager.WebApi.Visitors
+ ComponentModelVisitor
IndicatorsManager.BusinessLogic.Interface.Exceptions
+ AccessRestrictedException
+ BusinessLogicException
+ EntityExistException
+ EntityNotExistException
+ EvaluationException
+ ImportException
+ IndicatorException
+ InvalidEntityException
+ UnauthorizedException
IndicatorsManager.BusinessLogic.Interface
+ ActiveIndicator
+ EvaluateConditionResult
+ <<interface>> IIndicatorImportLogic
+ <<interface>> IIndicatorItemLogic
+ <<interface>> IIndicatorLogic
+ <<interface>> ILogic
+ ImporterInfo
+ ImportResult
+ IndicatorConfiguration
+ IndicatorItemResult
+ IndicatorResult
+ <<interface>> IReportLogic
+ <<interface>> ISessionLogic
+ <<interface>> IUserAreaLogic
IndicatorsManager.Domain
+ AndCondition
+ Area
+ AuthenticationToken
+ Component
+ Condition
+ EqualsCondition
+ Indicator
+ IndicatorItem
+ ItemBoolean
+ ItemDate
+ ItemNumeric
+ ItemQuery
+ ItemText
+ MayorCondition
+ MayorEqualsCondition
+ MinorCondition
+ MinorEqualsCondition
+ OrCondition
+ User
+ UserArea
+ UserIndicator
IndicatorsManager.BusinessLogic.Visitors
+ BinaryCondition
+ BooleanDataType
+ ConditionImportVisitorToDomain
+ DataType
~ DataTypeFactory
+ DateDataType
+ DecimalDataType
+ StringDataType
+ VisitorComponentEvaluate
+ VisitorComponentToString
+ VisitorComponentValidation
IndicatorsManager.Logger.Interface.Exceptions
+ LoggerException
IndicatorsManager.BusinessLogic
+ AreaLogic
+ IndicatorImportLogic
+ IndicatorItemLogic
+ IndicatorLogic
+ ReportLogic
+ SessionLogic
+ UserLogic
IndicatorsManager.Logger.Database
+ ConnectionStringHandler
+ LogContext
+ LoggerDatabase
IndicatorsManager.Logger.Interface
+ ILogger
+ Log
IndicatorsManager.IndicatorImporter.Interface.Visitors
+ <<interface>> IConditionImportVisitor
IndicatorsManager.Domain.Visitors
+ <<interface>> IVisitorComponent
IndicatorsManager.DataAccess.Interface
+ <<interface>> IAreaQuery
+ <<interface>> IIndicatorQuery
+ <<interface>> IQueryRunner
+ <<interface>> IRepository
+ <<interface>> ITokenRepository
+ <<interface>> IUserQuery
IndicatorsManager.IndicatorImporter.Interface
+ <<interface>> IIndicatorImporter
+ ComponentImport
+ ConditionImport
+ IndicatorImport
+ IndicatorItemImport
+ ItemBooleanImport
+ ItemDateImport
+ ItemNumberImport
+ ItemQueryImport
+ ItemTextImport
+ LoadLibraries
+ Parameter
IndicatorsManager.IndicatorImporter.Interface.Exceptions
+ ImporterException
+ IncorrectParameterException
IndicatorsManager.DataAccess
+ AreaRepository
+ BaseRepository
+ DomainContext
+ IndicatorItemRepository
+ IndicatorRepository
+ QueryRunner
+ TokenRepository
+ UserRepository
IndicatorsManager.DataAccess.Interface.Exceptions
+ DataAccessException
+ IdExistException
IndicatorsManager.IndicatorImporter.Json
+ ComponentJsonParser
+ IndicatorImporterJson
+ JsonCreationConverter
IndicatorsManager.IndicatorImporter.Xml
+ IndicatorImporterXml
IndicatorsManager.DataAccess.Visitors
+ VisitorComponentToList
powered by Astah

Diagrama de Clases

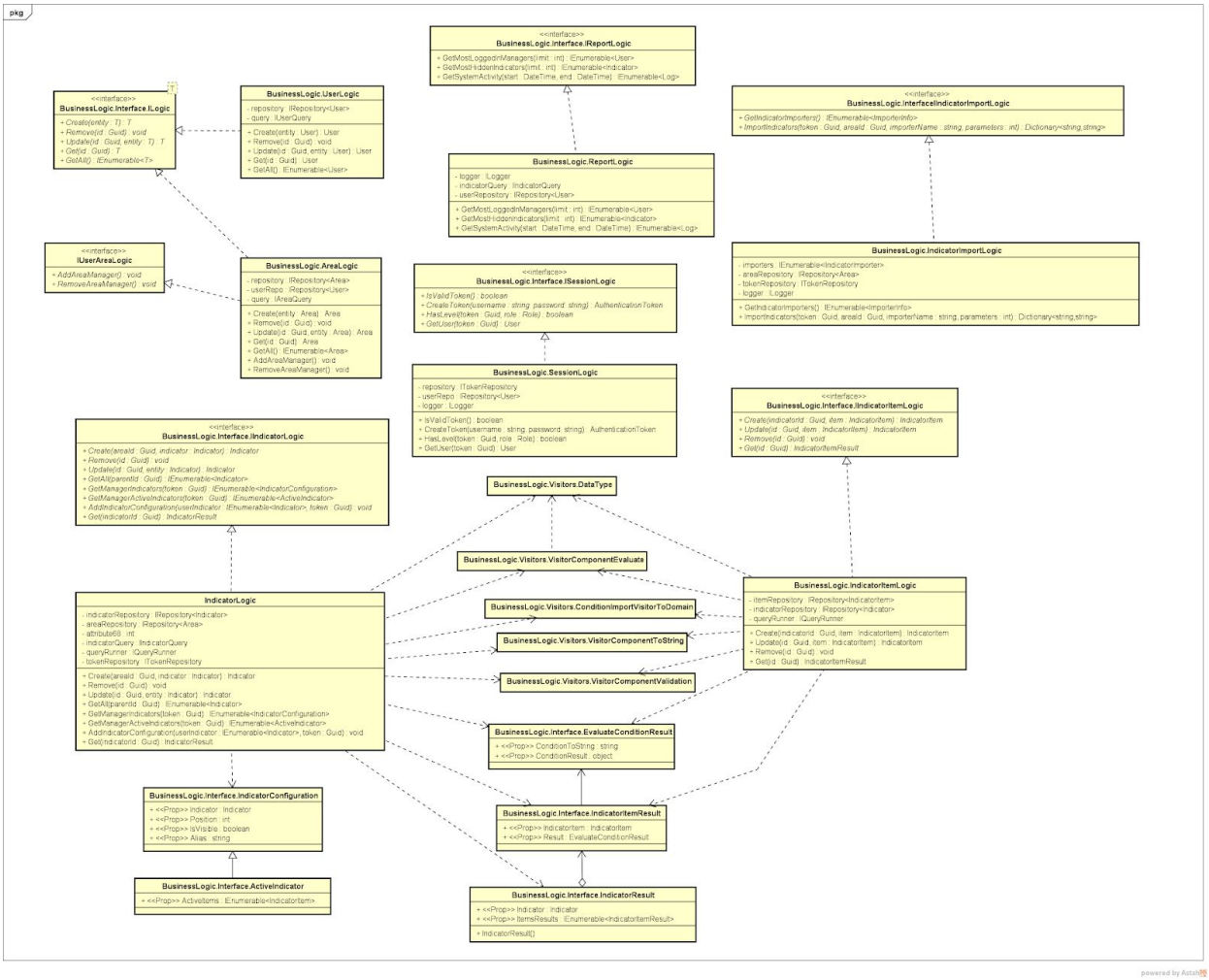
Dominio



Acceso a Datos

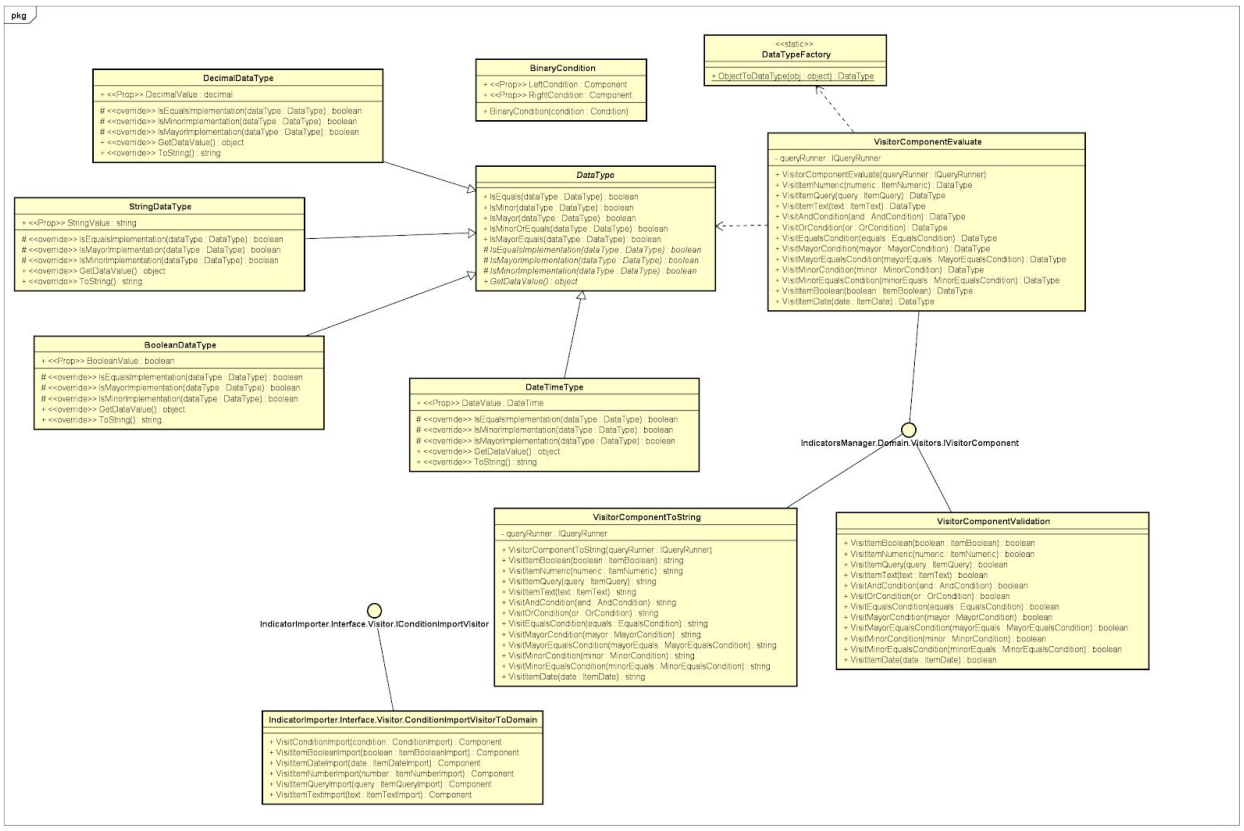


Business Logic

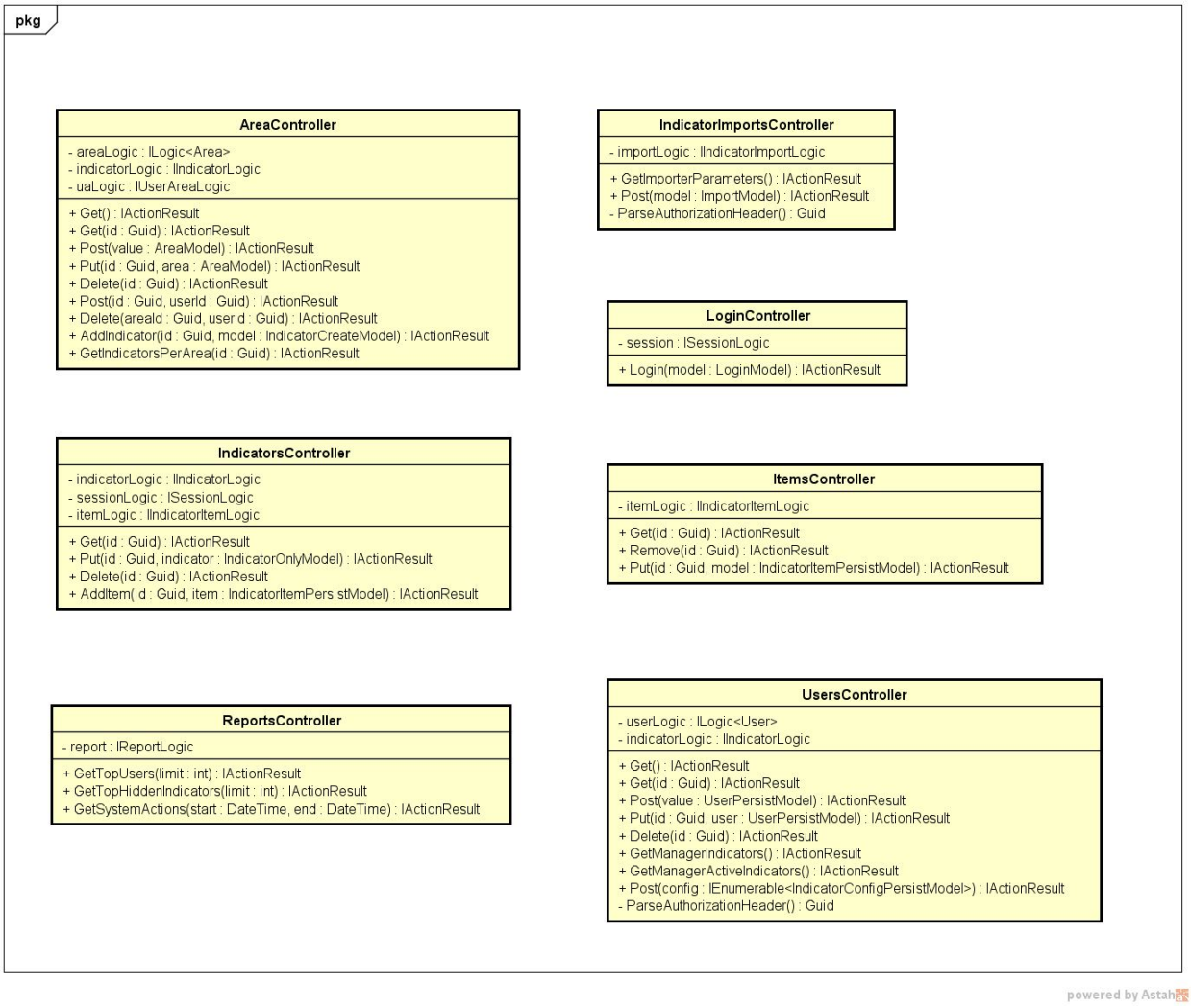


BusinessLogic Visitors

Este paquete se encuentra en el namespace IndicatorsManager.BusinessLogic.Visitors, este paquete agrupa los visitors implementados en la lógica así como las clases auxiliares utilizadas por el Visitor que evalúa condiciones (VisitorComponentEvaluate)


pkg
DecimalDataType
+ <<Prop>> DecimalValue : decimal
# <<override>> IsEqualsImplementation(dataType : DataType) : boolean
# <<override>> IsMinorImplementation(dataType : DataType) : boolean
# <<override>> IsMayorImplementation(dataType : DataType) : boolean
+ <<override>> GetDataValue() : object
+ <<override>> ToString() : string
BinaryCondition
+ <<Prop>> LeftCondition : Component
+ <<Prop>> RightCondition : Component
+ BinaryCondition(condition : Condition)
<<static>>
DataTypeFactory
+ ObjectToDataType(obj : object) : DataType
VisitorComponentEvaluate
- queryRunner : IQueryRunner
+ VisitorComponentEvaluate(queryRunner : IQueryRunner)
+ VisitItemNumeric(numeric : ItemNumeric) : DataType
+ VisitItemQuery(query : ItemQuery) : DataType
+ VisitItemText(text : ItemText) : DataType
+ VisitAndCondition(and : AndCondition) : DataType
+ VisitOrCondition(or : OrCondition) : DataType
+ VisitEqualsCondition(equals : EqualsCondition) : DataType
+ VisitMayorCondition(mayor : MayorCondition) : DataType
+ VisitMayorEqualsCondition(mayorEquals : MayorEqualsCondition) : DataType
+ VisitMinorCondition(minor : MinorCondition) : DataType
+ VisitMinorEqualsCondition(minorEquals : MinorEqualsCondition) : DataType
+ VisitItemBoolean(boolean : ItemBoolean) : DataType
+ VisitItemDate(date : ItemDate) : DataType
StringDataType
+ <<Prop>> StringValue : string
# <<override>> IsEqualsImplementation(dataType : DataType) : boolean
# <<override>> IsMayorImplementation(dataType : DataType) : boolean
# <<override>> IsMinorImplementation(dataType : DataType) : boolean
+ <<override>> GetDataValue() : object
+ <<override>> ToString() : string
DataType
+ IsEquals(dataType : DataType) : boolean
+ IsMinor(dataType : DataType) : boolean
+ IsMayor(dataType : DataType) : boolean
+ IsMinorOrEquals(dataType : DataType) : boolean
+ IsMayorEquals(dataType : DataType) : boolean
# IsEqualsImplementation(dataType : DataType) : boolean
# IsMayorImplementation(dataType : DataType) : boolean
# IsMinorImplementation(dataType : DataType) : boolean
+ GetDataValue() : object
BooleanDataType
+ <<Prop>> BooleanValue : boolean
# <<override>> IsEqualsImplementation(dataType : DataType) : boolean
# <<override>> IsMayorImplementation(dataType : DataType) : boolean
# <<override>> IsMinorImplementation(dataType : DataType) : boolean
+ <<override>> GetDataValue() : object
+ <<override>> ToString() : string
DateTimeType
+ <<Prop>> DateValue : DateTime
# <<override>> IsEqualsImplementation(dataType : DataType) : boolean
# <<override>> IsMinorImplementation(dataType : DataType) : boolean
# <<override>> IsMayorImplementation(dataType : DataType) : boolean
+ <<override>> GetDataValue() : object
+ <<override>> ToString() : string
IndicatorsManager.Domain.Visitors.IVisitorComponent
VisitorComponentToString
- queryRunner : IQueryRunner
+ VisitorComponentToString(queryRunner : IQueryRunner)
+ VisitItemBoolean(boolean : itemBoolean) : string
+ VisitItemNumeric(numeric : itemNumeric) : string
+ VisitItemQuery(query : ItemQuery) : string
+ VisitItemText(text : ItemText) : string
+ VisitAndCondition(and : AndCondition) : string
+ VisitOrCondition(or : OrCondition) : string
+ VisitEqualsCondition(equals : EqualsCondition) : string
+ VisitMayorCondition(mayor : MayorCondition) : string
+ VisitMayorEqualsCondition(mayorEquals : MayorEqualsCondition) : string
+ VisitMinorCondition(minor : MinorCondition) : string
+ VisitMinorEqualsCondition(minorEquals : MinorEqualsCondition) : string
+ VisitItemDate(date : ItemDate) : string
VisitorComponentValidation
+ VisitItemBoolean(boolean : ItemBoolean) : boolean
+ VisitItemNumeric(numeric : itemNumeric) : boolean
+ VisitItemQuery(query : ItemQuery) : boolean
+ VisitItemText(text : itemText) : boolean
+ VisitAndCondition(and : AndCondition) : boolean
+ VisitOrCondition(or : OrCondition) : boolean
+ VisitEqualsCondition(equals : EqualsCondition) : boolean
+ VisitMayorCondition(mayor : MayorCondition) : boolean
+ VisitMayorEqualsCondition(mayorEquals : MayorEqualsCondition) : boolean
+ VisitMinorCondition(minor : MinorCondition) : boolean
+ VisitMinorEqualsCondition(minorEquals : MinorEqualsCondition) : boolean
+ VisitItemDate(date : ItemDate) : boolean
IndicatorImporter.Interface.Visitor.IConditionImportVisitor
IndicatorImporter.Interface.Visitor.ConditionImportVisitorToDomain
+ VisitConditionImport(condition : ConditionImport) : Component
+ VisitItemBooleanImport(boolean : itemBooleanImport) : Component
+ VisitItemDateImport(date : itemDateImport) : Component
+ VisitItemNumberImport(number : itemNumberImport) : Component
+ VisitItemQueryImport(query : itemQueryImport) : Component
+ VisitItemTextImport(text : itemTextImport) : Component
powered by Astah

Web Api Controller



IndicatorsImporter

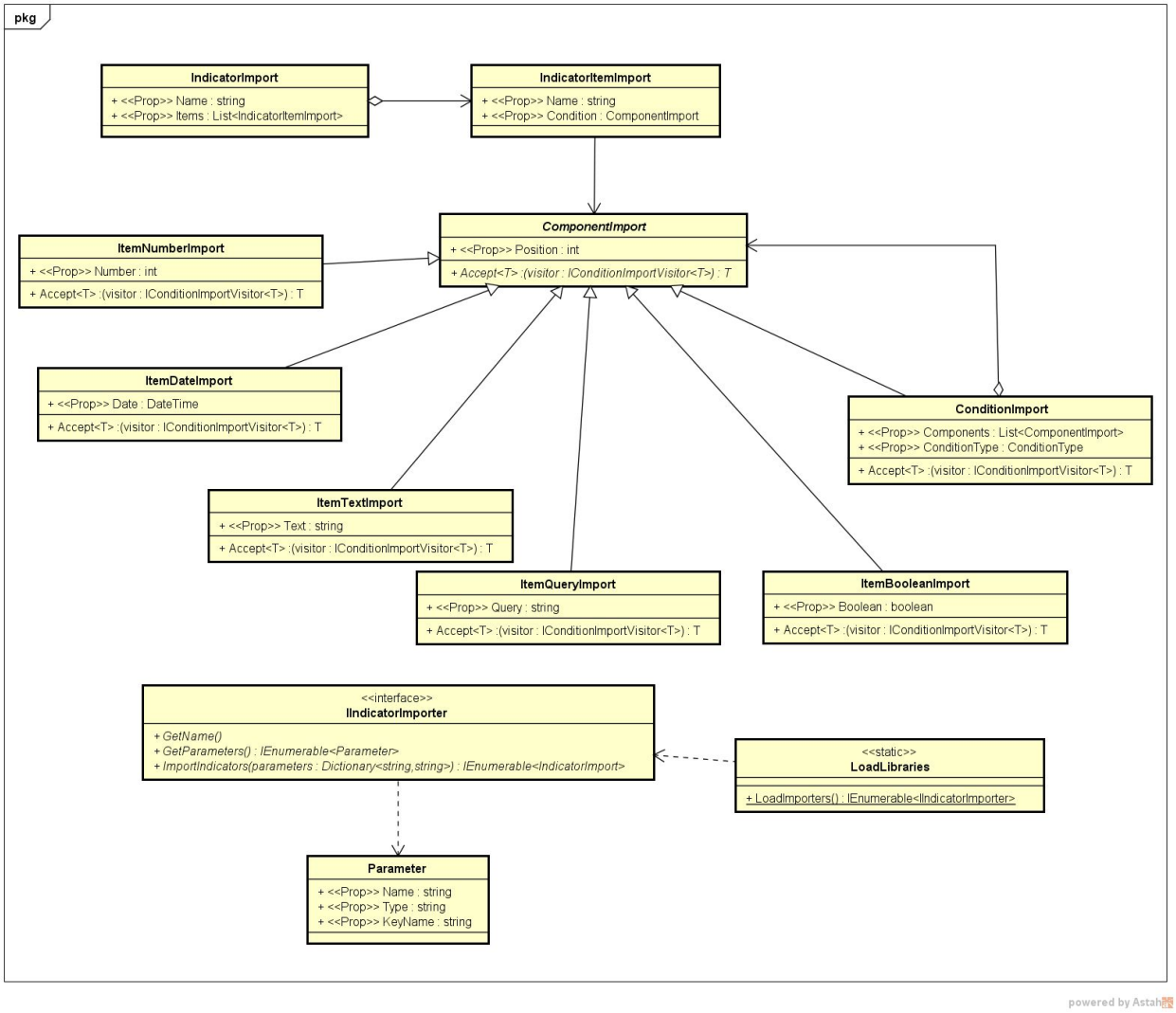


Diagrama de Componentes

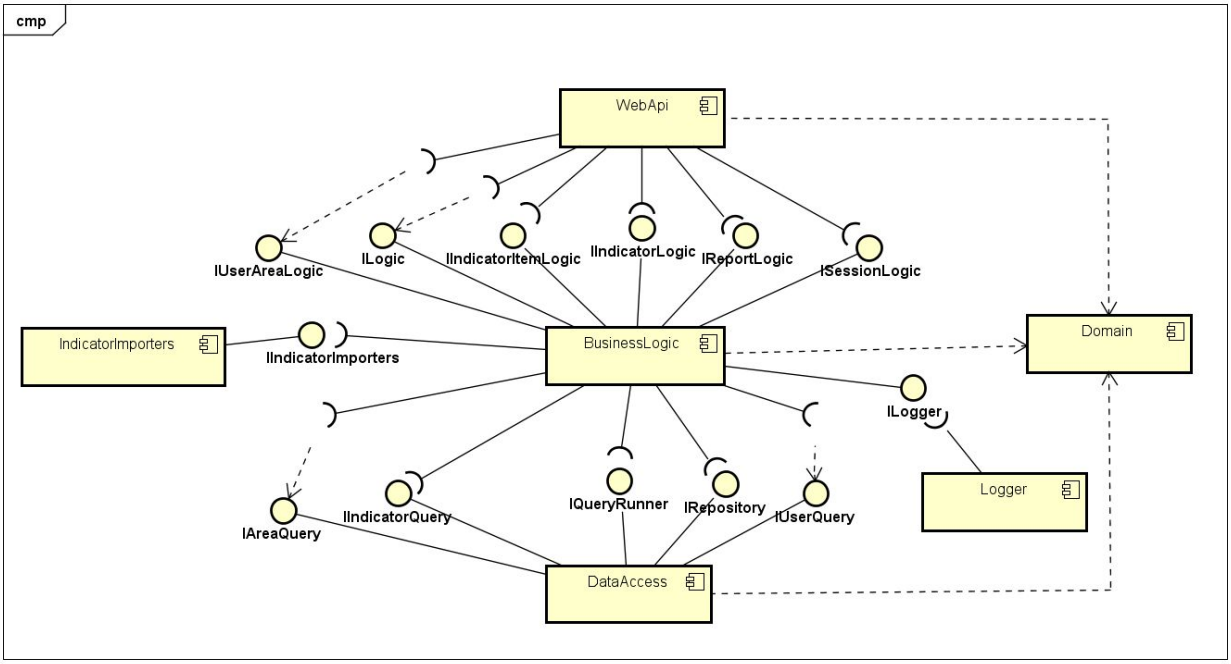
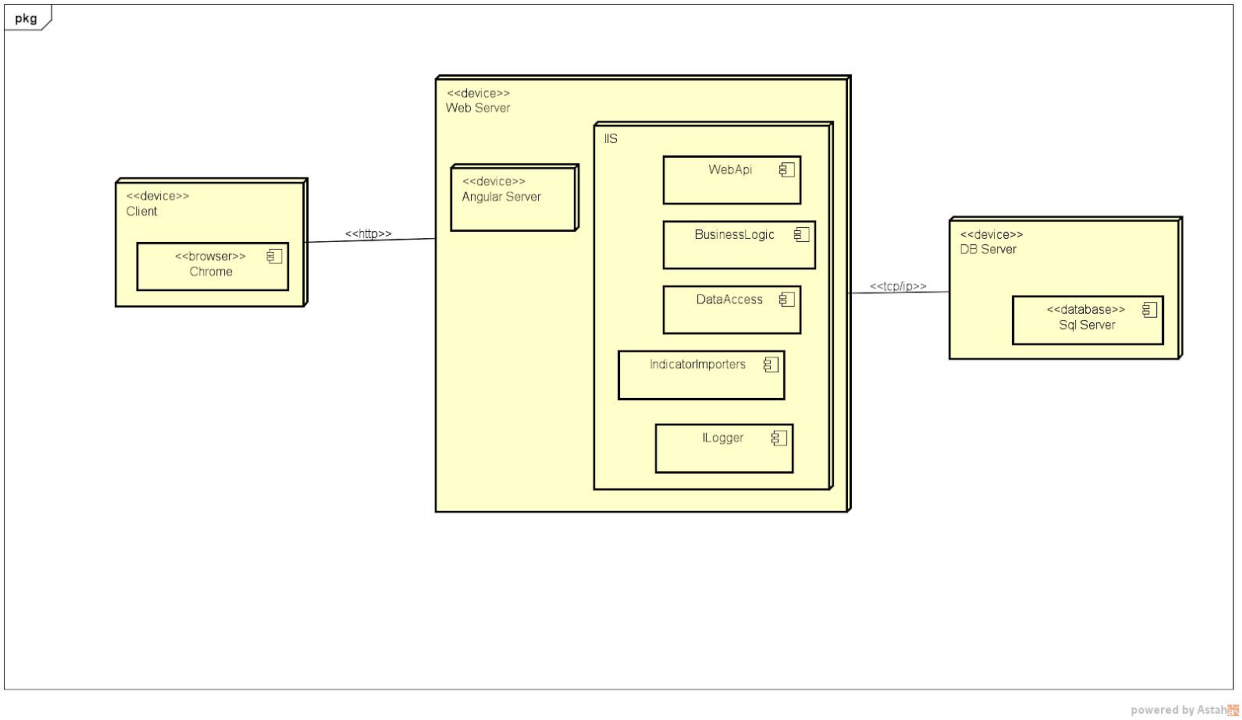


Diagrama de Despliegue



Justificación de las decisiones de diseño tomadas

Patrones de diseño utilizados

Uno de los patrones utilizados fue el Composite, este patrón nos permitió representar las condiciones del dominio definida en los requerimientos.
A su vez cada tipo de condición (Ej.: Equals) se representó como subtipos de la clase Condition(Ver en diagrama de clases del dominio). Esto nos permitió aprovechar el mecanismo de polimorfismo y otorgarnos extensibilidad en caso de nuevos requerimientos de condiciones.
En complemento al Composite se utilizó el patrón Visitor para poder extraer el código recursivo del composite hacia afuera del dominio. La razón principal de esto fue para evitar dependencias desde el dominio hacia cualquier otro paquete de la solución (Ej.: La lógica de ejecutar consultas). Esto es muy importante ya que el Paquete del Dominio es el más estable de nuestra solución.
Los visitor implementados necesitaban retornar objetos, esto era super importante debido a la recursividad del Composite, por esta razón la Interfaz de IVisitorComponent es de tipo Generic, permitiéndonos implementar diferentes Visitors en base al tipo de retorno deseado para cada implementación, evitando repetir la misma interfaz para cada tipo de dato deseado.

Nuevos Requerimientos

Para esta segunda entrega se pidió agregar dos nuevos tipos de elementos para las condiciones (Fechas y Boolean). En el dominio se agregaron dos nuevas clases (ItemBoolean y ItemDate) que extienden directamente de Component (clase abstracta del composite). Ambas clases tienen que implementar el método abstracto Accept, por esta razón se tuvo que agregar dos nuevos métodos a la interfaz IVisitorComponent.
Al agregar nuevos métodos a la interfaz se tuvo que agregar la implementación de dichos métodos a cada Implementación existente de dicha interfaz (5 clases que implementan la interfaz).

Otro cambio solicitado fue que un gerente pueda asignar un Alias a un Indicador determinado. Este cambio fue muy menor, ya que implicó agregar una nueva propiedad a la clase UserIndicator que se usa para guardar la configuración de indicadores por gerente.
Al ser un cambio de dominio afectó tanto a la capa lógica como a la capa de web api, pero ambos cambios fueron mínimos.

Otro requerimiento implementado fue el de mover la lógica del log hacia afuera de la solución, para esto simplemente se crearon dos proyectos nuevos (Interface.Logger y Logger.Database). Como en los otros proyectos de la solución, los que usan el Log lo

reciben por el mecanismo de inyección de dependencia que provee el framework. Un detalle importante es cómo se asigna el connection string, a través de un singleton que es asignado en el startup de la web api.

Por último se pidió agregar una nueva funcionalidad, la habilidad de importar Indicadores en diferentes formatos. Al ser una nueva funcionalidad tuvo un impacto casi nulo en el sistema. Para su implementación se pudo reutilizar código ya existente, como el visitor que verifica que una condición es válida.

Mejoras

Se aplicaron diversas mejoras al código entregado para el primer obligatorio, entre ellas se destacan las siguientes.

Evaluar Condiciones

Notamos que había mucho código repetido a través de cada uno de los métodos del visitor que evalúa condiciones y se hacía RTTI para identificar los tipos genéricos (Object).
La mejora fue crear una jerarquía de clases (DataType), la cual define varios métodos que nos permiten comparar dos DataType diferentes (mayor, menor igual, etc). Cada subtipo de DataType representa los diferentes tipos (Number, String, Date, etc) definidos en los requerimientos.
Para el caso de los resultados de las queries se creó una lógica que se encarga de identificar el tipo del objeto retornado y conectarlo con el subtipo de DataType más adecuado.

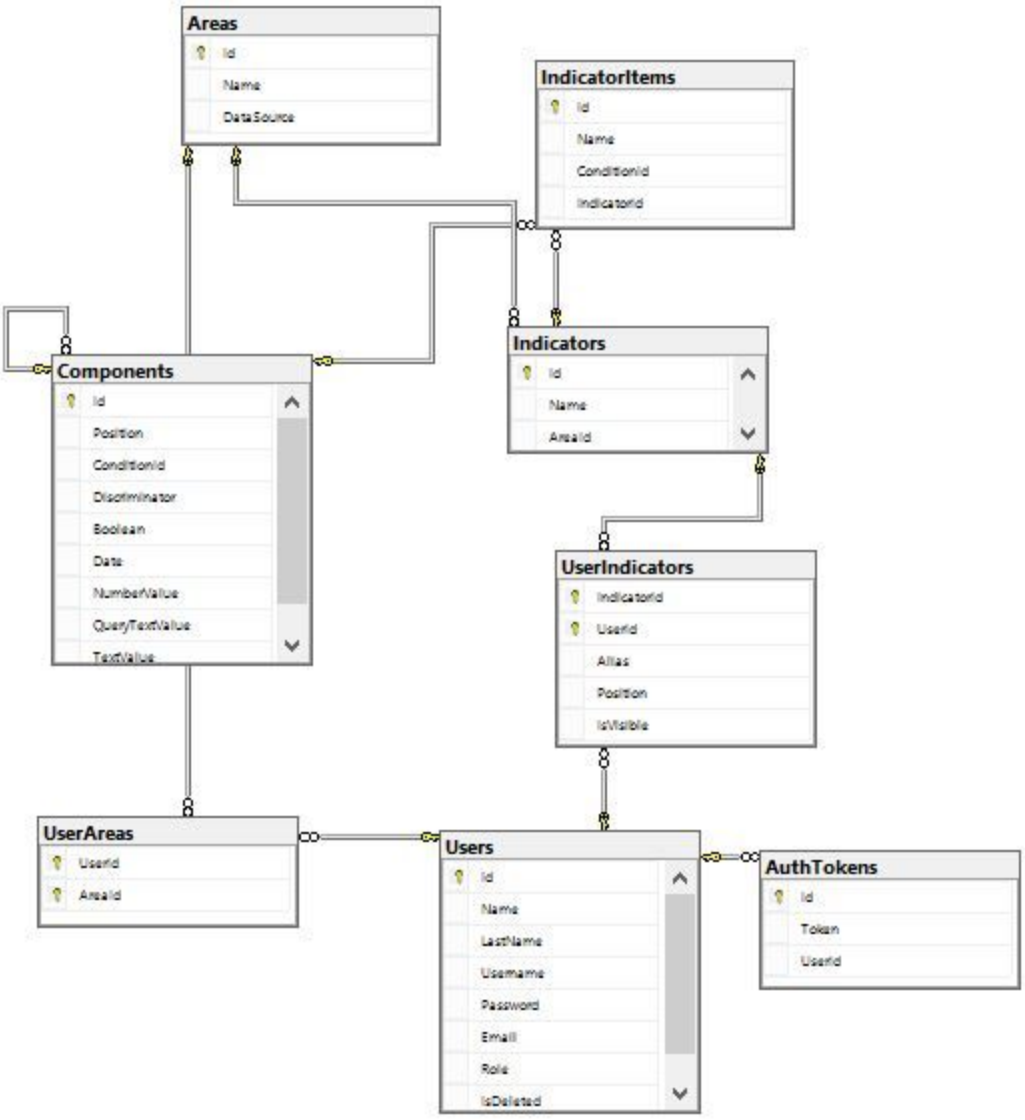
Manejo de errores de DataAccess

Algo que había quedado pendiente para la entrega anterior fue el manejo de errores de base de datos que ocurren si la base de datos no está disponible (servicio SQL server apagado) o el connection string era incorrecto, en caso de ocurrir dichos errores la Api responde con un 503 (servicio no disponible).

Reemplazo de strings por enums

Para la entrega anterior se utilizaron strings en varios modelos de la WebApi cuando la mejor opción era usar Enums, esto se debía a que por defecto los Enums se serializar a json con el valor numérico que representan realmente. Para la segunda entrega se descubrió una manera de devolver enums en string y se aplicó dicho cambio.

Modelo de tablas de la estructura de la base de datos



Informe de métricas sobre el diseño

Análisis de dependencias

Se buscó que módulos de alto nivel no dependan de modulos de bajo nivel sino de abstracciones estables, siguiendo el principio de Abstracciones Estables.

Es por esto que, por ejemplo, IndicatorsManager.BusinessLogic dependen de paquetes de interfaces como IndicatorsManager.DataAccess.Interface, IndicatorsManager.Logger.Interfaces o IndicatorsManager.IndicatorImporter.Interfaces y nunca de paquetes en donde se encuentran implementaciones concretas del mismo. En caso de que, por ejemplo, se modifique el acceso a datos, este cambio no impactará en la capa de servicios ya que ésta depende de una interfaz estable.

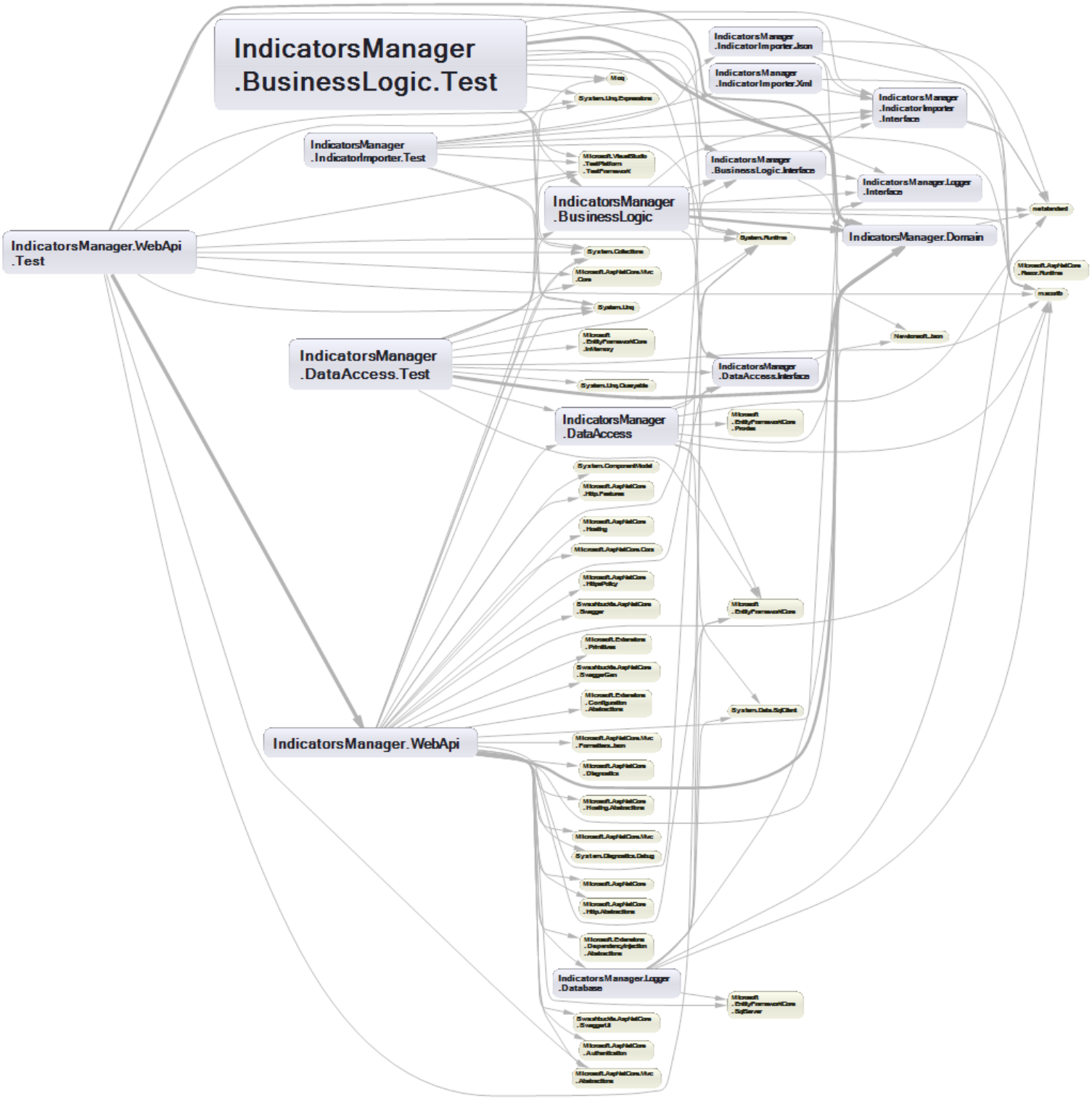
Decimos que se respeta el principio de clausura común ya que las clases pertenecientes a un paquete deben cambian por un mismo motivo de cambio, es decir, las agrupaciones lógicas encapsulan clases de mismo que tienen una responsabilidad similar, aquí nombramos como claro ejemplo los paquetes IndicatorsManager.Logger o IndicatorsManager.IndicatorImporter. Los cambios en estos paquetes afectan a solamente clases de sí mismos y a ningun otro paquete.

Por otra parte se respeta el principio de dependencias cíclicas dado que el Framework .NET no permite dichas dependencias.

Las dependencias entre paquetes de todo el proyecto van en el sentido de la estabilidad. Todos los paquete dependen solamente de paquetes que son más estables. El paquete IndicatorsManager.WebApi por ejemplo es el paquete mas inestable y como contrapartida, IndicatorsManager.Domain es el más estable.

En la siguiente imagen se pueden ver las dependencias entre proyectos, y como la dirección de las mismas va desde afuera (IndicatorsManager.WebApi) hacia adentro (IndicatorsManager.Domain), pasando por los servicios y luego por acceso a datos. (Nunca en la dirección contraria).

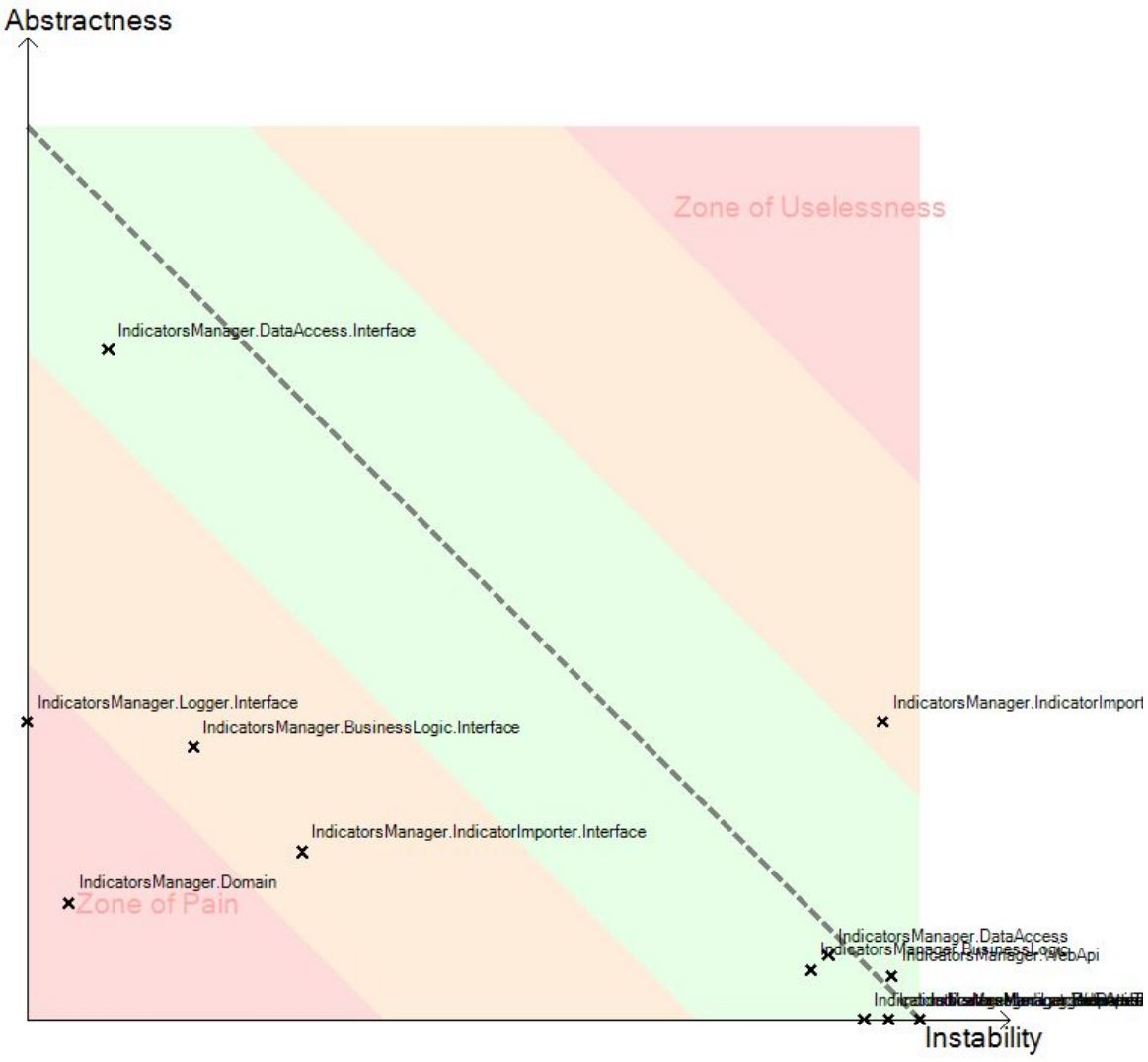
No supimos cómo ignorar para el análisis los paquetes System.* y Microsoft.* por lo que IndicatorsManager.WebApi tiene demasiadas dependencias salientes.



Inestabilidad vs. Abstracción

En este diagrama alcanzamos a ver como nuestra solucion cumple con que los paquetes que agrupan las clases concretas se encuentran el cuadrante de menor abstracción y mayor inestabilidad como lo son los paquetes IndicatorsManager.WebApi, IndicatorsManager.BussinessLogic, IndicatorsManager.DataAccess, IndicatorsManager.IndicatorsImporter.Json, IndicatorsManager.IndicatorsImporter.Xml y IndicatorsManager.Logger.Database.
(Los listamos debido a que la imagen superpuso dichos paquetes dado que todos son inestables).

Otro análisis relevante es que el paquete IndicatorsManager.Domain se encuentra en la zona de dolor ya que todos dependen de él y esto es totalmente válido.



Justificación de Clean Code

Nombre de las clases

Los nombres de las clases son descriptivos y no presentan ambigüedad, respetando el formato Upper Camel Case. Comenzamos cada clase con un sustantivo el cual ayuda al lector a comprender el motivo de existencia de dicha clase.

Nombre de los métodos

El nombre de los métodos comienzan con un verbo en infinitivo y al mismo tiempo estos sirven como palabra clave para anticipar el comportamiento del mismo.
Otra buena práctica que se llevó a cabo con los nombres de los métodos es que si se utiliza un prefijo para indicar algo como get o retrieve para la obtención de datos, se utilice el mismo en las diferentes clases para que de esta forma el desarrollador sepa que cuando se hace un get siempre va a ser para obtener algo y no que existan más de una misma forma de obtener datos.

Nombre de las interfaces

Si bien Clean Code no establece una norma clara sobre las buenas prácticas a la hora de nombrar nuestras interfaces, es una convención que las mismas comiencen con la letra mayúscula "I" seguida de su nombre haciendo correspondencia a una clase siendo escrita en formato Upper Camel Case.

Nombres de atributos, properties, variables y parametros

Los atributos y parámetros siguen la convención lower Camel Case mientras que para las properties se sigue el estándar utilizado en C# de escribirlas en formato Upper Camel Case. A su vez el nombre de los mismos es descriptivos sin llegar a ser excesivamente largos.

Excepciones

Se utilizan excepciones para controlar los diferentes escenarios de errores que pueden surgir en nuestro código. Muchas de estas excepciones que empleamos son personalizadas para hacer el error más entendible a la hora de mostrarlo.

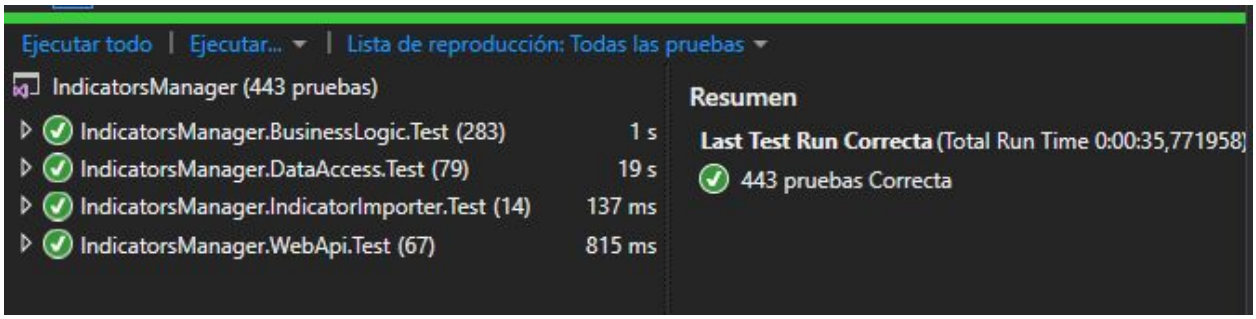
Resultado de Ejecución de Pruebas

El desarrollo fue basado en TDD (Test Driven Development) por lo que la solución del proyecto IndicatorsManager de lado del back-end posee un elevado número de pruebas realizadas.

Se llegó con un número sumamente elevado de pruebas lo cual es muy seguro introducir cualquier tipo de cambio dado que las funcionalidades van a ser respaldadas por los tests escritos anteriormente. Esto hace con que el equipo de desarrollo desarrolle un alto nivel de confianza al introducir un cambio ya que cuenta con documentación escrita en tests unitarios a la cual puede ir y validar si su nuevo código funciona.

No contamos con pruebas de integración ya que se optó por agregar más código unitario a las nuevas funcionalidades versus agregar pruebas de integración a código que estaba funcionando correctamente para la primera entrega. La falta de tiempo hizo con que tomaramos dicha decisión.

A continuación se muestra el resultado de la ejecución de las pruebas provistas por la herramienta integrada de Visual Studio 2017 Enterprise.



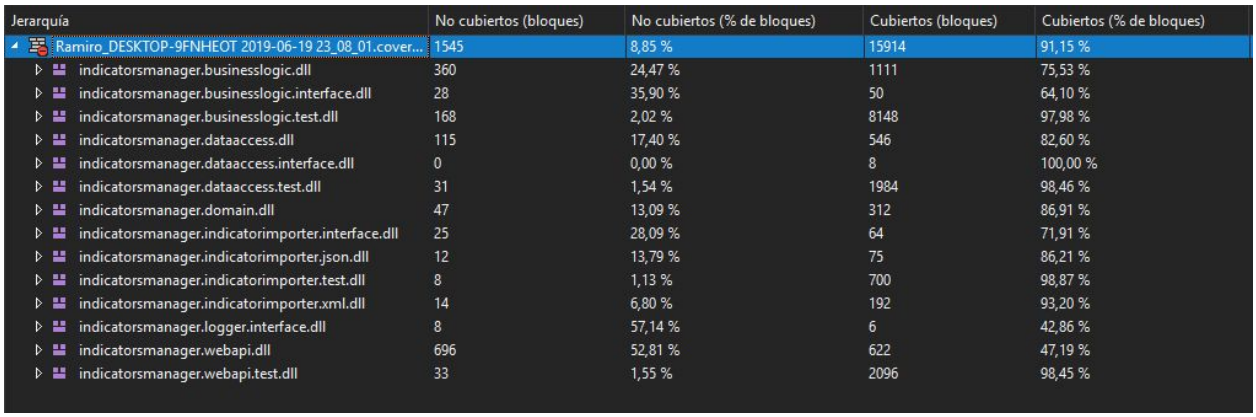
Como vemos en la imagen, el paquete de BusinessLogic es el que mas pruebas se le realizaron ya que allí es donde se encuentra toda la lógica de la aplicación.

El paquete IndicatorsManager.IndicatorImporter.Test tuvo un bajo nivel de cobertura pero se trato de abarcar los flujos más importantes.

El paquete de Logger no se le llegó realizar pruebas unitarias. Se decidió implementar la funcionalidad de extraerlo a un módulo aparte y dejar para un aspecto a mejorar a futuro la implementación de dichos tests.

La cobertura de código del proyecto en general es de un número muy elevado alcanzando un 91%. Se le dedico muchísimo esfuerzo a realizar los tests unitarios para cubrir la mayor cantidad de casos antes de comenzar a codificar.

Para destacar, el paquete de WebApi es el que tuvo menor cobertura de los paquetes concretos con un 47%, luego le sigue el paquete BusinessLogic con un 75%, DataAccess con un 85% y Domain con un 86%.



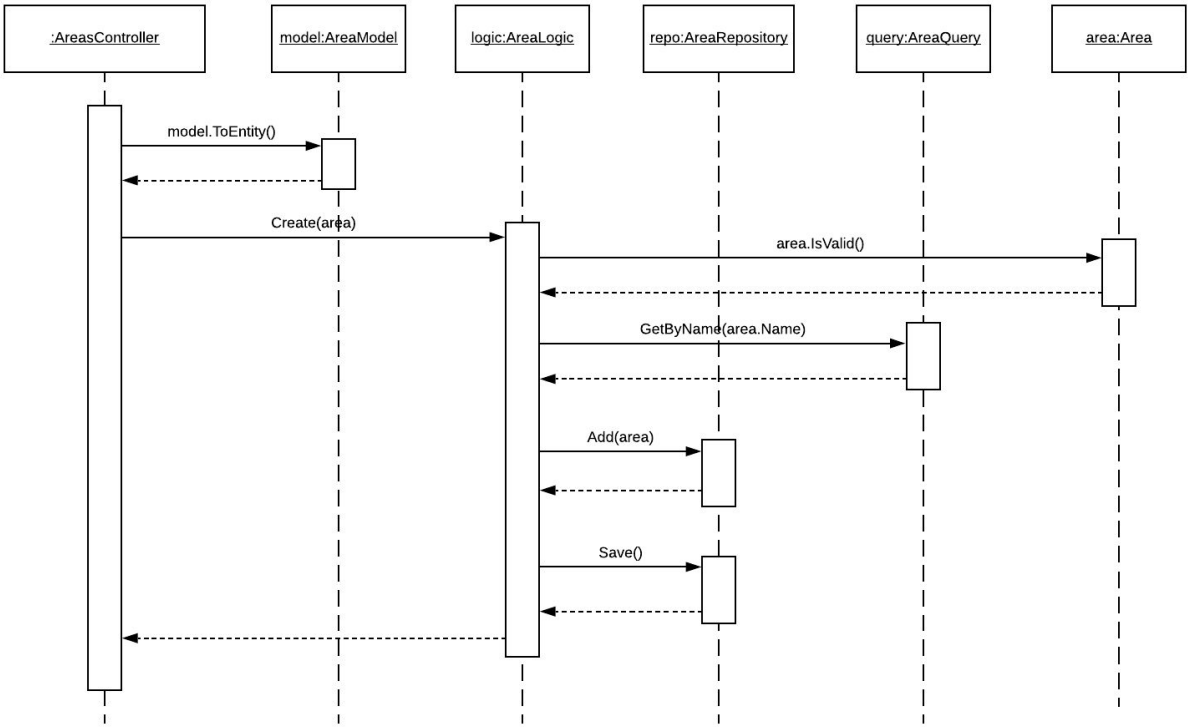
Jerarquía	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% de bloques)
Ramiro_DESKTOP-9FNHEOT 2019-06-19 23_08_01.cover...	1545	8,85 %	15914	91,15 %
indicatorsmanager.businesslogic.dll	360	24,47 %	1111	75,53 %
indicatorsmanager.businesslogic.interface.dll	28	35,90 %	50	64,10 %
indicatorsmanager.businesslogic.test.dll	168	2,02 %	8148	97,98 %
indicatorsmanager.dataaccess.dll	115	17,40 %	546	82,60 %
indicatorsmanager.dataaccess.interface.dll	0	0,00 %	8	100,00 %
indicatorsmanager.dataaccess.test.dll	31	1,54 %	1984	98,46 %
indicatorsmanager.domain.dll	47	13,09 %	312	86,91 %
indicatorsmanager.indicatorimporter.interface.dll	25	28,09 %	64	71,91 %
indicatorsmanager.indicatorimporter.json.dll	12	13,79 %	75	86,21 %
indicatorsmanager.indicatorimporter.test.dll	8	1,13 %	700	98,87 %
indicatorsmanager.indicatorimporter.xml.dll	14	6,80 %	192	93,20 %
indicatorsmanager.logger.interface.dll	8	57,14 %	6	42,86 %
indicatorsmanager.webapi.dll	696	52,81 %	622	47,19 %
indicatorsmanager.webapi.test.dll	33	1,55 %	2096	98,45 %

Como aspectos a mejorar, lo ideal sería dedicarle tiempo a cubrir la mayor cantidad de bloques de WebApi dado que con el correr del tiempo, se le fueron agregando distintos bloques de excepciones aumentando de esta forma el nivel de bloques sin cubrir. Pero las funcionalidades del flujo sin error aseguramos que están totalmente cubiertas.

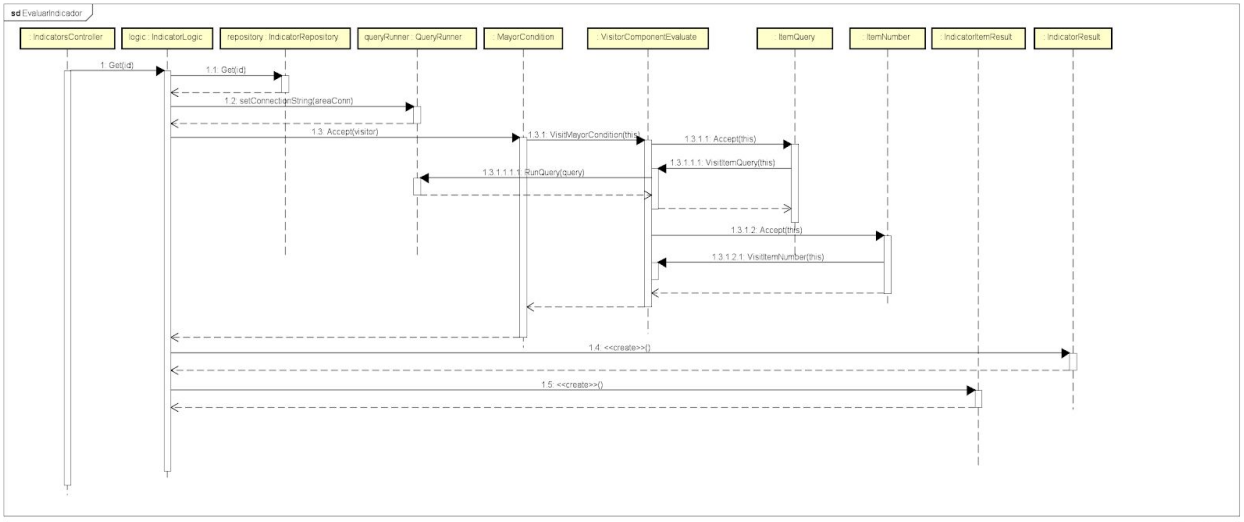
El otro paquete que quedó un gran porcentaje de condicionado a una mejora es el paquete de BusinessLogic ya que existen algunas funcionalidades sin test que las cubran.

Anexo

Diagrama de Secuencia: POST Area



Evaluar un Indicador



Crear un Indicador