**Parte 1**

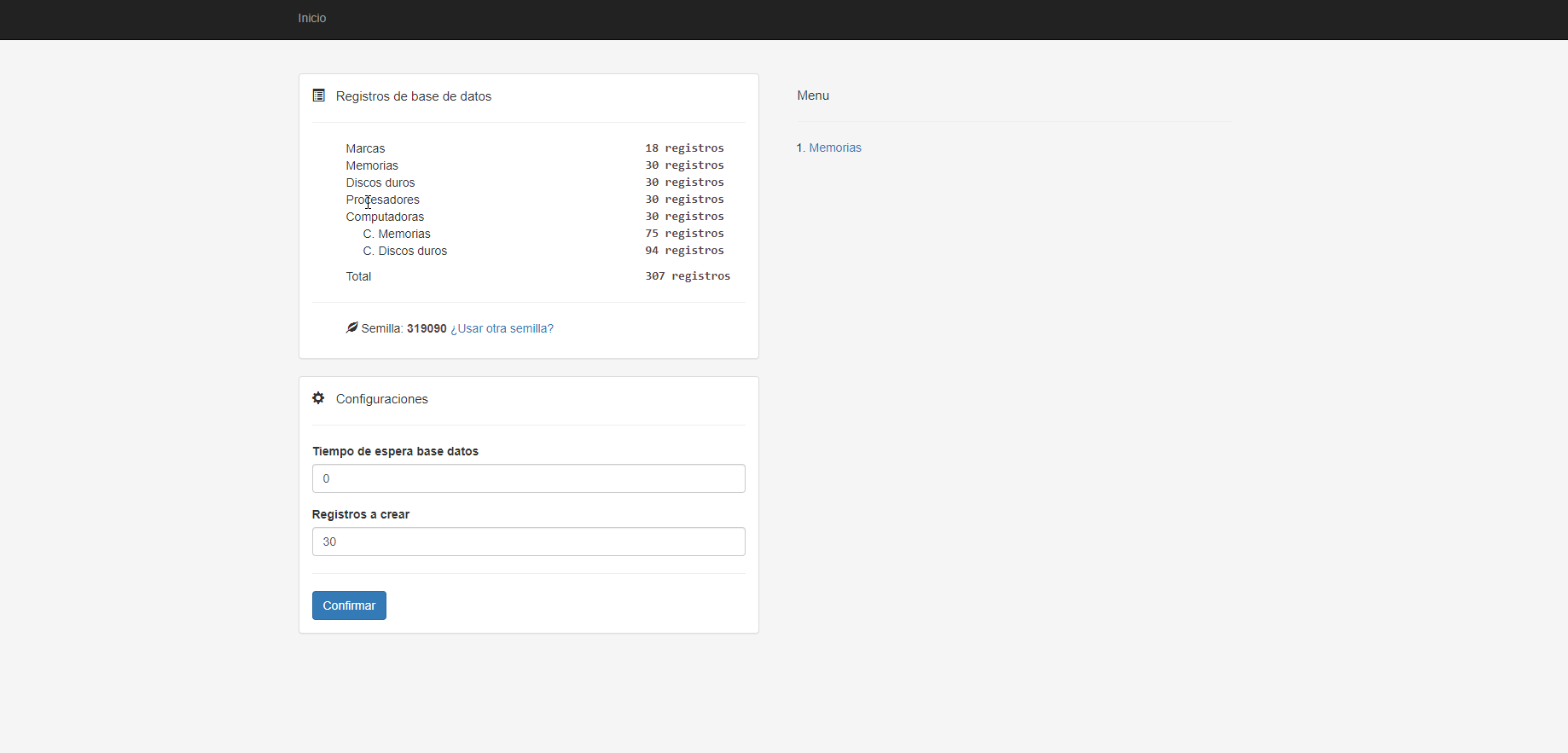
**PARTE**

**DISEÑO UI**

**Resumen.**

El sistema fue realizado con tecnología MVC y WebApi 2, se utilzo también Boostrap junto con JQuery para realizar pequeñas funciones de UX.

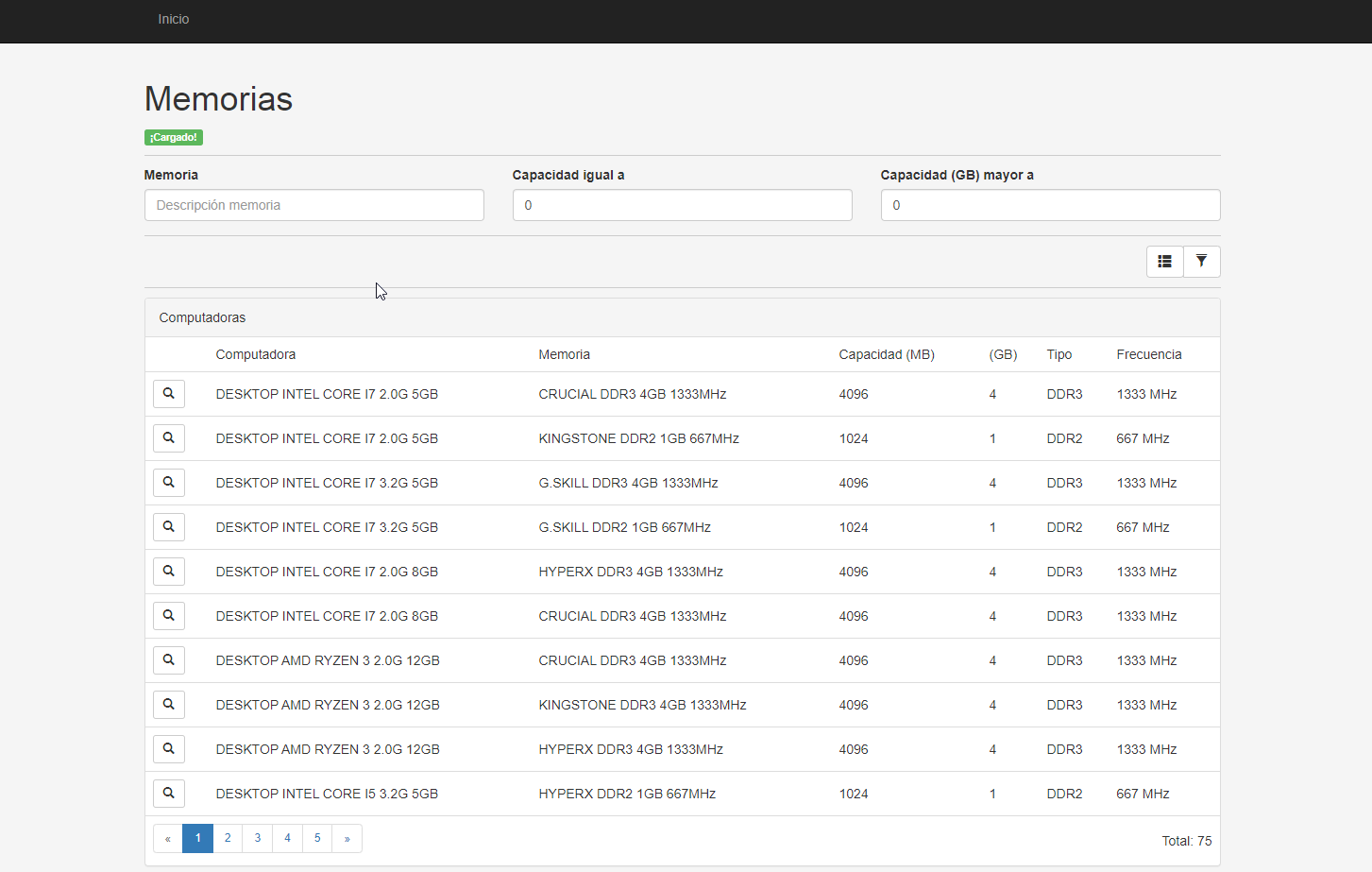
Pantalla de inicio.



En esta pantalla se puede mostrar la cantidad de registros creados por modelo, la respuesta del servidor por cada petición (como obtener la grilla, visualizar, etc.), también se disponibiliza la posibilidad de generar n registros por semilla (o base de datos.).

En la opción de Menu, se puede ir a las distintas páginas, en este momento la aplicación solo posee la acción de ver las memorias por computadoras.

**Menú – Memorias.**



En esta pantalla se mostrara una grilla con las computadoras y sus memorias, con la frecuencia, cantidad (siempre expresado en Megabytes) y el tipo.

También se agregan tres filtros que permiten filtrar la grilla de la pantalla.

Además se puede modificar el paginado de la grilla y el tamaño de esta.

Para filtrar datos se debe apretar el botón de filtro una vez ingresado alguno de los tres campos superiores.

**Parte 2**

**PARTE TECNICA**

**Proyecto**

El proyecto presentado se encuentra bajo las siguientes dependencias usadas.

1. **.NET Framework Estándar 4.6.1**
2. **Bootstrap 3**
3. **JQuery 3.3.1**
4. **Asp Mvc**
5. **Web Api 2**
6. **Visual Studio 2019**

El proyecto fue realizado bajo el modelo **MVC**, se utiliza la mínima cantidad de framework externos y se aplica lo más predeterminado posible la aplicación.

Se utiliza una arquitectura de 3 capas: View, Models, DAL. Con una capa adicional entre DAL y Vista llamada services, quedando de la siguiente forma.

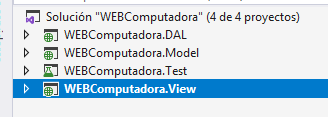
Controller

Services

DAL

Model

Existe un cuarto proyecto de Test Unitarios llamado **WEBComputadora.Test**.

****

**Proyecto capa Vista**.

La aplicación de arranque se encuentra en la capa vista, en esta se utiliza un enfoque MVC, utilizando las siguientes dependencias en la estructura de proyecto.

1. Bootstrap 3
2. JQuery
3. Razor

Para la visualización de páginas se utiliza sintaxis Razor 3, también se utilizan controladores (**controllers**), para el intermediario del UI/Datos. De los controladores existen uno relevante.

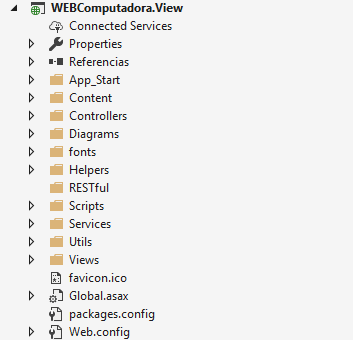
1. **RESTComputadoraMemoriaController**:
   1. Este controlador es el que se encarga de obtener los datos de la base de datos, mediante API Rest, devuelve un JSON e implementa dos métodos, **Get**() y **GetByComputerId**().

Los demás controladores obtienen las vistas modelos, tratan los datos y los arroja al usuario.

Hasta ahora existen solo dos pantallas para mostrar al usuario.

1. **Inicio**: pantalla de inicio donde se puede modificar la cantidad de registros por semilla que genera la base de datos, refrescar semillas y modificar el tiempo de respuesta por consulta o pool de conexión DB.
2. **Memorias**: pantalla con una grilla y tres filtros que permiten trabajar las memorias por computadora.

Estructura de proyecto.

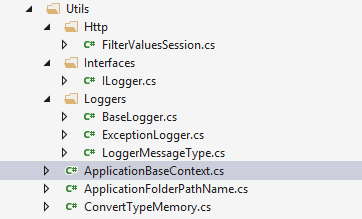


Las carpetas relevantes de este proyecto son:

1. **Controllers**: controladores
2. **Diagrams**: diagramas de clases de visual studio.
3. **Helpers**: Generadores HTML de visual studio.
4. **Services**: services de controladores (se utiliza una capa adicional de services por controladores)
5. **Utils**: utilidades de clases
6. **Views**: vistas

Se utiliza una capa adicional en el proyecto MVC, llamada services, se evita utilizar Unity o cualquier framework de Inyección de Dependencia, debido que se entiende que solo se puede utilizar tecnologías .NET.

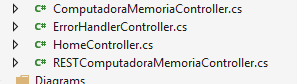
Estructura **Utils**.



La carpeta Utils presenta clases que permiten ayudar el desarrollo en el proyecto vista de la aplicación.

1. FilterValuesSession: clase que permite almacenar los filtros en sesión.
2. Loggers: colección de clases que sirven para generar un archivo logs, en la aplicación.
3. ApplicationBaseContext: clase que permite obtener información como URL, base aplicación o guardar sesiones.
4. ConverTypeMemory: clase que permite pasar de Megabytes a Gigabytes

Estructura **Controllers.**



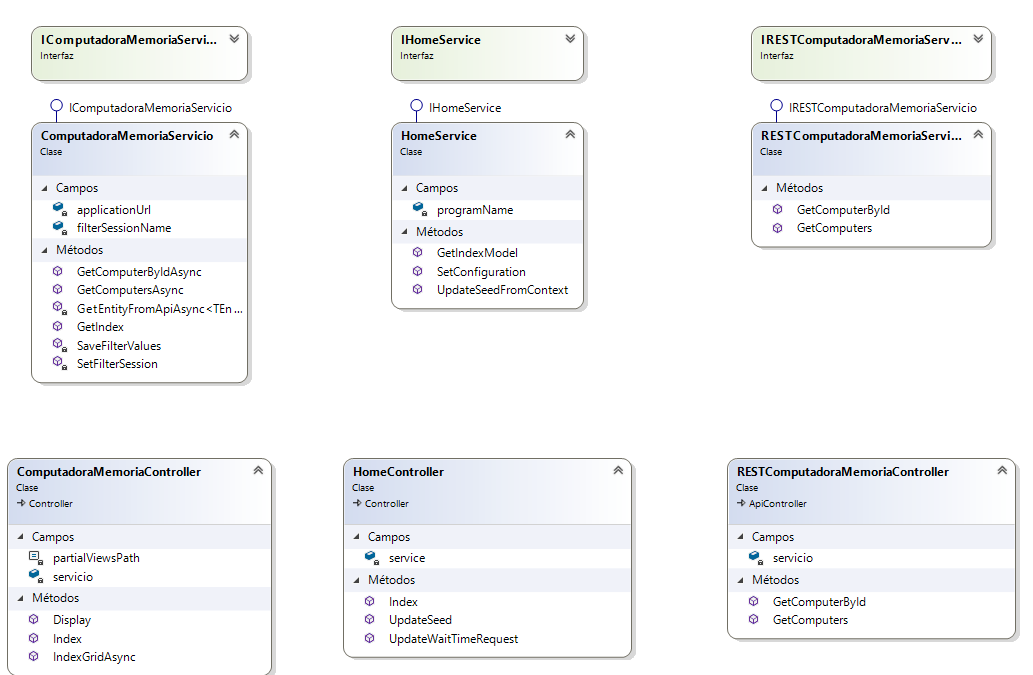
Existen cuatro controladores en la aplicación.

1. **ComputadoraMemoria**: controlador encargado de mostrar los datos de memorias por computadora (grilla y visualización).
2. **ErrorHandler**: controlador usado principalmente para los errores 404.
3. **Home**: controlador de página de inicio.
4. **RESTComputadoraMemoria**: controlador de API REST que permite consultar en la base de datos y devolver en formato Json.

Todos estos controladores implementan su propio **services**.

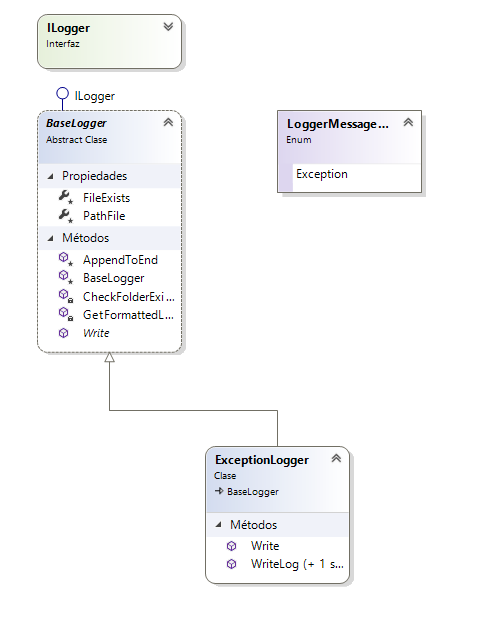
Diagramas relevantes.

Controladores.



Cada controlador implementa su propio services.

Logger:

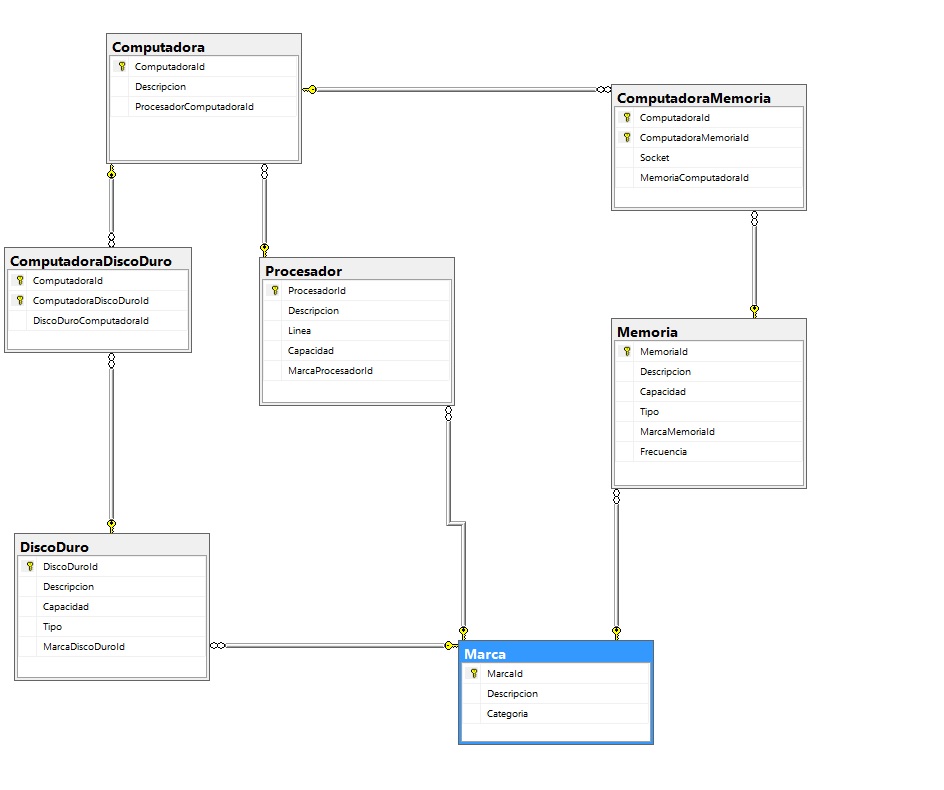


Logger arrojara excepciones o mensajes relevantes que tiene la aplicación en un archivo de texto, hasta ahora solo se implementa para el servicio REST cuando posee un error de URL o por HttpClient (véase **RESTComputadoraMemoriaServices.GetComputers**)

Logger dejara todos los archivos txt en la carpeta “<**SOLUCION>/LOGTEMP/application\_logger\_dd-MM-yyyy.txt**”

**Proyecto capa Modelo.**

*Modelo físico conceptual de referencia al desarrollo de entidades/modelos.*

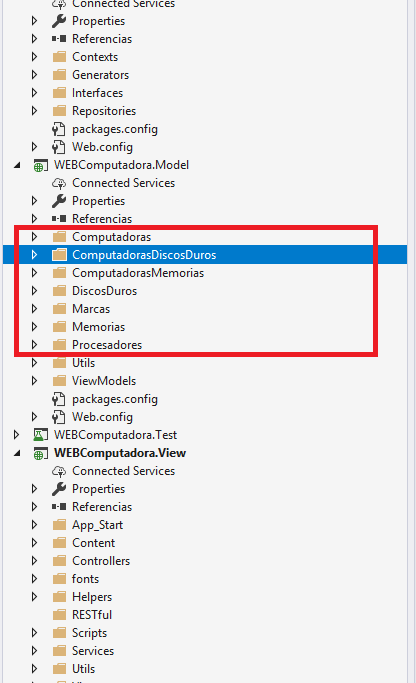


Este diagrama físico sirvió como método conceptual para generar los modelos que posteriormente servirán en el ejercicio del desarrollo de la aplicación MVC.

Se utilizó un enfoque de base de datos para trabajar los datos, se utilizó como referencia si se utilizara el Framework “**EntityFramework**”, donde se utiliza un **Context** y propiedades de modelos.

Es importante mencionar que las cantidades de todos los modelos (memoria, disco duro, etc.) se expresan en megabytes.

Los modelos pueden ser encontrados en el proyecto “**WEBComputadora.Model**”, bajo la siguiente carpeta/estructura.



El recuadro rojo son todas las carpetas que se utilizan como modelo en el ejercicio de desarrollo, dentro de las carpetas se encuentran clases parciales que tienen el siguiente patrón.

**NombreClase.<Tipo>.cs**

Donde **<Tipo>** es la representación parcial de una tipología (relaciones, métodos, etc.).



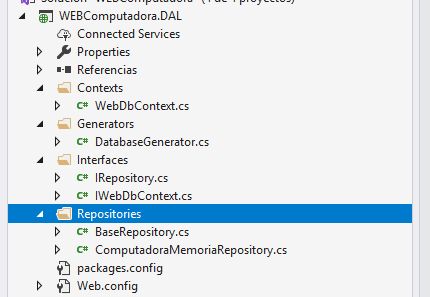
Existen dos carpetas adicionales en este proyecto/capa.

1. **Utils**: Utilidades utilizadas dentro del proyecto modelo usados en **ViewModels**.
   1. Paginado: Clase que permite realizar un páginado completo sin necesidad de implementar lógica dentro de los servicios (services).
   2. Tamaño de páginas (PageSize): clase que sirve para obtener los valores de paginados por defecto (10, 20, 30, 0)
2. **ViewModels**: vistas de modelos usados en controladores.

**Proyecto capa DAL (Data Access Layer)**

En esta capa o proyecto de la aplicación MVC, se puede encontrar el Context y Repositorios.

Se utiliza un enfoque **SQLServer y EntityFramework**, pero NO ocupa estas dependencias, solo se utilizan como si fueran una referencia.



La siguiente estructura representa el proyecto DAL, en resumen, se puede decir:

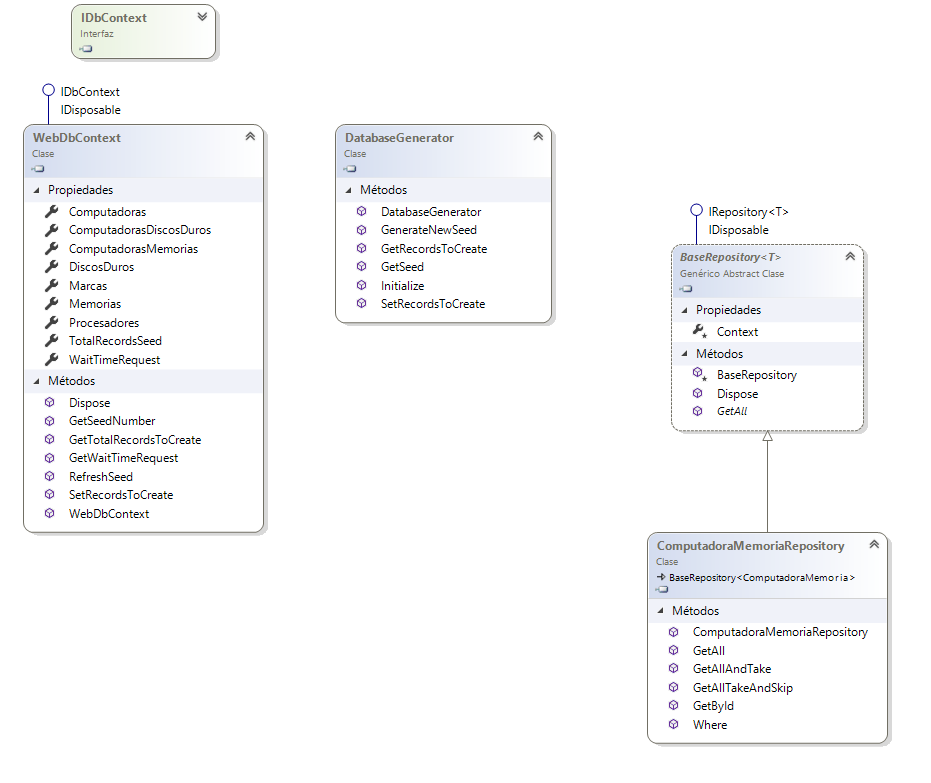
1. **Context**: aquí es donde se encuentra la clase **WebDbContext** que permite obtener los modelos de datos, utilizando el mismo enfoque de **EntityFramework**, implementa IDisposable y tiene persistencia de datos.
2. **DatabaseGenerator**: Esta clase permite generar los datos de forma aleatoria, utiliza Random y descripciones predefinidos para generar el modelo de datos, solo lo realiza una sola vez y puede ser “Refrescado”, cuantas veces el usuario quiera.
3. **Interfaces**: interfaz de proyecto.
4. **Repositories**: repositorios de la aplicación, la clase **BaseRepository** es una clase abstracta y **ComputadoraMemoriaRepository** es una sub-clase (o hereda) de **BaseRepository**.

Forma de generación de datos.

La forma en como genera los datos, es realizada por la clase **DatabaseGenerator**, los métodos, más relevantes son: **Initialize** (permite generar los datos por primera vez y luego traspasarlos a nuevas instancias de IDbContext), **GetSeed** (obtiene número de semilla de los datos), **GenerateNewSeed** (genera de nuevo los datos de forma aleatoria).

Esto puede ser configurado a nivel de aplicación.

Diagramas relevantes.



|  |  |
| --- | --- |
|  | Propiedades   1. **TotalRecordsSeed**: total de datos que crea por cada semilla (**static**). 2. **WaitTimeRequest**: tiempo de espera cuando se consulta los datos (cuando se desea implementar un tiempo de espera mayor por conexión pool a la DB) (**static**).   Métodos   1. **Dispose**: para generar un ambiente “Entityframework” 2. **GetSeedNumber**: obtiene número de semilla de la instancia. 3. **GetTotalRecordsToCreate**: total de datos que crea por semilla desde la instancia. 4. **RefreshSeed**: crea una nueva semilla. 5. **SetRecordsToCreate**: modifica la cantidad de registros que crea la clase DatabaseGenerator cuando se utiliza GenerateNewSeed. |

|  |  |
| --- | --- |
|  | Métodos   1. **GenerateNewSeed**: genera una nueva semilla de datos. 2. **GetRecordsToCreate**: obtiene la cantidad de registros que crea por semilla. 3. **GetSeed**: Obtiene el número de semilla actual. 4. **Initialize**: Genera los datos u obtiene los disponibles por semilla. 5. **SetRecordsToCreate**: Modifica la cantidad de registros que crea por semilla. |