Bases de datos relacionales

Los datos se almacenan de forma persistente (que duren más allá del ciclo de vida de la aplicación o una sesión de usuario) en discos, pero para poder garantizar el acceso y la manipulación de esos datos de forma consistente y segura es que existen las bases de datos. Hay diferentes tipos de bases de datos con diferentes mecanismos de estructuración de los datos así como de las herramientas que brindan. El modelo relacional es sin duda el más usado y el más extendido en el mundo

Origen

En los primeros años de las bases de datos, cada aplicación almacena datos en su propia estructura única. Cuando los desarrolladores querían crear aplicaciones para usar esos datos, tenían que saber mucho sobre la estructura de datos particular para encontrar los datos que necesitaban. Estas estructuras de datos eran ineficientes, difíciles de mantener y difíciles de optimizar para ofrecer un buen rendimiento de la aplicación. El modelo de base de datos relacional se diseñó para resolver el problema de varias estructuras de datos arbitrarias.

El modelo de datos relacionales proporcionó una forma estándar de representar y consultar datos que cualquier aplicación podría utilizar. Desde el principio, los desarrolladores reconocieron que la principal fortaleza del modelo de base de datos relacional estaba en el uso de tablas, que eran una forma intuitiva, eficiente y flexible de almacenar y acceder a información estructurada.

Con el tiempo, cuando los desarrolladores comenzaron a utilizar el lenguaje de consulta estructurado (SQL) para escribir y consultar datos en una base de datos, surgió otra fortaleza del modelo relacional. Durante muchos años, se utilizó ampliamente el SQL como lenguaje para consultas de bases de datos. El SQL, que se basa en el álgebra relacional, proporciona un lenguaje matemático internamente consistente que facilita la mejora del rendimiento de todas las consultas de la base de datos. En comparación, otros enfoques deben definir consultas individuales.

Utilidades y ventajas

El modelo relacional es el mejor para mantener la consistencia de los datos en todas las aplicaciones y copias de la base de datos (denominadas instancias). Por ejemplo, cuando un cliente deposita dinero en un cajero automático y, luego, mira el saldo de la cuenta en un teléfono móvil, el cliente espera ver que ese depósito se refleja inmediatamente en un saldo de cuenta actualizado. Las bases de datos relacionales se destacan en este tipo de consistencia de datos, lo que **garantiza que múltiples instancias de una base de datos tengan los mismos datos todo el tiempo**.

Es difícil para otros tipos de bases de datos mantener este nivel de coherencia oportuna con grandes cantidades de datos. Algunas bases de datos recientes, como NoSQL, sólo pueden

proporcionar “coherencia eventual”. Según este principio, cuando se escala la base de datos o cuando varios usuarios acceden a los mismos datos al mismo tiempo, los datos necesitan algo de tiempo para “ponerse al día”. La coherencia eventual es aceptable para algunos usos, como mantener listados en un catálogo de productos, pero para operaciones comerciales críticas, como transacciones de carrito de compras, la base de datos relacional sigue siendo lo ideal.

Estructura

En una base de datos relacional los datos se almacenan en tablas. Las tablas tienen columnas en cada una de ellas se especifica el nombre del atributo a almacenar(Ej: para la tabla Animales una columna puede tener de nombre de atributo “dueño”) y el tipo de dato (Ej: varchar, que es como string en c#)

Uno de los aspectos más importantes de las bases de datos es la posibilidad de identificar unívocamente a cada registro de las tablas. Es decir cada registro es identificable por un dato o conjunto de datos.(Ej: para la tabla Personas el dato que lo difiere no podría ser nombre porque hay personas que se llaman igual, pero puede ser el DNI). Esto en las bases de datos relacionales se logran a través de **las claves.**

Cada tabla debe tener una **clave primaria**, es decir un dato o conjunto de datos que permiten identificar cada registro de la misma. No puede haber dos registros con una clave idéntica. Las claves primarias pueden ser simples, compuestas por un único atributo, o compuestas, es decir compuestas por más de un atributo. Ejemplo: para la tabla personas su clave deberá ser compuesta por dos atributos, “tipo” y “nro de documento”

Además de las claves primarias existen otras claves que se llaman **claves foráneas,** estas permiten relacionar un registro de una tabla con otro registro de otra tabla. Esto es muy útil y muy usado. Por ejemplo, la tabla “CuentasBancarias” probablemente tenga los atributos “TipoDocumento” y “NroDocumento” que permiten relacionar una cuenta bancaria con una persona de la tabla Personas. Las claves foráneas como acabamos de ver pueden ser simples o compuestas

Object Relational Mapper - ORM

¿Qué es?

Un ORM es un modelo de programación para mapear estructuras de una base de datos relacional, como por ejemplo SQL Server, MySQL u Oracle, entre otras. Su objetivo es simplificar y acelerar el desarrollo de las aplicaciones.

Las estructuras de la Base de datos relacional se vinculan con las entidades lógicas o con la BD virtual que define el ORM, para que las distintas acciones de CRUD (crear, leer, actualizar o borrar) se puedan realizar de manera indirecta a través del ORM.

No solo permiten mapear, sino también liberarnos de picar código SQL que habitualmente hace falta para las queries o consultas y gestionar la persistencia de datos Por lo que, los objetos se pueden manipular mediante lenguajes según el tipo de ORM utilizado. Un ejemplo es LINQ sobre Entity Framework de Microsoft.

Asimismo, los ORMs más completos ofrecen distintos servicios y sin escribir código de SQL. Es una ventaja, dado que permite atacar las entidades de la Base de datos virtual sin generar el código. Por lo que, se acelera el desarrollo de las aplicaciones.

¿Para qué sirve un mapeador de objetos relacionales?

Su principal funcionalidad pasa porque el proceso de programación de la BD sea más rápido, dado que se consiguen reducir los códigos y que el mapeo sea más automático. De tal forma que el programador o desarrollador no tenga que adaptar los códigos a las tablas en base a las necesidades específicas de cada aplicación.

Es muy complicado convertir la información que recibes de la base de datos (que suele ser en tablas) a los objetos del programa en sí y viceversa. Con la utilización de un ORM, conseguirás que este proceso de mapeo sea automático e independiente de la BD, pudiendo realizar cambios, añadir más campos, etc sin tener que tocar todo el código.

También presenta importantes ventajas en cuanto a seguridad, debido a la capa de acceso que actúa como barrera frente a los ataques. A su vez la forma con la que los ORMs realizan sus consultas a las bases de datos suelen ser muy óptimas y seguras ya que están hechas para que se hagan de la mejor manera. Por lo que es otra importante característica a tener en cuenta.

Es más fácil de mantener, aunque para ello se recomienda tener conocimientos avanzados sobre su funcionamiento. Es una tarea que ante la duda se puede dejar en manos de un tercero para que sea el programador profesional el que se encargue de todo el back-end de tu página o aplicación.

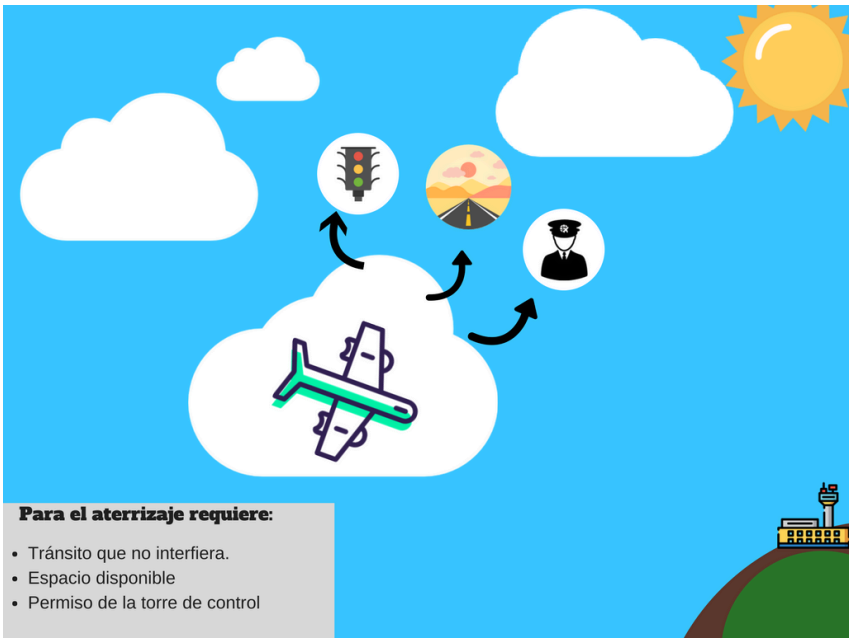
En definitiva, funciona como una solución intermedia que facilita la tarea al programador y permite que el acceso a los datos sea automático. Por lo que es una herramienta interesante para quienes manejan diferentes bases de datos de empresas, organizaciones, entidades, etc

Cada lenguaje tiene uno o varios ORMs, en el caso de todo el ambiente de .NET el ORM por excelencia es Entity Framework en sus diferentes versiones, sin embargo existen otras opciones como NHibernate

Consumo de servicios e Inyección de dependencia

Imagínese que usted como piloto de avión comercial, cada vez que tenga pilotear tuviera que llenar el tanque de combustible, preparar la pista, verificar que todo esta listo para el despegue, sabemos que no es humanamente posible, por eso existen distintas personas

trabajan en conjunto para que sea posible el despegue. Entonces si adaptamos esta situación a nuestro idioma. Tendríamos que:



Cuando un piloto va a aterrizar en una pista, necesita tener el espacio para poder descender y que el tránsito actual del aeropuerto no interfiera con su aterrizaje. Pero el no va a comunicarse directamente con el controlador de Tierra o el controlador de Ruta o Área. En lugar de ello, se comunicara con la torre de control quien se encarga de gestionar todo el proceso y darle toda la información que necesita para el aterrizaje.

El aeropuerto hace la función de Contenedor de dependencia, pues se encarga de gestionar todo el proceso para que los vuelos puedan tener lugar. Los controladores aéreos, hacen la función de Framework, pues se encargan de inyectar las dependencias a los módulos dependientes y gestionar los recursos para que los módulos puedan funcionar correctamente, las interfaces son los lineamientos que se necesitan para que el avión pueda aterrizar. Todos estos elementos en conjunto conforman la inyección de dependencia y la inversión de control.

Ventajas de Utilizar Inyección de Dependencias

Flexible

- No hay necesidad de tener un código de búsqueda en la lógica de negocio.
- Elimina el acoplamiento entre módulos

Testable

- No se necesita un espacio específico de testeo

- Testeo automático como parte de las construcciones

Mantenible

- Permite la reutilización en diferentes entornos de aplicaciones modificando los archivos de configuración en lugar del código.
- Promueve un enfoque coherente en todos aplicaciones y equipos

Ejemplo de código

en este caso la clase a inyectar es una clase de tipo repository

CLASE EN LA QUE SE INYECTA

```
public class LogicaNegocio
{
    AccesDataCliente _dataAccess;

    public LogicaNegocio(AccesDataCliente custDataAccess)
    {
        _dataAcces = custDataAccess;
    }

   public string ProcesoDataCliente(int id)
    {
        return _dataAcces.GetDataCliente(id);
    }
}
```

```
public interface IAccesoDataCliente
{
    string GetDataCliente(int id);
}
```

CLASE A INYECTAR

```
public class AccesDataCliente: IAccesoDataCliente
{
   public AccesDataCliente()
    {
    }

    public string GetDataCliente(int id)
    {
        //obtener data del cliente desde una BD
        return "Dummy Customer Data";
    }
```

```
}
```

En el ejemplo anterior, LogicaNegocio incluye constructor con un parámetro de tipo IAccesoDataCliente. Ahora, la clase de llamada debe inyectar un objeto de IAccesoDataCliente.

Continuemos con el ejemplo

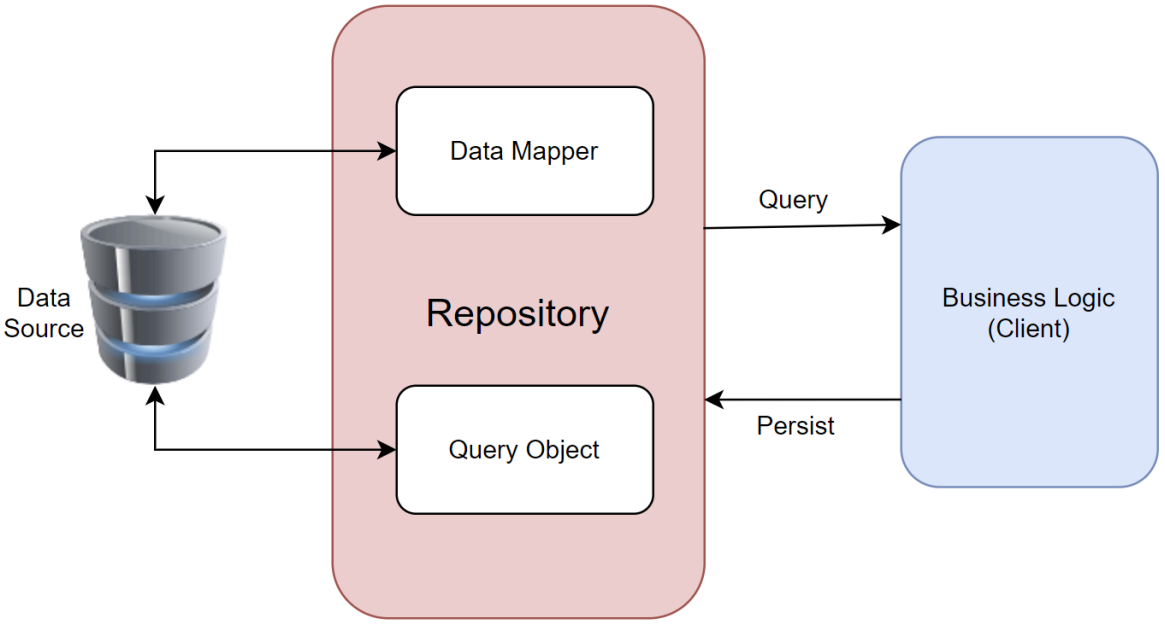
```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

   public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic(new
CustomerDataAccess());
    }

   public string GetCustomerName(int id) {
       return _customerBL.GetCustomerName(id);
    }
}
```

Existen múltiples herramientas que pueden ser inyectadas como servicios en algunas clases para poder usarlos sin necesidad de instanciar un objeto de cada servicio en cada consulta. Algunos de estos pueden ser un servicio de logs o un servicio de mailing o bien un servicio de ORM, como Entity Framework

Patrón repository

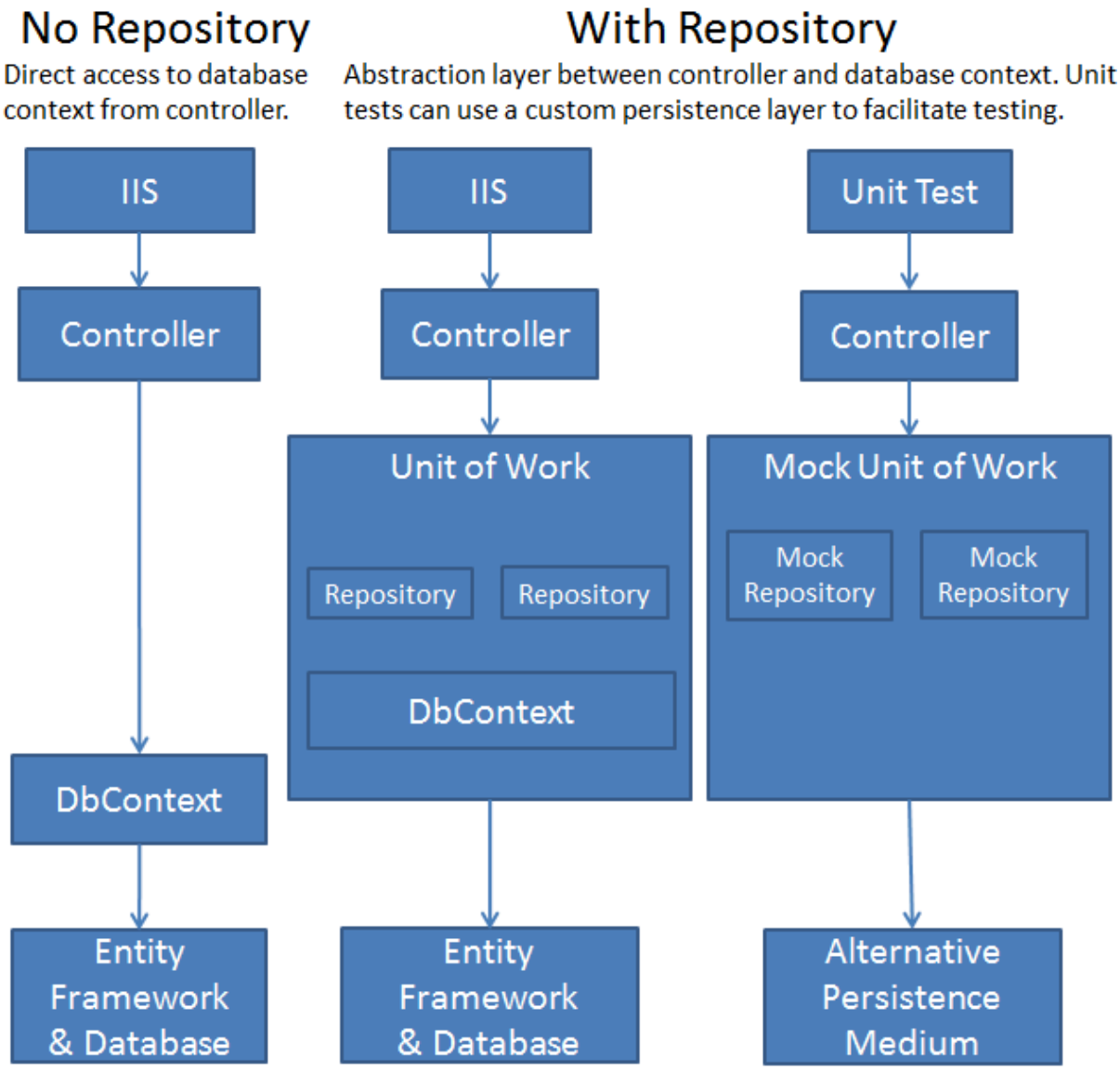


Está diseñado para crear una capa de abstracción entre la capa de acceso a datos y la capa de lógica de negocios de una aplicación. Implementar estos patrones puede ayudar a aislar la aplicación de cambios en el almacén de datos y puede facilitar la realización de pruebas unitarias automatizadas o el desarrollo controlado por pruebas (TDD).

Este patrón permite crear una clase (o varias) en donde estén definidos todos los métodos que interactúen con la base de datos, esta clase consume el contexto de la base de datos.

Otro de los aspectos fundamentales es que esta clase/s que se encarga de los métodos de datos implementando una interfaz en donde se definen la firma de los métodos(tipo nombre y parámetros del método) que componen a la clase que hace de repository. Si tengo dividido el repository en varias clases entonces debo definir una interfaz por clase y hacer que dicha clase implemente la interfaz.

La interfaz tiene la utilidad de que si en un futuro quiero implementar otro repository ya sea que porque tengo otro contexto debido a que algunos datos se manejan en otra base datos para inteligencia de negocio o por motivos de disponibilidad, entonces solo hace falta implementar la misma interfaz en la nueva clase repository análoga y de esta manera nos aseguramos que la clase contenga la totalidad de los métodos definidos de la capa de datos y definidos de la misma manera así nos evitamos inconvenientes en las capas lógicas que utilizan estos métodos



Documentación de EF para crear un modelo

EF Core usa un modelo de metadatos para describir como se mapean las entidades con la correspondiente base de datos.
El modelo se puede construir usando un conjunto de convenciones establecidas en el framework, usando atributos de mapeo (data annotations) y/o ejecutando métodos del objeto ModelBuilder al sobreescribir OnModelCreating del objeto context. (Fluent API).
Fluent API tiene prioridad sobre las data annotations y las data annotations tienen prioridad sobre las convenciones.

Creating and Configuring a Model