The Repository Pattern

Context

In many applications, the business logic accesses data from data stores such as databases, SharePoint lists, or Web services. Directly accessing the data can result in the following:

- Duplicated code
- A higher potential for programming errors
- Weak typing of the business data
- Difficulty in centralizing data-related policies such as caching
- An inability to easily test the business logic in isolation from external dependencies

Objectives

Use the Repository pattern to achieve one or more of the following objectives:

- You want to maximize the amount of code that can be tested with automation and to isolate the data layer to support unit testing.
- You access the data source from many locations and want to apply centrally managed, consistent access rules and logic.
- You want to implement and centralize a caching strategy for the data source.
- You want to improve the code's maintainability and readability by separating business logic from data or service access logic.
- You want to use business entities that are strongly typed so that you can identify problems at compile time instead of at run time.
- You want to associate a behavior with the related data. For example, you want to calculate fields or enforce complex relationships or business rules between the data elements within an entity.
- You want to apply a domain model to simplify complex business logic.

Solution

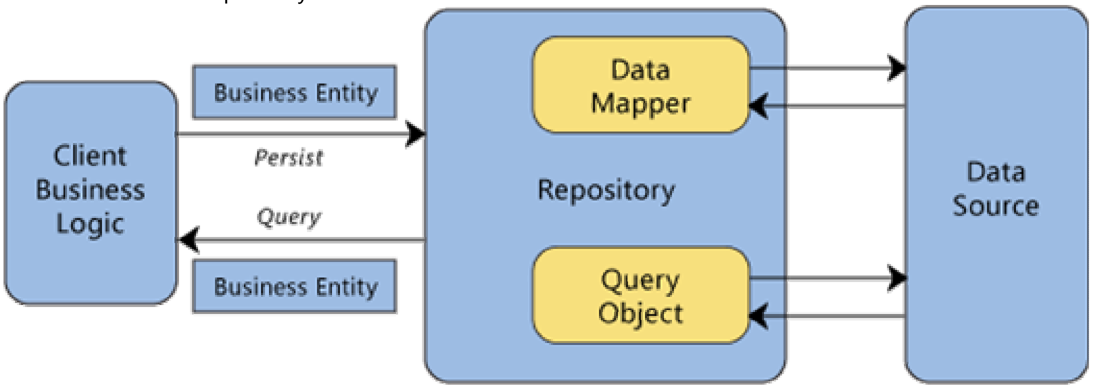
Use a repository to separate the logic that retrieves the data and maps it to the entity model from the business logic that acts on the model. The business logic should be agnostic to the type of data that comprises the data source layer. For example, the data source layer can be a database, a SharePoint list, or a Web service.

The repository mediates between the data source layer and the business layers of the application. It queries the data source for the data, maps the data from the data source to a business entity, and persists changes in the business entity to the data source. A repository separates the business logic from the interactions with the underlying data source or Web service. The separation between the data and business tiers has three benefits:

- It centralizes the data logic or Web service access logic.
- It provides a substitution point for the unit tests.
- It provides a flexible architecture that can be adapted as the overall design of the application evolves.

There are two ways that the repository can query business entities. It can submit a query object to the client's business logic or it can use methods that specify the business criteria. In the latter case, the repository forms the query on the client's behalf. The repository returns a matching set of entities that satisfy the query. The following diagram shows the interactions of the repository with the client and the data source.

Interactions of the repository



The client submits new or changed entities to the repository for persistence. In more complex situations, the client business logic can use the Unit of Work pattern. This pattern demonstrates how to encapsulate several related operations that should be consistent with each other or that have related dependencies. The encapsulated items are sent to the repository for update or delete actions. This guidance does not include an example of the Unit of Work pattern. For more information, see Unit of Work on Martin Fowler's Web site.

Repositories are bridges between data and operations that are in different domains. A common case is mapping from a domain where data is weakly typed, such as a database or SharePoint list, into a domain where objects are strongly typed, such as a domain entity model. One example is a database that uses **IDbCommand** objects to execute queries and returns **IDataReader** objects. Another example is SharePoint, which uses **SPQuery** objects to return **SPListItem** collections. A repository issues the appropriate queries to the data source, and then it maps the result sets to the externally exposed business entities. Repositories often use the Data Mapper pattern to translate between representations. Repositories remove dependencies that the calling clients have on specific technologies. For example, if a client calls a catalog repository to retrieve some product data, it only needs to use the catalog repository interface. For example, the client does not need to know if the product information is retrieved with SQL queries to a database or Collaborative Application Markup Language (CAML) queries to a SharePoint list. Isolating these types of dependences provides flexibility to evolve implementations.

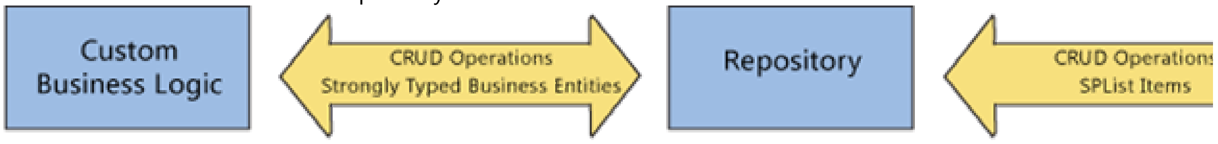
Implementation Details

This section discusses the implementation strategies for SharePoint list repositories and Web service repositories.

SharePoint List Repositories

The following diagram illustrates the interactions of a SharePoint list repository with SharePoint lists and the business logic.

Interactions of a SharePoint list repository



Using the Repository pattern in a SharePoint application addresses several concerns.

- SharePoint applications often store business information in SharePoint lists. To retrieve data from SharePoint lists requires careful use of the SharePoint API, knowledge of the GUIDs that are related to the lists and their fields, and a working knowledge of CAML. Repositories centralize this logic.
- The amount of code that is required to query or update a SharePoint list item is enough to warrant its encapsulation into helper methods. When Web Forms, event receivers, and workflow business logic all require access to the same lists, the code that accesses the SharePoint lists can be duplicated throughout the application. This can make the application prone to bugs and difficult to maintain. Repositories eliminate this duplication.
- Without a repository, the application is difficult to unit test because the business logic has direct dependencies on the SharePoint lists. Repositories centralize the access logic and provide a substitution point for the unit tests.

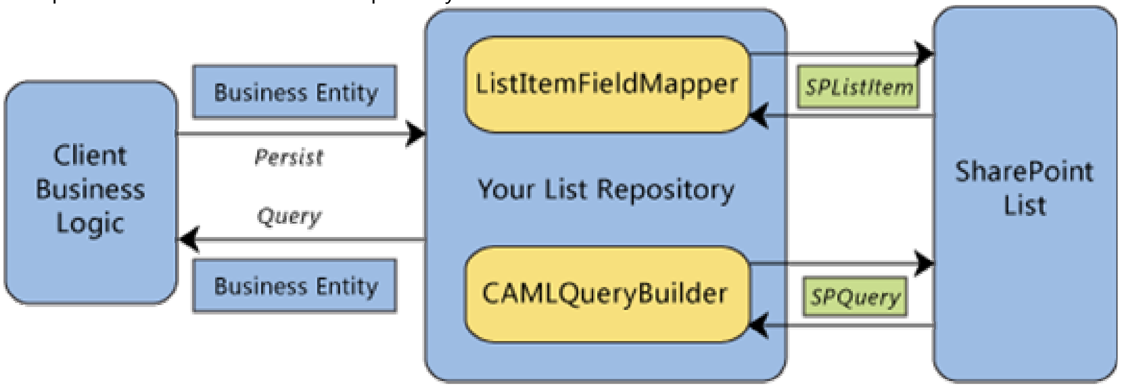
Externally, the repository exposes strongly-typed business entities. Internally, it works with SharePoint-specific objects, such as the **SPQuery** and the **SPListItem** objects. The SharePoint Guidance Library, which is a part of this guidance, provides classes for mapping and querying that make it easier to build repositories for SharePoint lists. The **ListItemFieldMapper** class converts strongly-typed business entities to and from **SPListItem** objects based on a set of mapping definitions. The **CAMLQueryBuilder** class builds **SPQuery** objects based on common query operations. The **SPQuery** object is used to query a SharePoint list.

The following sections show how the repository pattern is implemented in the SharePoint Guidance Library. For more information, see List-Based Repositories.

SharePoint Guidance Library Helper Classes

The following diagram shows the major components of a SharePoint list repository.

Components of a SharePoint list repository



The list repository contains a query object and a data mapper object that are specific to SharePoint. These are the **ListItemFieldMapper** and **CAMLQueryBuilder** classes. The data mapper translates between an **SPListItem** and the business entity that is defined by the application. The query object internally constructs an **SPQuery** object and uses CAML to query the list.

Note:

When you design a SharePoint list repository, keep in mind that a list can contain fields from multiple content types. The logic that is implemented in the **ListItemFieldMapper** and **CAMLQueryBuilder** objects does not prevent fields from multiple content types from being retrieved. In some cases, if the content types have the same parent content type, you can use a single repository to project a common view across these content types.
However, it is generally inadvisable to create repositories that deal with dissimilar content types and return different business entities from the same repository. In this situation, create a repository for each content type because the content types logically represent different entities.

Implementation Variations

When you create a SharePoint list repository, you should consider how the repository locates the list that it is going to access. A list typically resides in a site, and it can be accessed either through its Uniform Resource Identifier (URI) or its GUID. The repository needs one of these, but passing this information to a repository can be challenging if you use the repository in conjunction with a service location. For more information, see The Service Locator Pattern.

There are three ways in which a repository can access a list:

- **A list can be centrally located**. In this case, a repository is associated with a list that is at a fixed location. All sites retrieve the data from this central location. The **PartnerPromotionsRepository** class in the Partner Portal application is an example of a repository that uses such as list.
- **A list can be accessed relative to the current site context**. In this case, a repository is associated with a list whose location is relative to the current site. The **IncidentManagementRepository** class in the Partner Portal application is an example of such a repository.
- **A list can be accessed according to a context that is supplied by a consumer**. In this case, only the consumer of the repository knows which list the repository should access. There is no example of this in the Partner Portal application. However, you can extend the Training Management application to support this use case. You can implement a repository that accesses the list of training courses for your department on your local installation and also accesses related training courses that are located on another departmental site. In this case, the consumer instructs the repository to target a particular list.

The following sections describe more details about how to associate repositories with lists.

Lists That Are Centrally Accessed from a Fixed Location

In this case, a list is at a fixed location, and all sites access it from this central point. Its location cannot be determined based on the current context. Although it is possible to hard code the location of the list, this is not recommended, because the topology of the site can change. It is often better to make the location of the list a configuration setting. In that case, defining the list's location is an administrative task. The location is established when the site topology is set up.

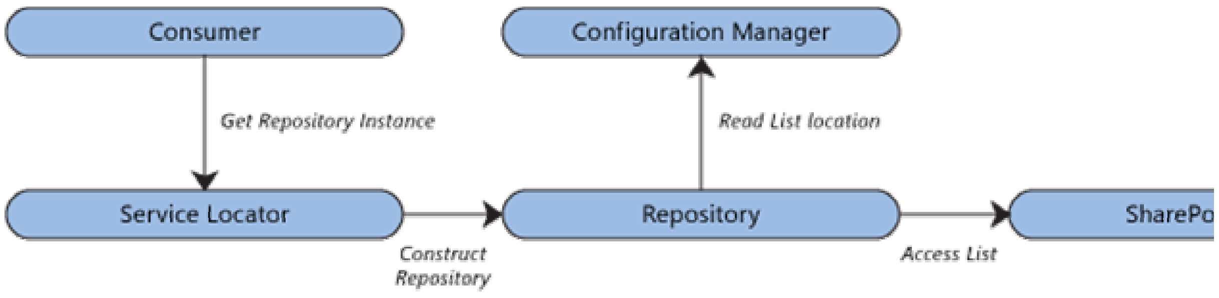
For example, the Partner Portal application centrally manages the published promotions for all partners by locating them on one site collection. Partners see their particular promotions on their collaboration home pages. Each partner collaboration site is hosted in its own site collection. This establishes security boundaries and isolates the data intended for one partner from the data intended for another partner. Because the relationship between the list and consumers of the list is based on the operational topology, the list location is defined with configuration data.

The following are characteristics of a list with a fixed location:

- There is typically only one instance of that list within the Web application scope.
- The location of the list is determined when the site topology is designed or when the site is installed.
- The list location should be retrieved from configuration data that is shared by all consumers. This data is typically at the Web application level or Web farm level.

The following diagram shows the flow of information among components that access a list at a central location.

Associating a repository with a list at a central location



The service locator constructs a repository object, which then reads the configuration information. The repository accesses the list based on this data. Because the repository relies on the configuration data, it can be constructed independently of the application context. It does not need any additional information from the list consumers.

This approach is susceptible to run-time exceptions because it depends on configuration data, which can be erroneous, lost, or corrupted. Make sure that you provide adequate diagnostics that inform IT administrators of any configuration errors. Problems that are caused by configuration errors are difficult to resolve without adequate logging information.

Lists with a Location That Is Fixed, Relative to the Current Context

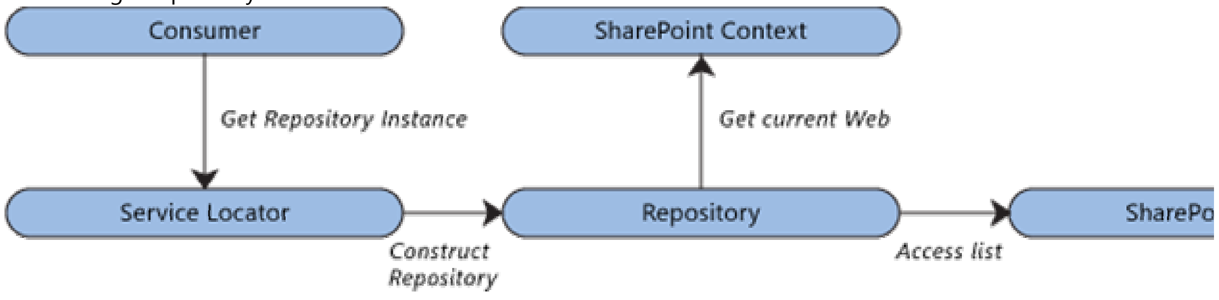
In this case, the repository is associated with a list whose location is fixed, relative to the current context. For example, in the Training Management application, registration and course lists are located at the same relative location within a site. However, there can be several Training Management sites within a SharePoint farm. In this situation, the list repository is loaded from the current context.

The following are characteristics of a list with a location that is relative to the context:

- The list has a fixed location that is relative to a site (an **SPWeb** object).
- The location is independent of the site topology.
- The list location is based on the current SharePoint context.

The following diagram shows the components and flow of information that access a list with a location that is relative to the context.

Associating a repository with a list whose location is relative to the context



SharePoint often has a number of instances of the same Web application. In this situation, the repository gets the current site from the SharePoint context (the **SPContext.Current.Web** object) and loads the list information from this context. Because this relationship is fixed relative to the current site, the repository needs no additional information. The repository instance can be directly constructed by the service locator.

Lists Whose Context Is Supplied by a Consumer

In this case, the repository is associated with a particular content type and can access any list that has **SPListItem**s of this content type. The consumer must provide context to the repository. For example, the consumer might provide the **SPWeb** object that holds the list or the GUID of the list. Although it is generally a good practice to keep technology-specific dependencies (such as a reliance on SQL Server) out of the repository interface, providing context when the repository is constructed is an accepted, widely used practice.

This scenario occurs with sites that have a dynamic topology, or where relationships are established by a user who supplies configuration information. If you add or remove sites at run time that contain lists that the repository accesses, you often have to provide the context. An example is if you use the Finance training site to view courses but you also want to see the courses on the Human Resources training site.

To provide this capability, you can build a general purpose Web Part to view the courses from other departmental training sites. You can add this Web Part to the Finance training site and configure it to view the related Human Resources department courses. The repository that the Web Part on the Finance site uses receives the location of the Human Resources course list as context information when the Web Part constructs it.

One challenge with this type of repository is using it in combination with the SharePoint Guidance Library service locator. The **ActivatingServiceLocator** class can only use parameterless constructors for the repositories. It is not possible to pass the contextual information (in this case, the location of the list) into the repository through the constructor. One way to solve this is to pass the location of the list with each method call, but this inserts a dependency on the list's URL into the interface definition. The Training Management application uses this approach.

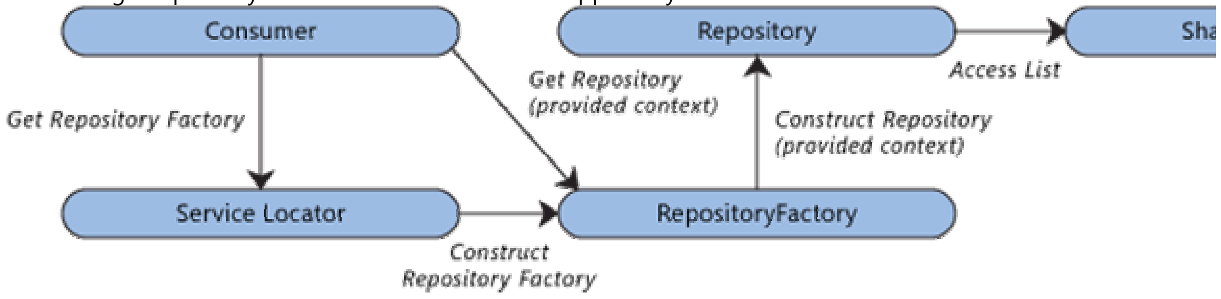
A better, but more complicated way to pass the location of the list to the repository is to use a factory. The factory includes a method that creates the repository. The consumer passes the location of the list to the method. The consumer then uses the **ActivatingServiceLocator** to access the factory and uses it to create the repository. With this approach, the consumer provides the location of the list to the factory, which in turn creates the repository. The factory passes the context through the repository constructor. This technique is known as constructor injection.

The following are characteristics of a list whose context is supplied by a consumer:

- The consumer can determine which list the repository should access.
- The location of the list is often determined at run time.
- This list location is derived from the current business context.

The following diagram shows the components and flow of information that are involved in accessing a list whose context is supplied by the consumer.

Associating a repository with a list whose context is supplied by the consumer


Consumer
Repository
Sha
Access List
Get Repository (provided context)
Get Repository Factory
Construct Repository (provided context)
Service Locator
RepositoryFactory
Construct Repository Factory

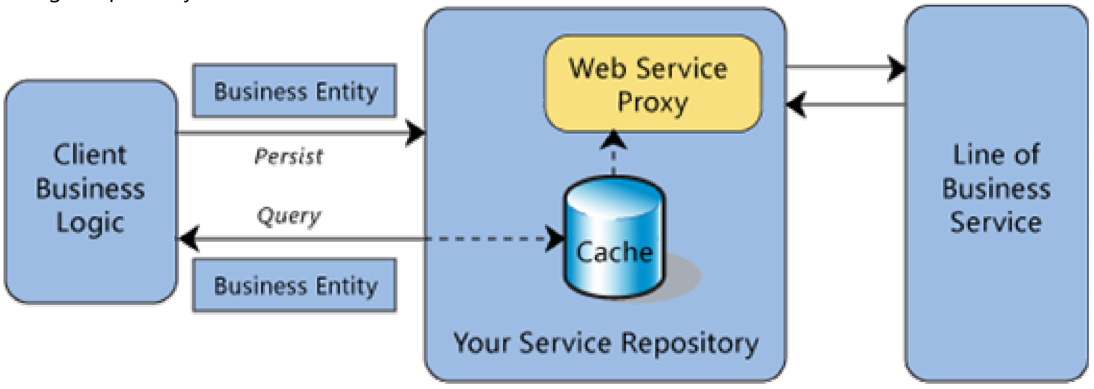
The consumer constructs the context for the repository. The consumer retrieves an instance of a repository factory from the service locator. The consumer then uses the repository factory to construct the repository. The consumer provides the context for the list. The repository uses this information to locate the list. Because the repository is decoupled from both the configuration data and the context, it is suitable for many scenarios. However, because the consumer provides the context, it increases the coupling between the consuming code and the repository.

Web Service Repositories

A common backing store for data is a business service that is exposed by a line-of-business (LOB) application. Generally, these business services are at a higher level of abstraction than the standard Create/Read/Update/Delete (CRUD) semantics of a database or SharePoint list. However, from the perspective of the client, they often are equivalent to a data source. Like with SharePoint lists, accessing Web services can be complex and prone to error. A repository centralizes the access logic for a service and provides a substitution point for unit tests. Note that services are often expensive to invoke and benefit from caching strategies that are implemented within the repository.

The following diagram shows a service back-end repository that uses caching.

Using a repository with a Web service



In this case, the query logic in the repository first checks to see whether the queried items are in the cache. If they are not, the repository accesses the Web service to retrieve the information. Although it is possible to access services directly, it is also possible to access them through the SharePoint Business Data Catalog (BDC). The BDC can aggregate several data sources, including Web services, and expose them through a uniform, generic interface. The BDC allows you to use standard Web Parts to display and modify data. For more information, see Consuming Web Services with the Business Data Catalog (BDC).

You may need more complex security options than the BDC supports. In this situation, you can use the Windows Communication Foundation (WCF). This requires that your own code and configuration data manage the service information and security context. For more information, see Integrating Line-of-Business Systems.

Repository Examples

For an example of the list repository pattern, see Development How-to Topics. Also, the Partner Portal application includes the following list repositories that can be used as starting points:

- The Partner Promotion Repository is in the PartnerPromotionRepository.cs file of the PartnerPortal\Contoso.PartnerPortal.Promotions directory. There is also a mock implementation for unit testing in the PartnerPromotionsPresenterFixture.cs file of the PartnerPortal\Contoso.PartnerPortal.Promotions.Tests directory.
- The Business Event Type Configuration Repository is in the BusinessEventTypeConfigurationRepository.cs file of the Microsoft.Practices.SPG2 \Microsoft.Practices.SPG.SubSiteCreation\BusinessEventTypeConfiguration directory. There is also a mock implementation for unit testing in the ResolveSiteTemplateFixture.cs file of the Microsoft.Practices.SPG2 \Microsoft.Practices.SPG.SubSiteCreation.Tests directory.
- The Subsite Creation Requests Repository is in the SubSiteCreationRequestsRepository.cs file of the directory Microsoft.Practices.SPG2\Microsoft.Practices.SPG.SubSiteCreation\SubSiteCreationRequests.

For an example of the data repository pattern using Web services, see the following areas of the reference implementation:

- The Incident Management Repository is in the IncidentManagementRepository.cs file of the directory PartnerPortal\Contoso.LOB.Services.Client\Repositories.
- The Pricing Repository is in the PricingRepository.cs file of the directory PartnerPortal\Contoso.LOB.Services.Client\Repositories.
- The Cached BDC Product Catalog Repository is in the CachedBdcProductCatalogRepository.cs file of the directory PartnerPortal\Contoso.LOB.Services.Client\Repositories. There is also a mock implementation for unit testing in the ProductDetailsPresenterFixture.cs file of the directory PartnerPortal\Contoso.PartnerPortal.ProductCatalog.Tests.

The Partner Portal application also contains two other repositories:

- The Full Text Search IncidentTask Repository uses SharePoint Search as its data source. This repository is found in the FullTextSearchIncidentTaskRepository.cs file of the directory PartnerPortal\Contoso.PartnerPortal.Collaboration.Incident\Repositories.
- The Partner Site Directory uses the site directory list to provide the Partner site collection URL and the user profile to provide the PartnerID. The repository is implemented in the PartnerSiteDirectory.cs file of the directory PartnerPortal\Contoso.PartnerPortal.PartnerDirectory.

For more information about the Repository pattern, Unit of Work pattern, and Data Mapper pattern, see Repository on Martin Fowler's Web site.

Considerations

The Repository pattern increases the level of abstraction in your code. This may make the code more difficult to understand for developers who are unfamiliar with the pattern. Although implementing the pattern reduces the amount of redundant code, it generally increases the number of classes that must be maintained.

The Repository pattern helps to isolate both the service and the list access code. Isolation makes it easier to treat them as independent services and to replace them with mock objects in unit tests. Typically, it is difficult to unit test the repositories themselves, so it is often better to write integration tests for them.

When caching data in a multithreaded environment, consider synchronizing access to the cache in addition to the cached objects. Often, common caches, such as the ASP.NET cache, are already thread safe, but you must also ensure that the objects themselves can operate in a multithreaded environment.

If you are caching data in heavily loaded systems, performance can be an issue. Consider synchronizing access to the data source. This ensures that only a single request for the data is issued to the list or back-end service. All other clients rely on the retrieved data. For more information, see Techniques for Aggregating List and Site Information.

Related Patterns

The following two patterns are often used in conjunction with the Repository pattern:

- Data Mapper. This pattern describes how to map data to different schemas. It is often used to map between a data store and a domain model.
- Unit of Work. This pattern keeps track of everything that happens during a business transaction that affects the database. At the conclusion of the transaction, it determines how to update the database to conform to the changes.

Home page on MSDN | Community site